**3.  Longest Palindromic Substring**

**def \_longestPalindromeSubseq(self, s):**

**if not s or len(s) == 0:**

**return 0**

**dp = [[0] \* len(s) for \_ in range(len(s))]**

**for i in range(len(s)):**

**dp[i][i] = 1**

**max\_len = 1**

**for l in range(2, len(s) + 1):**

**for i in range(len(s) - l + 1):**

**j = i + l - 1**

**if s[i] == s[j]:**

**if l == 2:**

**dp[i][j] = 2**

**else:**

**dp[i][j] = 2 + dp[i+1][j-1]**

**else:**

**dp[i][j] = max(dp[i+1][j], dp[i][j-1])**

**max\_len = max(max\_len, dp[i][j])**

**return max\_len**

**4. Product of Array Except Self**

**1. Use resulting array to safe product of from 1 to I:product[i] = product[i-1]\*nums[i-1]**

**product[0] = 1**

**2. Go from product[i] = product[i]\* right using right as product value from len-1 to I and right = right \* nums[i+1]**

**class Solution:**

**def productExceptSelf(self, nums):**

**if not nums:**

**return nums**

**# initial products**

**res = [1] \* len(nums)**

**left\_product = 1**

**# go from left, multiply bu left\_product - product without current item**

**# where res[i] is product of the left items[0:i]**

**for i in range(len(nums)):**

**res[i] \*= left\_product**

**left\_product \*= nums[i]**

**right\_product = 1**

**for i in range(len(nums) - 1, -1, -1):**

**res[i] \*= right\_product**

**right\_product \*= nums[i]**

**return res**

**5.Find the Duplicate Number**

* **Sort the items and find same adjacent items.**

**class Solution:**

**def findDuplicate(self, nums: List[int]) -> int:**

**t = nums[0]**

**h = nums[0]**

**while True:**

**t = nums[t]**

**h = nums[nums[h]]**

**if t == h:**

**break**

**t = nums[0]**

**while t != h:**

**h = nums[h]**

**t = nums[t]**

**return h**

**6 Intersection of Two Linked Lists**

**P1 = a+c+b**

**P2 = b+c+a**

**class Solution:**

**def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:**

**p1 = headA**

**p2 = headB**

**while p1 != p2:**

**if not p1:**

**p1 = headB**

**else:**

**p1 = p1.next**

**if not p2:**

**p2 = headA**

**else:**

**p2 = p2.next**

**return p1**

**7 Symmetric Tree**

**Create method for x and y to compare x.left and y.right and x.right and y.left. To pass (root, root) as first parameters to this method.**

**class Solution:**

**class Solution:**

**def isSymmetric(self, root):**

**if not root:**

**return True**

**def traverse(left\_tree, right\_tree):**

**if not left\_tree and not right\_tree:**

**return True**

**if not left\_tree and right\_tree or left\_tree and not right\_tree or left\_tree.val != right\_tree.val:**

**return False**

**return traverse(left\_tree.left, right\_tree.right) and traverse(left\_tree.right, right\_tree.left)**

**return traverse(root, root)**

**8.Binary Tree Vertical Order Traversal**

**from collections import defaultdict**

**class Solution:**

**def verticalOrder(self, root):**

**if not root:**

**return []**

**m = defaultdict(list)**

**q = [[root,0]]**

**while q:**

**node,col = q.pop(0)**

**m[col].append(node.val)**

**if node.left:**

**q.append([node.left, col - 1])**

**if node.right:**

**q.append([node.right, col + 1])**

**res = []**

**for col in sorted(m.keys()):**

**res.append(m[col])**

**return res**

**Graph Valid Tree:**

**Use union-find algorithm with rank and path-compression**

**class Solution:**

**def find(self, parent, i):**

**if parent[i] != i:**

**return self.find(parent, parent[i])**

**return parent[i]**

**def union(self, parent, rank, x, y):**

**xroot = self.find(parent,x)**

**yroot = self.find(parent, y)**

**if rank[xroot] < rank[yroot]:**

**parent[xroot] = yroot**

**elif rank[xroot] > rank[yroot]:**

**parent[yroot] = xroot**

**else:**

**parent[yroot] = xroot**

**rank[xroot] += 1**

**def validTree(self, n, edges):**

**if n == 0:**

**return False**

**parent = [i for i in range(n)]**

**rank = [0] \* n**

**for edge in edges:**

**#find parent for both vertices**

**x = self.find(parent, edge[0])**

**y = self.find(parent, edge[1])**

**#if it's cycle let's return False**

**if x == y:**

**return False**

**self.union(parent, rank, edge[0], edge[1])**

**return len(edges) == n - 1**

**9.Alien Dictionary**

Use topological sort.

**import collections**

**class Solution:**

**def alienOrder(self, words):**

**if not words:**

**return ''**

**#it's for topological sort**

**indegree = Counter({c:0 for word in words for c in word})**

**adj\_lst = defaultdict(set)**

**for w1,w2 in zip(words,words[1:]):**

**for c,d in zip(w1,w2):**

**if c != d:**

**# c is less then d**

**if d not in adj\_lst[c]:**

**adj\_lst[c].add(d)**

**# c -> d, increase indegree of d**

**indegree[d] += 1**

**# it doesn't consider the rest of chars**

**break**

**#if all chars are the same w1 and w2 then this words may differ by length**

**else:**

**#if w2 is substring of w1**

**if len(w2) < len(w1):**

**return ''**

**#take chars that don't have incoming edges**

**q = [c for c in indegree if indegree[c] == 0]**

**res = []**

**while q:**

**c = q.pop(0)**

**res.append(c)**

**for d in adj\_lst[c]:**

**indegree[d] -= 1**

**if indegree[d] == 0:**

**#it it doesn't exist more edges d is our next candidate**

**q.append(d)**

**#if not whole dict is represented,then it's failure**

**if len(res) < len(indegree):**

**return ''**

**return ''.join(res)**

**Kth Largest Element in an Array**

**class Solution:**

**def findKthLargest(self, nums, K):**

**if not nums or K > len(nums):**

**return None**

**def parition(s, e):**

**i = s**

**x = nums[e]**

**for j in range(s, e):**

**if nums[j] > x:**

**nums[i], nums[j] = nums[j], nums[i]**

**i += 1**

**nums[e], nums[i] = nums[i], nums[e]**

**return i**

**def quick\_select(s, e, k):**

**if s == e:**

**return nums[s]**

**p = parition(s, e)**

**if p == k:**

**return nums[p]**

**if p > k:**

**return quick\_select(s, p - 1, k)**

**return quick\_select(p + 1, e, k)**

**return quick\_select(0, len(nums) - 1, K-1)**

**WordLadder2**

**class Solution:**

**def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) -> List[List[str]]:**

**if not wordList or len(wordList[0]) == 0:**

**return []**

**adj\_list = defaultdict(list)**

**def find\_neighbors(word, word\_list):**

**nei = []**

**chars = list(word)**

**for i,ch in enumerate(chars):**

**prev = ch**

**for c in range(ord('a'), ord('z') + 1):**

**chars[i] = chr(c)**

**new\_word = ''.join(chars)**

**if c == prev or new\_word not in word\_list:**

**continue**

**nei.append(new\_word)**

**chars[i] = prev**

**return nei**

**def backtrack(adj\_list, src, dest, cur\_path, shortest\_path):**

**if src == dest:**

**shortest\_path.append(cur\_path[::])**

**if src not in adj\_list:**

**return**

**for w in adj\_list[src]:**

**cur\_path.append(w)**

**backtrack(adj\_list, w, dest, cur\_path, shortest\_path)**

**cur\_path.pop()**

**def bfs(adj\_list, begin\_word, end\_word,word\_list):**

**q = [begin\_word]**

**if begin\_word in word\_list:**

**word\_list.discard(begin\_word)**

**proc = set()**

**proc.add(begin\_word)**

**while q:**

**visited = set()**

**n = len(q)**

**for i in range(n):**

**cur\_word = q.pop(0)**

**nei = find\_neighbors(cur\_word, word\_list)**

**for w in nei:**

**visited.add(w)**

**adj\_list[cur\_word].append(w)**

**if w not in proc:**

**q.append(w)**

**proc.add(w)**

**for w in visited:**

**if w in word\_list:**

**word\_list.discard(w)**

**word\_list = set(wordList)**

**bfs(adj\_list, beginWord, endWord, word\_list)**

**cur\_path = [beginWord]**

**shortest\_path = []**

**backtrack(adj\_list, beginWord, endWord, cur\_path, shortest\_path)**

**return shortest\_path**

**Best Time Buy Sell Stock**

**def maxProfit(prices: Array[Int]): Int = {**

**if (prices.isEmpty) 0**

**else {**

**var maxprofit = 0**

**var minprice = Int.MaxValue**

**for (i <- 0 to prices.length - 1) {**

**if (minprice > prices(i)) {**

**minprice = prices(i)**

**}**

**else if ((prices(i) - minprice) > 0) {**

**maxprofit = (prices(i) - minprice) max maxprofit**

**}**

**}**

**maxprofit**

**}**

**}**

class HitCounter:

def \_\_init\_\_(self):

self.hits = [0] \* 300

self.seconds = [-1] \* 300

def hit(self, timestamp: int) -> None:

i = (timestamp - 1) % 300

if self.seconds[i] != timestamp:

self.hits[i] = 1

self.seconds[i] = timestamp

else:

self.hits[i] += 1

def getHits(self, timestamp: int) -> int:

res = 0

# we consider recent the 300 seconds

#because we divide by module 300 we will subtract 300

for i in range(len(self.hits)):

if (timestamp - 300) < self.seconds[i] <= timestamp:le

res += self.hits[i]

return res

Binary Search Tree Iterator

class BSTIterator:

def \_\_init\_\_(self, root: TreeNode):

self.st = []

self.add(root)

def next(self) -> int:

t = self.st.pop()

self.add(t.right)

return t.val

def hasNext(self) -> bool:

return self.st

def add(self, node):

while node:

self.st.append(node)

node = node.left

Find a celebrity

public int findCelebrity(int n) {  
 int canditate = 0;  
 for(int i = 1; i< n;i++){  
 if(knows(canditate,i)){  
 canditate = i;  
 }  
 }  
 for(int i = 0; i< n;i++){  
 if((i!=canditate) &&(knows(canditate,i) || !knows(i, canditate))) return -1;  
 }  
 return canditate;  
}

Minimum Window substring

from collections import Counter

from collections import defaultdict

class Solution:

def minWindow(self, s, t):

if not s and not t:

return t

if not s or not t:

return ""

min\_wind = [float('inf'), None, None]

# chars from t

chars = Counter(t)

# number of uniq chars in t

need = len(chars)

# current window

wind = defaultdict(int)

left = 0

#number of formeed chars in current windows

formed = 0

for right in range(len(s)):

ch = s[right]

#update window

wind[ch] += 1

if ch in chars and wind[ch] == chars[ch]:

formed += 1

# if number of unique chars in current window as required, let's decrease left boundary of current window

while left <= right and formed == need:

if min\_wind[0] > right - left + 1:

min\_wind = [right - left + 1, left, right]

#remove leftmost char from the current window

wind[s[left]] -= 1

# if freq of char is decreased,let's update

if s[left] in chars and wind[s[left]] < chars[s[left]]:

formed -= 1

left += 1

formed, l, r = min\_wind

return s[l: r + 1] if formed != float('inf') else ‘'

Generate Parentheses

class Solution:

def generateParenthesis(self, n):

assert self.\_generateParenthesis(0) == [], 'test1'

return self.\_generateParenthesis(n)

def \_generateParenthesis(self, n):

if n <= 0:

return []

def backtrack(res, st,pos,open\_paren,closed\_paren):

if closed\_paren == n:

res.append(''.join(st))

return

if closed\_paren < open\_paren:

st.append(')')

backtrack(res, st, pos+1, open\_paren, closed\_paren + 1)

st.pop()

if open\_paren < n:

st.append('(')

backtrack(res, st, pos+1, open\_paren + 1,closed\_paren)

st.pop()

paren = []

backtrack(paren, [], 0, 0, 0)

return paren

**Evaluate Division**

**from collections import defaultdict**

**class Solution:**

**def calcEquation(self, equations, values, queries):**

**adj\_list = defaultdict(list)**

**for ab,w in zip(equations, values):**

**a,b = ab**

**adj\_list[a].append([w, b])**

**adj\_list[b].append([1 / w, a])**

**def dfs(adj\_list, a,b,seen):**

**seen.add(a)**

**for w,v in adj\_list[a]:**

**if v == b:**

**return w**

**if v in seen:**

**continue**

**t = w \* dfs(adj\_list, v, b, seen)**

**if t >= 0.0:**

**return t**

**return -1.0**

**res = []**

**for c,d in queries:**

**t = dfs(adj\_list, c, d, set())**

**res.append(t)**

**return res**

**Decode Ways**

1. **dp[0:i] = dp[i-1] + dp[i-2] if s[i-1] != 0 and 10 <= s[i-2:i] <= 26**

**class Solution:**

**def numDecodings(self, s: str) -> int:**

**dp0 = 1**

**dp1 = 0**

**if s[0] != '0':**

**dp1 = 1 # 0 in s**

**for j in range(2, len(s) + 1):**

**# if curent char is 0, then we can count approach to reach it**

**new\_dp = 0**

**if s[j-1] != '0':**

**new\_dp = dp1**

**if 10 <= int(s[j-2:j]) <= 26:**

**new\_dp += dp0**

**dp0 = dp1**

**dp1 = new\_dp**

**return dp1**

**Task Scheduler**

**class Solution:**

**def leastInterval(self, tasks: List[str], n: int) -> int:**

**freq = [0] \* 26**

**for ch in tasks:**

**freq[ord(ch) - ord('A')] += 1**

**freq.sort()**

**f\_max = freq.pop()**

**#group number**

**idle\_slots = (f\_max-1) \* n**

**while freq and idle\_slots > 0:**

**f = freq.pop()**

**#min(f\_max-1, f) if f = f\_max, we take min of two**

**idle\_slots -= min(f\_max - 1, f)**

**return len(tasks) + max(idle\_slots,0)**

**Subarray Sum Equals K**

**from collections import defaultdict**

**class Solution:**

**def subarraySum(self, nums: List[int], k: int) -> int:**

**cur\_sum = 0**

**dp = defaultdict(int)**

**dp[0] = 1**

**res = 0**

**for a in nums:**

**cur\_sum += a**

**if cur\_sum -k in dp:**

**res += dp[cur\_sum - k]**

**dp[cur\_sum] += 1**

**return res**

**Validate IP Address**

**class Solution:**

**def check\_ip4(self,ip):**

**nums = ip.split('.')**

**for x in nums:**

**if len(x) == 0 or len(x) > 3 or x[0] == '0' and len(x) != 1 or not x.isdigit() or int(x) > 255:**

**return 'Neither'**

**return 'IPv4'**

**def check\_ip6(self, ip):**

**nums = ip.split(':')**

**hex = '0123456789abcdefABCDEF'**

**for x in nums:**

**if len(x) == 0 or len(x) > 4 or not all(c in hex for c in x):**

**return 'Neither'**

**return 'IPv6'**

**def validIPAddress(self, IP):**

**if IP.count('.') == 3:**

**return self.check\_ip4(IP)**

**if IP.count(':') == 7:**

**return self.check\_ip6(IP)**

**return 'Neither'**

**Game of Life**

**class Solution(object):**

**def gameOfLife(self, board):**

**n = len(board)**

**m = len(board[0])**

**dirs = [(1,0), (1,-1), (0,-1), (-1,-1), (-1,0), (-1,1), (0,1), (1,1)]**

**for i in range(n):**

**for j in range(m):**

**#count live cells for cell**

**live\_cells = 0**

**for dr,dc in dirs:**

**nr = i + dr**

**nc = j + dc**

**if 0 <= nr < n and 0 <= nc < m and abs(board[nr][nc]) == 1:**

**live\_cells += 1**

**if board[i][j] == 1 and (live\_cells < 2 or live\_cells > 3):**

**#to prevent counting new live cells**

**board[i][j] = -1**

**if board[i][j] == 0 and live\_cells == 3:**

**board[i][j] = 2**

**#decode new live cells back**

**for i in range(n):**

**for j in range(m):**

**if board[i][j] > 0:**

**board[i][j] = 1**

**else:**

**board[i][j] = 0**

**Permutations**

**1.create the function backtrack(first):**

**- check if first = len of arr then it returns count**

**- iterate from first to end:**

**a) swap i and first and call backtrack recursively with first+1**

**b) swap i and first back**

**class Solution(object):**

**def permute(self, nums):**

**"""**

**:type nums: List[int]**

**:rtype: List[List[int]]**

**"""**

**def perm(first, buf):**

**if first == len(nums):**

**buf.append(nums[::])**

**return**

**for j in range(first,len(nums)):**

**nums[first],nums[j] = nums[j],nums[first]**

**perm(first + 1, buf)**

**nums[first],nums[j] = nums[j],nums[first]**

**all\_perms = []**

**perm(0, all\_perms)**

**return all\_perms**

**Reverse Words in a String**

**class Solution:**

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

**if not s:**

**return s**

**s = list(s.strip())**

**def reverse\_str(lst, start,end):**

**while start < end:**

**lst[start],lst[end] = lst[end],lst[start]**

**start += 1**

**end -= 1**

**reverse\_str(s, 0, len(s) - 1)**

**start = 0**

**for i in range(len(s)):**

**if s[i] == ' ':**

**if start < i - 1 and s[start] != ' ':**

**reverse\_str(s, start, i - 1)**

**start = i+1**

**reverse\_str(s, start, len(s)-1)**

**words = ''.join(s).split(' ')**

**s = []**

**for word in words:**

**word = word.strip()**

**if word:**

**s.append(word)**

**return ' '.join(s)**

>>>**. Word Ladder**

**1. Use BFS for search endWord from beginWord**

**2. Store visited words to a dict to prevent repeated using of it.**

**3. Store in a queue a tuple (word,length)**

**4. When adding new neighbours to a queue it add theirs to visited dict and check if added word is endWord.**

380. [Insert Delete GetRandom O(1)](https://leetcode.com/problems/insert-delete-getrandom-o1)

**1. Declare an array for values and a map for value and its position in the array.**

**2. While inserting new value it has to check if new value already exists. If not exist it will add this value to the end of this array**

**3.While removing a value it swaps last item with the value and remove last item in the array and the value in the map.**

**79. Word Search**

1. Use backtracking to check if a word in list.

2. if backtrack function it checks if word in position equals a character in the matrix. If so it calls this with i+1,i-1,j-1, j+1 characters in matrix by adding the used characters in seen before each recursive call and then it removes it later.

3. In main function it iterates through all characters and call the backtracking for each character.

FriendCircles

1. Use FindUnion with path compression and rank.
2. If m[i][j] == 1 or m[p1]m[p2] == 1 let’s union friends
3. Filter out the unique parent values. There are circle size

Word Frequencies

cat words.txt | tr -s ' ' '\n' | awk '{nums[$1]++}END{for(word in nums) print word, nums[word]}' | sort -rn -k2

**923. 3Sum With Multiplicity**

**1. Use 2 pointer technic**

**2. Sort array, take i-th item, and look for target – a[i] from i + 1 to len(arr)-1.**

**3. if a[j] + a[k] == target – a[i] it has 2 cases:**

**3.1 if a[j] == a[k] num = (k – j + 1)\*(k-j) / 2**

**3.2 in another case left is count of a[j] and right is count of a[k]:**

**while j + 1 < k and arr[j] == arr[j + 1]:**  
 **left += 1**  
 **j += 1**  
**right = 1**  
**while k - 1 > j and arr[k] == arr[k - 1]:**  
 **right += 1**  
 **k -= 1**  
**num += left \* right**

* 1. **num += left \* right**

**394. Decode String**

**1. recursive decent**

**class Decoded:**  
 **def \_\_init\_\_(self):**  
 **self.pos = 0**  
 **self.res = ''**  
  
  
**class Solution:**  
 **def is\_digit(self, ch):**  
 **return '0' <= ch <= '9'**  
  
 **def is\_char(self, ch):**  
 **return 'a' <= ch <= 'z' or 'A' <= ch <= 'Z'**  
  
 **def str(self, s, decoded):**  
 **if decoded.pos == len(s):**  
 **return ""**  
 **if self.is\_char(s[decoded.pos]):**  
 **j = decoded.pos**  
 **while decoded.pos < len(s) and self.is\_char(s[decoded.pos]):**  
 **decoded.pos += 1**  
 **decoded.res += s[j: decoded.pos]**  
  
 **def rep\_or\_str(self, s, decoded):**  
 **if decoded.pos == len(s):**  
 **return ""**  
 **if self.is\_char(s[decoded.pos]):**  
 **self.str(s, decoded)**  
 **return**  
 **j = decoded.pos**  
 **while decoded.pos < len(s) and self.is\_digit(s[decoded.pos]):**  
 **decoded.pos += 1**  
 **k = int(s[j: decoded.pos])**  
 **decoded.pos += 1**  
 **prev = decoded.pos**  
 **for i in range(k):**  
 **decoded.pos = prev**  
 **while s[decoded.pos] != ']':**  
 **self.str(s, decoded)**  
 **if self.is\_digit(s[decoded.pos]):**  
 **self.rep\_or\_str(s, decoded)**  
 **elif self.is\_char(s[decoded.pos]):**  
 **self.str(s, decoded)**  
 **decoded.pos += 1**  
  
 **def decodeString(self, s: str) -> str:**  
 **if not str:**  
 **return ""**  
 **decoded = Decoded()**  
 **while decoded.pos < len(s):**  
 **self.rep\_or\_str(s, decoded)**  
 **return decoded.res**

**528. Random Pick with Weight**

**1. Consider array of weight sums: w[0], w[0] + w[1],…**

**2. Take random value from the interval [0, total)**

**3. Search least sum > random value using binary search:**

**While lo != hi:**

**If a >= array[mid] lo = mid + 1**

**Else hi = mid – 1**

**Return lo**

**755. Pour Water**

**1. Go to the left to look up the leftmost min:**

**While height[i+d] <= height[i]**

**2. Do the same for the right side.**

**3. in another case to increment height[k]:**

**class Solution:**  
 **def find\_min(self, heights, v, d):**  
 **i = v**  
 **res = -1**  
 ***#look up leftmost/rightmost min***  
**while 0 <= (i + d) < len(heights) and heights[i + d] <= heights[i]:**  
 **if heights[i + d] < heights[i]:**  
 **res = i + d**  
 **i += d**  
 **if res != -1:**  
 **return res**  
 **return -1**  
  
 **def pourWater(self, heights, V: int, K: int):**  
 **if not heights or V == 0:**  
 **return heights**  
 **for v in range(V):**  
 **left = self.find\_min(heights, K, -1)**  
 **if left >= 0:**  
 **heights[left] += 1**  
 **else:**  
 **right = self.find\_min(heights, K, 1)**  
 **if right >= 0:**  
 **heights[right] += 1**  
 **else:**  
 **heights[K] += 1**  
  
 **return heights**

**973. K Closest Points to Origin**

**1. Use quick\_select to find k-th closest point to origin**

**2. partition uses forward and backward iterations**

**3. quick\_select uses mid - r + 1 and check if k < or > mid – l + 1 then choose the part having K. For right part it changes K = K – (mid –l +1) = K –mid + l -1**

**103. Binary Tree Zigzag Level Order Traversal**

**1. Use recursive helper function to traverse tree in order.**

**2. Use map as function parameter and level -> list**

**3. Function calling itself for left child and right child passes map and level + 1.**

**4. If level is even it appends node.val to list looking up in map by level. In other case it inserts node.val at the begging of list.**

**332. Reconstruct Itinerary**

**1. Put a ticket departure as key and list of indexes of tickets to map. Sort a list in lexical order.**

**2. Use dfs to walk around all tickets to find unique path from all tickets so dfs takes one of parameters used\_tickets and other parameters are dep, seen and path.**

**3. Call dfs until use all of tickets for a path**

**4. Reuse path in the same recursive level by reseting the path after calling of dfs.**

**347. Top K Frequent Elements**

**1. Make map element to its frequency**

**2. Create min heap of size k and if it reaches size k then it remove min element if it has less frequency than adding one has.**

**Search in rotated Sorted array**

1. **Use binary search**
2. **If leftmost <= middle and target is out of left half – it returns right half**
3. **If middle <= rightmost and target is out of the right half – it returns left half**
4. **If leftmost <= middle – it returns left half**
5. **If middle <= rightmost – it returns right half**

**947. Most Stones Removed with Same Row or Column**

**1. Use disjoin set union to store each x and y coordinate separately –**

**Let’s y will be in 10000+ y. if maximum coordinate is 10000**

**2. stones may hold in one component**

**3. So union x,y coordinates of stones**

**4. max moves = number of stones – number of components. In each component is left one stone so number of components are number of left stones.**

**341. Flatten Nested List Iterator**

**class NestedIterator:**  
 **def \_\_init\_\_(self, lst):**  
 **self.q = []**  
 **self.flatten(lst)**  
  
 **def flatten(self, lst):**  
 **for ni in lst:**  
 **if ni.isInteger():**  
 **self.q.append(ni.getInteger())**  
 **else:**  
 **self.flatten(ni.getList())**  
  
 **def next(self):**  
 **a = self.q[0]**  
 **self.q.pop(0)**  
 **return a**  
  
 **def hasNext(self):**  
 **return len(self.q)**

**692. Top K Frequent Words**

**1. Count word frequency in map**

**2. Use priority queue to get frequency minimum or lower lexicographical word if frequencies are equal**

**3.Add word to list and reverse list**

**Course Schedule**

1. **Create graph as adjacent list, list visit is with 3 states: 0 is if vertex is not seem,1 is if way is seen, -1 is if vertex in cycle. We can mark the course we seen as visited and use it in the next time.**
2. **Before each traversation we mark vertex as cycle part then as seen.**
3. **Use dfs to traverse graph**
4. **def canFinish(self, numCourses: int, prerequisites):**  
    **if not numCourses:**  
    **return False**  
    **graph = [[] for \_ in range(numCourses)]**  
    **visit = [0 for i in range(numCourses)]**  
    **for i in range(len(prerequisites)):**  
    **graph[prerequisites[i][0]].append(prerequisites[i][1])**  
     
    **def dfs(v, graph, visit):**  
    **if visit[v] == 1:**  
    **return True**  
    **if visit[v] == -1:**  
    **return False**  
    **visit[v] = -1**  
    **for u in graph[v]:**  
    **if not dfs(u, graph, visit):**  
    **return False**  
    **visit[v] = 1**  
    **return True**  
     
    **for v in range(len(graph)):**  
    **if not dfs(v, graph, visit):**  
    **return False**  
    **return True**

**895. Maximum Frequency Stack**

**1. Create map value to frequency, map frequency to stack**

**2. If we push value we will add value to map : a -> (frequency + 1). If frequency > maxFrequency, we will update maxFrequency and put maxFrequency -> stack, push value to stack**

**3. When we pop value, we pop value from stack in second map, and decsrease value frequency in first map.**

1. **ZigZag Conversion**

**def convert(self, s, numRows):**  
 **if not s or not numRows:**  
 **return s**  
 **r = 0**  
 **i = 0**  
 **m = defaultdict(list)**  
 **while i < len(s):**  
 **while r < numRows and i < len(s):**  
 **m[r].append(s[i])**  
 **i += 1**  
 **r += 1**  
 **r -= 2**  
 **while r > 0 and i < len(s):**  
 **m[r].append(s[i])**  
 **i += 1**  
 **r -= 1**  
 **r = 0**  
 **res = ''**  
 **for i in range(numRows):**  
 **for j in range(len(m[i])):**  
 **res += m[i][j]**  
 **return res**

**BasicCalculator||**

**class Solution:**

**def prec(self, op):**

**if op == '+' or op == '-':**

**return 0**

**return 1**

**def eval(self, stack, ops):**

**while len(stack) > 0:**

**zn = stack.pop()**

**a = ops.pop()**

**b = ops.pop()**

**if zn == '+':**

**ops.append(a + b)**

**elif zn == '-':**

**ops.append(b - a)**

**elif zn == '\*':**

**ops.append(a \* b)**

**else:**

**ops.append(b // a)**

**return ops.pop()**

**def postfix(self, s):**

**j = 0**

**stack = []**

**buf = []**

**while j < len(s):**

**if '0' <= s[j] <= '9':**

**i = j**

**while i < len(s) and '0' <= s[i] <= '9':**

**i+= 1**

**buf.append(s[j:i])**

**j = i**

**else:**

**if s[j] in ['+','-','\*','/']:**

**while len(stack) > 0 and self.prec(s[j]) <= self.prec(stack[len(stack) - 1]):**

**buf.append(stack.pop())**

**stack.append(s[j])**

**j += 1**

**while len(stack) > 0:**

**buf.append(stack.pop())**

**return buf**

**def calculate(self, s):**

**if not s:**

**return None**

**stack = []**

**postfix = self.postfix(s)**

**i = 0**

**while i < len(postfix):**

**x = postfix[i]**

**if x.isdecimal():**

**stack.append(int(x))**

**else:**

**a = stack.pop()**

**b = stack.pop()**

**if x == '+':**

**stack.append(a + b)**

**elif x == '-':**

**stack.append(b - a)**

**elif x == '\*':**

**stack.append(a \* b)**

**else:**

**stack.append(b // a)**

**i += 1**

**return stack.pop()**

**974. Subarray Sums Divisible by K**

**Let there be a subarray (i, j) whose sum is divisible by k**

**sum(i, j) = sum(0, j) - sum(0, i-1)**

**Sum for any subarray can be written as q\*k + rem where q**

**is a quotient and rem is remainder**

**Thus,**

**sum(i, j) = (q1 \* k + rem1) - (q2 \* k + rem2)**

**sum(i, j) = (q1 - q2)k + rem1-rem2**

**We see, for sum(i, j) i.e. for sum of any subarray to be**

**divisible by k, the RHS should also be divisible by k.**

**(q1 - q2)k is obviously divisible by k, for (rem1-rem2) to**

**follow the same, rem1 = rem2 where**

**rem1 = Sum of subarray (0, j) % k**

**rem2 = Sum of subarray (0, i-1) % k**

1. **v \* (v - 1) // 2 - (n choose 2) ways to combine 2 subarrays.**
2. **class Solution:**  
    **def subarraysDivByK(self, arr, k):**  
    **if not arr or len(arr) == 0 or k == 0:**  
    **return 0**  
    **p = [0]**  
    **for x in arr:**  
    **p.append((p[-1] + x) % k)**  
     
    **freq = Counter(p)**  
    **return int(sum(v \* (v - 1) // 2 for v in freq.values()))**

**12. Integer to Roman**

1. **Add all roman number to map decimal to roman numbers**
2. **Find greatest minimum number and subtract it from current decimal number by appending roman to result.**
3. **class Solution:**  
    **def find\_greater\_min(self, num):**  
    **if 1 <= num < 4:**  
    **return 1**  
    **if 4 <= num < 5:**  
    **return 4**  
    **if 5 <= num < 9:**  
    **return 5**  
    **if 9 <= num < 10:**  
    **return 9**  
    **if 10 <= num < 40:**  
    **return 10**  
    **if 40 <= num < 50:**  
    **return 40**  
    **if 50 <= num < 90:**  
    **return 50**  
    **if 90 <= num < 100:**  
    **return 90**  
    **if 100 <= num < 400:**  
    **return 100**  
    **if 400 <= num < 500:**  
    **return 400**  
    **if 500 <= num < 900:**  
    **return 500**  
    **if 900 <= num < 1000:**  
    **return 900**  
    **return 1000**  
     
    **def intToRoman(self, num):**  
    **if num <= 0:**  
    **return ''**  
    **roman = {}**  
    **roman[1] = 'I'**  
    **roman[4] = 'IV'**  
    **roman[5] = 'V'**  
    **roman[9] = 'IX'**  
    **roman[10] = 'X'**  
    **roman[40] = 'XL'**  
    **roman[50] = 'L'**  
    **roman[90] = 'XC'**  
    **roman[100] = 'C'**  
    **roman[500] = 'D'**  
    **roman[400] = 'CD'**  
    **roman[900] = 'CM'**  
    **roman[1000] = 'M'**  
    **res = ''**  
    **while num > 0:**  
    **d = self.find\_greater\_min(num)**  
    **res += roman[d]**  
    **num -= d**  
    **return res**

#### Find first and last position of element in sorted array

#### Use binary search :

1. **def find\_least(self, nums, target):**  
    **s = 0**  
    **e = len(nums) - 1**  
    **res = -1**  
    **while s <= e:**  
    **mid = s + (e - s) // 2**  
    **if nums[mid] < target:**  
    **s = mid + 1**  
    **else:**  
    **if nums[mid] == target:**  
    **res = mid**  
    **e = mid - 1**  
    **return res**  
     
   **def find\_greatest(self, nums, target):**  
    **s = 0**  
    **e = len(nums) - 1**  
    **res = -1**  
    **while s <= e:**  
    **mid = s + (e - s) // 2**  
    **if nums[mid] > target:**  
    **e = mid - 1**  
    **else:**  
    **if nums[mid] == target:**  
    **res = mid**  
    **s = mid + 1**  
    **return res**  
     
   **def searchRange(self, nums, target):**  
    **if not nums:**  
    **return [-1, -1]**  
    **return [self.find\_least(nums, target), self.find\_greatest(nums, target)]**

**986. Interval List Intersections**

1.Use merge two sorted lists.

2. If lefts of interval are equal then merge each of them with other list and increment indices by 1

3. If one of interval is less just merge it with other interval list and increment its index by 1

**Validate Binary Search Tree**

1. Traverse tree in-order: left subtree, root, right subtree and add remember last node value and check if invariant is keeping: last value is less than current node value.Also update last variable before running traverse for right subtree.

**Add Two Numbers**

1. Sum digest from left to right. If sum is greater 10 to set up the variable carry to 1.

**535. Encode and Decode TinyURL**

class Codec:

def \_\_init\_\_(self):

self.key = [0] \* 6

self.encodedToUrl = {}

def next(self):

arr = [0] \* len(self.key)

for i in range(len(self.key)):

arr[i] = self.key[i] + 65

for i in range(len(self.key)):

if arr[i] < 63:

arr[i] += 1

return ''.join(map(chr, arr))

def encode(self, longUrl):

*"""Encodes a URL to a shortened URL.*

***:type*** *longUrl: str*

***:rtype****: str*

*"""*

if not longUrl:

return ''

key = self.next()

self.encodedToUrl[key] = longUrl

return 'http://tinyurl.com/{}'.format(key)

def decode(self, shortUrl):

*"""Decodes a shortened URL to its original URL.*

***:type*** *shortUrl: str*

***:rtype****: str*

*"""*

if not shortUrl:

return None

parts = shortUrl.split('/')

encoded = parts[len(parts) - 1]

if encoded in self.encodedToUrl:

return self.encodedToUrl[encoded]

return None

codec = Codec()

print(codec.encode('https://leetcode.com/problems/design-tinyurl'))

print(codec.decode(codec.encode('https://leetcode.com/problems/design-tinyurl')))

**Convert Binary Search Tree to Sorted Doubly Linked List**

1. Use in-order to convert tree to double-linked list.

2. Declare first and last, when it traverse left subtree let's assign to last current node.

If last == None let’s assign first to current node. If we have last we update last.right = node and node.left to last

class Solution(object):

def treeToDoublyList(self, root):

def traverse(node):

nonlocal first, last

if node:

traverse(node.left)

if last:

node.left = last

last.right = node.left

else:

first = node

last = node

traverse(node.right)

first, last = None, None

traverse(root)

last.right = first

first.left = last

return first

**177. Nth Highest Salary**

**CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT**

**BEGIN**

**DECLARE X INT;**

**SET X = (SELECT MAX(salary) FROM employee);**

**WHILE N > 1 DO**

**SET X = (SELECT MAX(salary) FROM employee WHERE salary < X);**

**SET N = N - 1;**

**END WHILE;**

**RETURN (X);**

**END**

CloneGraph

1. Use DFS to clone graph
2. If we meet node again then take it from map

class Solution:

def cloneGraph(self, node):

if not node:

return node

def clone(node, nodes):

if not node:

return None

if node.val in nodes:

return nodes[node.val]

cloned = Node(node.val, [])

nodes[cloned.val] = cloned

for x in node.neighbors:

new\_x = clone(x, nodes)

cloned.neighbors.append(new\_x)

return cloned

return clone(node, {})

**Minesweeper**

class Solution:

def mines(self, board, row, col):

mines = 0

if row - 1 >= 0 and board[row - 1][col] in 'MX':

mines += 1

if row + 1 < len(board) and board[row + 1][col] in 'MX':

mines += 1

if col - 1 >= 0 and board[row][col - 1] in 'MX':

mines += 1

if col + 1 < len(board[0]) and board[row][col + 1] in 'MX':

mines += 1

if row - 1 >= 0 and col - 1 >= 0 and board[row - 1][col - 1] in 'MX':

mines += 1

if row - 1 >= 0 and col + 1 < len(board[0]) and board[row - 1][col + 1] in 'MX':

mines += 1

if row + 1 < len(board) and col - 1 >= 0 and board[row+1][col-1] in 'MX':

mines += 1

if row + 1 < len(board) and col + 1 < len(board[0]) and board[row + 1][col + 1] in 'MX':

mines += 1

return mines

def updateBoard(self, board, click):

if not board or not click:

return

def reveal(board, row, col):

if board[row][col] == 'M':

board[row][col] = 'X'

if board[row][col] == 'E':

mines = self.mines(board, row, col)

if mines > 0:

board[row][col] = chr(ord('0') + mines)

else:

board[row][col] = 'B'

if mines > 0:

return

if row - 1 >= 0:

reveal(board, row - 1, col)

if row + 1 < len(board):

reveal(board, row + 1, col)

if col - 1 >= 0:

reveal(board, row, col - 1)

if col + 1 < len(board[0]):

reveal(board, row, col + 1)

if row - 1 >= 0 and col - 1 >= 0:

reveal(board, row - 1, col - 1)

if row - 1 >= 0 and col + 1 < len(board[0]):

reveal(board, row - 1, col + 1)

if row + 1 < len(board) and col - 1 >= 0:

reveal(board, row + 1, col - 1)

if row + 1 < len(board) and col + 1 < len(board[0]):

reveal(board, row + 1, col + 1)

reveal(board, click[0], click[1])

return board

**Restore IP Addresses**

Use backtracking.

Time complexity is O(27). 3 dots give 27 permutations.

Memory complexity is O(19).

class Solution:

def restoreIpAddresses(self, s):

if not s:

return s

def generate\_ip(part, i, ip, res):

if part == 4:

ip\_len = len(ip[0]) + len(ip[1]) + len(ip[2]) + len(ip[3])

if ip\_len >= len(s):

ip\_str = '.'.join(ip)

if len(res) > 0:

if res[-1] != ip\_str:

res.append(ip\_str)

else:

res.append(ip\_str)

return

for j in range(1, 4):

if (i + j) > len(s):

break

a = int(s[i: i + j])

if a <= 255:

ip[part] = repr(a)

generate\_ip(part + 1, i + j, ip, res)

res = []

ip = [0] \* 4

generate\_ip(0, 0, ip, res)

return res

Maximal Square

1. Use DP
2. dp[i][j] is length of square matrix with lower right corner

class Solution:

def maximalSquare(self, matrix):

if not matrix:

return 0

dp = [0] \* (len(matrix) + 1)

for i in range(len(dp)):

dp[i] = [0] \* (len(matrix[0]) + 1)

max\_len = 0

for i in range(1, len(dp)):

for j in range(1, len(dp[0])):

if matrix[i - 1][j - 1] == '1':

dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1

max\_len = max(max\_len, dp[i][j])

return max\_len \* max\_len

Longest Increasing Subsequence

1. Use DP. Try to add nums[i+1] to each subsequence [0, k] where k = [1, i]:

[0,1], [0,2],[0, k]

2. Allocate the array dp to store maximum subsequence length.

def lengthOfLIS(self, nums):

if not nums:

return 0

lis = [1] \* len(nums)

for i in range(1, len(nums)):

for j in range(0, i):

if nums[i] > nums[j] and lis[i] < lis[j] + 1:

lis[i] = lis[j] + 1

return max(lis)

**Increasing Triplet Subsequence**

1. **Find first and second in triplet.**

class Solution:

def increasingTriplet(self, nums):

if not nums or len(nums) < 3:

return False

small = sys.maxsize

large = sys.maxsize

for i in range(len(nums)):

if nums[i] <= small:

small = nums[i]

elif nums[i] <= large:

large = nums[i]

else:

return True

return False

**Maximum Length of Pair Chain**

1.Sort pairs by first item

2. Use algorithm as for LIS.

**Knight Dialer**

**1.Use DP to solve it.**

**2. Make map where key is digit and value at which digit I can go.**

**3. Allocate array 10 x N where N is step number.**

4. Num[i][j] = Num[i-1][k1] + Num[i-1][k2] + … where Nym

Design Log Storage System

1. Store log entry in tree map by converting timestamp to seconds
2. Set up all fields to 0 if them are not considered.
3. Increment granularity field for end value.
4. Use tailMap to get timestamp is greater or equal to start.

import java.util.\*;

public class LogSystem {

ArrayList < long[] > list;

public LogSystem() {

list = new ArrayList < long[] > ();

}

public void put(int id, String timestamp) {

int[] st = Arrays.*stream*(timestamp.split(":")).mapToInt(Integer::*parseInt*).toArray();

list.add(new long[] {convert(st), id});

}

public long convert(int[] st) {

st[1] = st[1] - (st[1] == 0 ? 0 : 1);

st[2] = st[2] - (st[2] == 0 ? 0 : 1);

return (st[0] - 1999L) \* (31 \* 12) \* 24 \* 60 \* 60 + st[1] \* 31 \* 24 \* 60 \* 60 + st[2] \* 24 \* 60 \* 60 + st[3] \* 60 \* 60 + st[4] \* 60 + st[5];

}

public List < Integer > retrieve(String s, String e, String gra) {

ArrayList < Integer > res = new ArrayList();

long start = granularity(s, gra, false);

long end = granularity(e, gra, true);

for (int i = 0; i < list.size(); i++) {

if (list.get(i)[0] >= start && list.get(i)[0] < end)

res.add((int) list.get(i)[1]);

}

return res;

}

public long granularity(String s, String gra, boolean end) {

HashMap < String, Integer > h = new HashMap();

h.put("Year", 0);

h.put("Month", 1);

h.put("Day", 2);

h.put("Hour", 3);

h.put("Minute", 4);

h.put("Second", 5);

String[] res = new String[] {"1999", "00", "00", "00", "00", "00"};

String[] st = s.split(":");

for (int i = 0; i <= h.get(gra); i++) {

res[i] = st[i];

}

int[] t = Arrays.*stream*(res).mapToInt(Integer::*parseInt*).toArray();

if (end)

t[h.get(gra)]++;

return convert(t);

}

}

**Count Complete Tree Nodes**

1. **Solve it with recursion: go to right subtree and go to the left subtree:**

**Def countNodes(tree):**

**Return 1 + countNodes(tree.right) + countNodes(tree.left) if root else 0**

**Find Minimum in Rotated Sorted Array**

1. **If first element is less than mid then it takes right part in other case it takes left part.**
2. **If elm[mid] > elm[mid+1] return mid+1. If elm[mid-1] > elm[mid] return mid**

class Solution:

def findMin(self, nums):

if not nums:

return

if len(nums) == 1:

return nums[0]

if nums[0] < nums[len(nums) - 1]:

return nums[0]

def find\_inflection\_index(s, e):

if s > e:

return s

mid = s + (e - s) // 2

if mid + 1 < len(nums) and nums[mid + 1] < nums[mid]:

return mid + 1

if nums[mid - 1] > nums[mid]:

return mid

if nums[0] < nums[mid]:

s = mid + 1

else:

e = mid - 1

return find\_inflection\_index(s, e)

ind = find\_inflection\_index(0, len(nums) - 1)

return nums[ind]

RotateList

1.Add items to stack

2.Pop from stack and insert in existent list. If length is odd to pop length/2+1 items and in other case to pop lenght/2 ones.

3.When popping new node then set up next to None.

class Solution:

def reorderList(self, head):

if not head or not head.next:

return

p = head

q = []

while p:

q.append(p)

p = p.next

p = head

count = len(q) // 2

if len(q) % 2 == 1:

count += 1

while count > 0:

last = q.pop()

last.next = None

t = p.next

p.next = last

last.next = t

p = t

count -= 1

if p:

p.next = None

**Binary Tree Right Side View**

1. **Traverse right subtree, then a left subtree.**
2. **Store depth as key and value in map if it doesn’t have such key yet.**

class Solution:

def rightSideView(self, root):

if not root:

return []

max\_depth = 0

def traverse(node, level, m):

nonlocal max\_depth

if not node:

return

max\_depth = max(max\_depth, level)

traverse(node.right, level + 1, m)

traverse(node.left, level + 1, m)

if level not in m:

m[level] = node.val

m = {}

traverse(root, 0, m)

return [m[depth] for depth in range(max\_depth+1)]

Custom Sort String:

1.All chars in S should be in T in the same order

2.

from collections import Counter

class Solution:

def customSortString(self, S, T):

if not S or not T:

return

chars = Counter(T)

res = []

for ch in S:

res.append(ch \* chars[ch])

chars[ch] = 0

for ch,count in chars.items():

res.append(ch \* count)

return ''.join(res)

**Partition Labels**

1. **Calculate rightmost char position in a string.**
2. **Let’s set start, end are current string**
3. **If last[char] < end then this char is inside current string, in other case**

**If last[char] == I then the current string is finished, let’s start new string and append size to result array.**

class Solution:

def partitionLabels(self, S):

if not S:

return []

last = {c: i for i,c in enumerate(S)}

j,start = 0,0

res = []

for i in range(len(S)):

j = max(j, last[S[i]])

if i == j:

res.append(i - start + 1)

start = i + 1

return res

Partition List

1. Use 2 lists one is for < x and another is >= x

class Solution:

def partition(self, head, x):

if not head:

return

min\_p = None

e1 = None

max\_p = None

e2 = None

cur = head

while cur:

if cur.val < x:

if not min\_p:

min\_p = cur

e1 = min\_p

cur = cur.next

e1.next = None

else:

e1.next = cur

e1 = e1.next

cur = cur.next

e1.next = None

else:

if not max\_p:

max\_p = cur

e2 = max\_p

cur = cur.next

e2.next = None

else:

e2.next = cur

e2 = e2.next

cur = cur.next

e2.next = None

if min\_p and max\_p:

e1.next = max\_p

return min\_p

if min\_p:

return min\_p

return max\_p

Car Pooling

1. Iterate with step 1 and and check which trips starts and which one finishes to add or subtract people.
2. Check if we don’t exceed capacity

class Solution:

def carPooling(self, trips, capacity):

if not trips or capacity == 0:

return False

start = {}

end = {}

max\_end = 0

for people,s,e in trips:

if s not in start:

start[s] = []

start[s].append(people)

if e not in end:

end[e] = []

end[e].append(people)

max\_end = max(e, max\_end)

occupied = 0

for i in range(max\_end + 1):

if i in end:

for num in end[i]:

occupied -= num

if i in start:

for num in start[i]:

occupied += num

if occupied > capacity:

return False

return True

**Squares of a Sorted Array**

1. **Find last negative element**
2. **Use merge to compare squares of negative or positive numbers**

**class Solution:**

**def sortedSquares(self, arr):**

**if not arr:**

**return []**

**j = 0**

**while j < len(arr) and arr[j] < 0:**

**j += 1**

**i = j - 1**

**res = []**

**while i >= 0 and j < len(arr):**

**a = arr[i] \*\* 2**

**b = arr[j] \*\* 2**

**if a < b:**

**res.append(a)**

**i -= 1**

**else:**

**res.append(b)**

**j += 1**

**while i >= 0:**

**res.append(arr[i] \*\* 2)**

**i -= 1**

**while j < len(arr):**

**res.append(arr[j] \*\* 2)**

**j += 1**

**return res**

**4Sum:**

**1.Use one by one 3Sum, 2Sum.**

2. 2Sum: sort an array, use 2 pointers and go from start and from tail until they meet.

**3.**

class Solution:

def twoSum(self, nums, target, start, end):

res = []

i = start

j = end

while i < j:

sum\_of\_two = nums[i] + nums[j]

if sum\_of\_two > target:

j -= 1

elif sum\_of\_two < target:

i += 1

else:

res.append([nums[i], nums[j]])

i += 1

j -= 1

while i < j and nums[i] == nums[i - 1]:

i += 1

while i < j and nums[j] == nums[j + 1]:

j -= 1

return res

def threeSum(self, nums, target, start, end):

res = []

for j in range(start, end - 1):

rest = target - nums[j]

candidates = self.twoSum(nums, rest, j + 1, len(nums) - 1)

if len(candidates) > 0:

# [[a,b],[c,d]]

# flatten it

for a,b in candidates:

res.append([nums[j], a, b])

return res

def hash(self, list\_of\_nums):

h = 11 \* list\_of\_nums[0]

for i in range(1, len(list\_of\_nums)):

h += list\_of\_nums[i]

return h

def has\_dup(self, nums, list\_of\_nums):

for num in nums:

if num == list\_of\_nums:

return True

return False

def fourSum(self, nums, target):

nums.sort()

res = {}

neg\_res = set()

for i in range(len(nums) - 3):

rest = target - nums[i]

if rest in neg\_res:

continue

candidates = self.threeSum(nums, rest, i + 1, len(nums) - 1)

if len(candidates) > 0:

#[[a,b,c],[e,f,g]]

for a,b,c in candidates:

hash = self.hash([a, b, c])

if hash in res:

if self.has\_dup(res[hash], [nums[i], a,b,c]):

continue

if hash not in res:

res[hash] = []

res[hash].append([nums[i], a,b,c])

else:

neg\_res.add(rest)

return list([item for k,items in res.items() for item in items])

Find All Duplicates in Array

1. Go though array and mark value as negative
2. If we meet negative value it means we find duplicate and we can add it to result array

class Solution:

def findDuplicates(self, nums):

if not nums:

return []

res = []

for z in nums:

i = abs(z) - 1

if nums[i] < 0:

res.append(i + 1)

else:

nums[i] = -nums[i]

return res

Max chunks to make sorted

1. Look for item in own position with and is greater then last maximum

class Solution:

def maxChunksToSorted(self, arr):

if not arr:

return 0

ans = 0

max\_val = 0

for i in range(len(arr)):

max\_val = max(max\_val, arr[i])

if max\_val == i:

ans += 1

return ans

Spiral Matrix II

1.Calculate min and max row and column.

class Solution:

def generateMatrix(self, n):

if n == 0:

return [[]]

if n == 1:

return [[n]]

new\_size = n \*\* 2

mat = [0] \* n

for i in range(len(mat)):

mat[i] = [0] \* n

num = 1

# to the left

r1 = 0

c1 = 0

while num <= new\_size:

r2 = r1 + n - 1

c2 = c1 + n - 1

if num == new\_size:

mat[r1][c1] = num

num += 1

break

#to the left

for j in range(c1, c2 + 1):

mat[r1][j] = num

num += 1

#to the bottom

for i in range(r1 + 1, r2 + 1):

mat[i][c2] = num

num += 1

#to the left

a = num + n - 2

for j in range(c1, c2):

mat[r2][j] = a

a -= 1

num = mat[r2][c1] + 1

#to the top

for i in range(r2 - 1, r1, -1):

mat[i][c1] = num

num += 1

r1 += 1

c1 += 1

n -= 2

return mat

**Construct Binary Tree from Preorder and Inorder Traversal**

**1.Use preorder as new root of building tree**

**2. Create 2 pointers there are start and end that represent an interval where is located current root from preorder.**

**3. On each recursive call find index in order for node from preorder and split the inorder into start:index for left subtree and index+1:end is for right subtree.**

class Solution:

def buildTree(self, preorder, inorder):

if not preorder or not inorder:

return None

in\_idx = {x: i for i, x in enumerate(inorder)}

def build(in\_idx, in\_s, in\_e):

nonlocal pre\_idx

if in\_s == in\_e:

return None

x = preorder[pre\_idx]

root = TreeNode(x)

index = in\_idx[x]

pre\_idx += 1

root.left = build(in\_idx, in\_s, index)

root.right = build(in\_idx, index + 1, in\_e)

return root

pre\_idx = 0

root = build(in\_idx, 0, len(inorder))

return root

Maximum Product Subarray

1. Allocate dp\_max and dp\_min arrays.
2. Set up dp\_max[0] = arr[0] and dp\_min[0] = arr[0]
3. Go through the rest items let’s calculate dp\_max[i] = max(dp\_mmdp\_max[i-1]\* arr[i],dp\_min[i]=dp\_min[i-1]\*arr[i] and arr[i] and dp\_min[i]=min(dp\_min[i-1]\*arr[i],dp\_max[i-1]\*arr[i] and arr[i])
4. max\_prod = max(max\_prod, dp\_max[i])

Maximum subarray

1.Go through array.Sum up items, if sum is less 0 then set up sum to 0

2. Memorize maximum contigues sum

class Solution:

def maxSubArray(self, nums):

if not nums:

return []

max\_so\_far = 0

max\_sum = float('-inf')

i = 0

while i < len(nums):

max\_so\_far += nums[i]

max\_sum = max(max\_sum, max\_so\_far)

if max\_so\_far < 0:

max\_so\_far = 0

i += 1

return max\_sum

House Rober

1.Allocate array by length + 1. We save loot for i-th home

2. loot[i] = max(loot[i-1], loot[i-2] \* money[i-1]) We select to rob home or don’t

3. Answer is at last item.

def rob(nums: Array[Int]): Int = {

if(nums.isEmpty) 0

else if(nums.length == 1) nums(0)

else if (nums.length == 2) nums.max

else {

val money = Array.ofDim[Int](nums.length + 1)

money(1) = nums(0)

for (i <- 2 to nums.length) {

val loot = money(i - 1) max (money(i - 2) + nums(i - 1))

money(i) = loot

}

money(nums.length)

}

}

**Number of Equivalent Domino Pairs**

1.Allocate new array where is each item is another array from 0 till 9.

2. Go through input array and update global count by value from arr[\_.\_1][\_.\_2] use \_.\_1 as key

3. Update second array by 1 to mark count of dominoes with (a,b)

4.

from collections import defaultdict

class Solution:

def numEquivDominoPairs(self, dominoes):

if not dominoes:

return 0

mem = defaultdict(lambda: [0] \* 10)

pairs = 0

for i in range(len(dominoes)):

c, d = dominoes[i]

arr = mem[c]

pairs += arr[d]

if c != d:

arr = mem[d]

pairs += arr[c]

mem[c][d] += 1

return pairs

**Maximum Product of Three Numbers**

1. **Sort array.**
2. **If first item is less 0 and last one is positive then to consider 2 variants: last \* item[0] \* item[1] or item[last] \* item[last-1] \* item[last-2]**
3. **In other case to return product of last three items.**

**Subarray Product Less Than K**

**1.consider array where each item is index of smallest number in the given array for the j: nums[i] \* nums[i+1] \*.. nums[j] < k**

**class Solution:**

**def numSubarrayProductLessThanK(self, nums, k):**

**if not nums or k <= 1:**

**return 0**

**prod = 1**

**left = 0**

**sub\_arr\_count = 0**

**for i in range(len(nums)):**

**prod \*= nums[i]**

**while prod >= k:**

**prod /= nums[left]**

**left += 1**

**sub\_arr\_count += i - left + 1**

**return sub\_arr\_count**

**Kill Process**

**1.Use BFS to process children of process.**

**2. Create map where the key is pid and value is children list.**

from collections import defaultdict

class Solution:

def killProcess(self, pid, ppid, kill):

if not pid or not ppid or not kill:

return

queue = [kill]

res = []

parent\_to\_child = defaultdict(lambda :[])

for i in range(len(ppid)):

parent\_to\_child[ppid[i]].append(pid[i])

while len(queue) > 0:

proc = queue.pop()

res.append(proc)

for child in parent\_to\_child[proc]:

queue.append(child)

return res

Smallest String Starting From Leaf

1. go from root to leaf by append node char to buffer. If node is a leaf let’s reverse the buffer and compare the result with earlier storing minimal lexographical string

O(N\*lg(N)) where N – node number and Log N is heigh of complete tree.

class Solution:  
 def \_\_init\_\_(self):  
 self.min\_str = None  
  
 def to\_char(self, code):  
 return chr(ord('a') + code)  
  
 def min\_string(self, s1, s2):  
 if not s1:  
 return s2  
 if not s2:  
 return s1  
 return min(s1, s2)  
  
 def smallestFromLeaf(self, root):  
 if not root:  
 return root  
 def traverse(node, str=''):  
 if node and not node.left and not node.right:  
 ch = self.to\_char(node.val)  
 s = ch + str[::-1]  
 self.min\_str = self.min\_string(s, self.min\_str)  
 return  
 if not node:  
 return  
 ch = self.to\_char(node.val)  
 traverse(node.left, str + ch)  
 traverse(node.right, str + ch)  
 self.min\_str = None  
 traverse(root)  
 return self.min\_str

Keys and rooms

1. Use DFS to visit all rooms if first room we visit is 0

2. If we visited this room we skip it.

3. allocate set to store all visited rooms.

class Solution:  
 def canVisitAllRooms(self, rooms):  
 if not rooms or len(rooms) == 0:  
 return False  
 def dfs(room, visited):  
 visited.add(room)  
 for key in rooms[room]:  
 if key in visited:  
 continue  
 dfs(key, visited)  
 visited = set()  
 dfs(0, visited)  
 return len(visited) == len(rooms)

Increasing Order Search Tree:

1. Use in-order to place nodes in such order and connect right subtree.

2. Left subtree is new root.

class Solution:  
 def increasingBST(self, root):  
 if not root:  
 return root  
  
 def rearrange(node):  
 if not node:  
 return None  
 if not node.left and not node.right:  
 return TreeNode(node.val)  
 left\_subtree = rearrange(node.left)  
 #left\_subtree is new root  
 #find the rightmost node  
 rightmost\_leave = left\_subtree  
 while rightmost\_leave and rightmost\_leave.right:  
 rightmost\_leave = rightmost\_leave.right  
 new\_node = TreeNode(node.val)  
 if rightmost\_leave:  
 rightmost\_leave.right = new\_node  
 new\_node.right = rearrange(node.right)  
 return left\_subtree if left\_subtree else new\_node  
  
 return rearrange(root)

Path In Zigzag Labelled Binary Tree

1. Traverse through the complete tree using for left subtree index is 2 \* index + 1 and for right subtree is 2 \* index + 2. Start traversing from label and go to the root. Let’s distinguish node label and node position: label is position + 1.

from math import log  
  
class Solution:  
 def pathInZigZagTree(self, label):  
 if label < 0:  
 return []  
 row = int(log(label, 2)) + 1  
 #start label  
 s = 2 \*\* (row - 1)  
 #end label  
 e = 2 \* s - 1  
 res = []  
 #find pos  
 pos = label - 1  
 #pos is label - 1  
 if row % 2 == 0:  
 off = e - label  
 pos = s - 1 + off  
 while pos >= 0:  
 label = pos + 1  
 if row % 2 == 0:  
 #numbers are from right to left  
 #fix label  
 off = label - s  
 label = e - off  
 if pos % 2 == 0:  
 pos -= 2  
 else:  
 pos -= 1  
 res.insert(0, label)  
 pos //= 2  
 row -= 1  
 s //= 2  
 e = s \* 2 - 1  
 return res

3 Sum Closest

**1. Sort number array**

**2. Take item and take subarray from next item to the end**

**3. Use two pointers technic to find diff = abs(target – sum of 3 items) then compare it to the previous diff**

class Solution:  
 def threeSumClosest(self, nums, target):  
 if not nums or len(nums) < 3:  
 return None  
 nums.sort()  
 closest = sum(nums[:3])  
 for i in range(len(nums) - 2):  
 s = i + 1  
 e = len(nums) - 1  
 while s < e:  
 candidate = nums[i] + nums[s] + nums[e]  
 if abs(target - candidate) < abs(target - closest):  
 closest = candidate  
  
 if candidate > target:  
 e -= 1  
 elif candidate < target:  
 s += 1  
 else:  
 return target  
 return closest

Can Make Palindrome from Substring

***1.Use array of all prefix frequencies; assume to use only 26 latin charecters.***

***2. R[i+1] – L[j[] is substring frequencies.***

***3. If half of odds frequencies is less or equal then the allowed changes let’s add true in result array in other case to add false.***

class Solution:  
 def canMakePaliQueries(self, s, queries):  
 if not s or not queries:  
 return []  
 dp = [[0] \* 26]  
 for i in range(1, len(s) + 1):  
 new\_dp = dp[i-1][:]  
 new\_dp[ord(s[i - 1]) - ord('a')] += 1  
 dp.append(new\_dp)  
 answer = []  
 for start,end,may\_change in queries:  
 R = dp[end + 1]  
 L = dp[start]  
 odds = sum((R[i] - L[i]) & 1 for i in range(26))  
 if odds // 2 <= may\_change:  
 answer.append(True)  
 else:  
 answer.append(False)  
 return answer

**Sort colors:**

1. Use 2 pointers: rightmost one is for 0, leftmost one is for 2

class Solution:  
 def sortColors(self, nums):  
 if not nums:  
 return nums  
 p0 = 0  
 p2 = len(nums) - 1  
 cur = 0  
 while cur <= p2:  
 if nums[cur] == 0:  
 nums[p0], nums[cur] = nums[cur], nums[p0]  
 cur += 1  
 p0 += 1  
 elif nums[cur] == 2:  
 nums[p2], nums[cur] = nums[cur], nums[p2]  
 p2 -= 1  
 else:  
 cur += 1

**Minimum Time Difference**

1. Convert string to hours extracting hours and minutes and convert them to hours.

I hours is less 12 then let’s add hours + 24 \* 60 because time may be time in the next day.

1. Sort new array and find difference between adjacent items.
2. class Solution:  
    def findMinDifference(self, timePoints):  
    if not timePoints:  
    return None  
    min\_diff = 24 \* 60 + 1  
    times = []  
    for timePoint in timePoints:  
    hours,mins = timePoint.split(':')  
    mins = int(mins)  
    hours = int(hours)  
    time = hours \* 60 + mins  
    times.append(time)  
    if hours < 12:  
    times.append(time + 24 \* 60)  
    times.sort()  
    for i in range(1, len(times)):  
    min\_diff = min(times[i] - times[i - 1], min\_diff)  
    return min\_diff

Search a 2D Matrix ||

1. Use approaches if target > item in matrix then go to the left, if target < item in matrix then go to the up

2. Start from bottom-left.

class Solution:  
 def searchMatrix(self, matrix, target):  
 if not matrix:  
 return False  
 col = 0  
 row = len(matrix) - 1  
 while row >=0 and col < len(matrix[0]):  
 if matrix[row][col] > target:  
 row -= 1  
 elif matrix[row][col] < target:  
 col += 1  
 else:  
 return True  
 return False

**Sort List**

1. Compare and merge sublists by length 1, 2, 4, .. log(list len)

class Solution:  
 def copy\_list(self, head):  
 p = head  
 new\_head = None  
 new\_p = None  
 while p:  
 if not new\_head:  
 new\_head = ListNode(p.val)  
 new\_p = new\_head  
 else:  
 new\_p.next = ListNode(p.val)  
 new\_p = new\_p.next  
 p = p.next  
 return new\_head  
  
 def go(self, head, step):  
 p1 = head  
 while p1 and step > 0:  
 p1 = p1.next  
 step -= 1  
 return p1  
  
 def merge\_sort(self, head, list\_len, size):  
 p0 = head  
 p1 = self.go(p0, size)  
 new\_head = None  
 pn = None  
 while p0 or p1:  
 if p0 and p1:  
 #merge both sublists in ascending order  
 sz0 = size  
 sz1 = size if list\_len >= 2 \* size else list\_len - size  
 while p0 and p1 and sz0 > 0 and sz1 > 0:  
 if p0.val > p1.val:  
 new\_node = ListNode(p1.val)  
 if not new\_head:  
 new\_head = new\_node  
 pn = new\_head  
 else:  
 pn.next = new\_node  
 pn = pn.next  
 sz1 -= 1  
 p1 = p1.next  
 else:  
 new\_node = ListNode(p0.val)  
 if not new\_head:  
 new\_head = new\_node  
 pn = new\_head  
 else:  
 pn.next = new\_node  
 pn = pn.next  
 sz0 -= 1  
 p0 = p0.next  
 if sz0 > 0:  
 while sz0 and p0:  
 pn.next = ListNode(p0.val)  
 pn = pn.next  
 p0 = p0.next  
 sz0 -= 1  
 if sz1 > 0:  
 while sz1 and p1:  
 pn.next = ListNode(p1.val)  
 pn = pn.next  
 p1 = p1.next  
 sz1 -= 1  
 elif p0 or p1:  
 if not new\_head:  
 new\_head = self.copy\_list(p0) if p0 else self.copy\_list(p1)  
 pn = new\_head  
 else:  
 pn.next = self.copy\_list(p0)  
 pn = pn.next  
 p0 = p1  
 p1 = self.go(p0, size)  
 return new\_head  
  
 def sortList(self, head):  
 if not head:  
 return  
 list\_len = 0  
 p0 = head  
 while p0:  
 list\_len += 1  
 p0 = p0.next  
 size = 1  
 while size <= list\_len:  
 head = self.merge\_sort(head, list\_len, size)  
 size \*= 2  
 return head

**Insert Sort List**

1. Use insertion sort to order single list items. Account for the if first item is greater then new inserted, store last minimum item to insert new one after.

class Solution:  
 def insertionSortList(self, head):  
 if not head:  
 return None  
 new\_head = None  
 p0 = head  
 while p0:  
 if not new\_head:  
 new\_head = ListNode(p0.val)  
 else:  
 p1 = new\_head  
 prev = None  
 while p1 and p1.val <= p0.val:  
 prev = p1  
 p1 = p1.next  
 if not prev:  
 t = new\_head  
 new\_head = ListNode(p0.val)  
 new\_head.next = t  
 else:  
 t = prev.next  
 prev.next = ListNode(p0.val)  
 prev.next.next = t  
 p0 = p0.next  
 return new\_head

**Longest string chain**

1.Consider each string is succ string without one character.

2. allocate 1 dimensional array length, sort words by length and length[word[i]] = max(length(word[i], word[j] + 1)) where j < i

class Solution:  
 def longestStrChain(self, words):  
 if not words:  
 return False  
  
 def is\_pred(pred, succ):  
 chars = [0] \* 26  
 for ch in words[pred]:  
 chars[ord(ch) - ord('a')] += 1  
 for ch in words[succ]:  
 pos = ord(ch) - ord('a')  
 chars[pos] -= 1  
 if chars[pos] < -1:  
 return False  
 zeros = 0  
 ones = 0  
 for a in chars:  
 if a == -1:  
 ones += 1  
 if a == 0:  
 zeros += 1  
 if ones > 1:  
 return False  
 return zeros == len(chars) - 1  
  
 #sort by length  
 words.sort(key=lambda s: len(s))  
 dp = [1] \* len(words)  
 max\_len = 1  
 for i in range(len(words) - 1):  
 for j in range(i + 1, len(words)):  
 if len(words[j]) > len(words[i]) + 1:  
 break  
 if len(words[i]) < len(words[j]) and is\_pred(i, j):  
 dp[j] = max(dp[j], dp[i] + 1)  
 max\_len = max(max\_len, dp[j])  
 return max\_len

**Triangle**

1. Go from bottom to up and allocate 1-dimensional array.

2. dp[i] = min(dp[i], dp[i+1]) + triangle[k][i]

3. In inner cycle we have the upper bound by triangle[k]

class Solution:  
 def minimumTotal(self, triangle):  
 if not triangle:  
 return 0  
 dp = [tri for tri in triangle[-1]]  
 for row in range(len(triangle) - 2, -1, -1):  
 for j in range(len(triangle[row])):  
 dp[j] = min(dp[j], dp[j + 1]) + triangle[row][j]  
 return dp[0]

**Ugly number**

1. Count number using binary search and the function F(A) = A/a + A/b + A/c – A/ac – A/ab – A/bc + A/abc. Where ab, ac, bc, abc is multiple of respective numbers and they are calculate with lcm = a\*b / gcd

2. Use the binary search to found out number of F(A) and compare it to the given n

class Solution:  
 def nthUglyNumber(self, n, a, b, c):  
 def gcd(a, b):  
 if b == 0:  
 return a  
 return gcd(b, a % b)  
  
 def lcm(a, b):  
 return a \* b // gcd(a, b)  
  
 ab, bc, ac = lcm(a, b),lcm(b, c), lcm(a, c)  
 abc = lcm(a, bc)  
 lo = 1  
 hi = 2 \* 10 \*\* 9  
 while lo < hi:  
 mid = lo + (hi - lo) // 2  
 cnt = mid // a + mid // b + mid // c - mid // ab - mid // ac - mid // bc + mid // abc  
 if cnt < n:  
 lo = mid + 1  
 else:  
 hi = mid  
 return lo

**Smallest string with swaps**

1. use union find to collect all indices in pair.

2. sort chars in one connected component(pairs)

from collections import defaultdict  
  
class Solution:  
 def smallestStringWithSwaps(self, s, pairs):  
 if not s:  
 return None  
 if len(pairs) == 0:  
 return s  
 def find(parent, i):  
 if parent[i] != i:  
 p = find(parent, parent[i])  
 parent[i] = p  
 return parent[i]  
  
 def union(parent, i, j):  
 p1 = find(parent, i)  
 p2 = find(parent, j)  
 if p1 != p2:  
 parent[p1] = p2  
  
 parent = [i for i in range(len(s))]  
 for i,j in pairs:  
 union(parent, i, j)  
  
 chars = defaultdict(list)  
 for i in range(len(s)):  
 chars[find(parent, i)].append(s[i])  
 for k in chars:  
 chars[k].sort()  
  
 res = []  
 for i in range(len(s)):  
 res.append(chars[find(parent, i)].pop(0))  
 return ''.join(res)

**Large Divisible Set**

1. Sort nums, allocate the array dp to store max length of subset from [X1,….Xi]

2. If X[i] % X[j] == 0 i > j in a subset it means X[i] is divided by other members of the current subset

class Solution:  
 def largestDivisibleSubset(self, nums):  
 if not nums:  
 return []  
 nums.sort()  
 dp = [1] \* len(nums)  
 for i in range(1, len(nums)):  
 for j in range(0, i):  
 if nums[i] % nums[j] == 0:  
 dp[i] = max(dp[i], dp[j] + 1)  
 #look up max by first item in tuple  
 max\_len,max\_index = max([(v, i) for i,v in enumerate(dp)])  
 cur\_tail,cur\_size = nums[max\_index],max\_len  
 res = []  
 for i in range(max\_index, -1, -1):  
 if dp[i] == cur\_size and cur\_tail % nums[i] == 0:  
 res.insert(0, nums[i])  
 cur\_size -= 1  
 cur\_tail = nums[i]  
 return res

**Sum target**

1. Use recursion to find if we can reach the given S.

2. Memoized the solution for position and sum and return it.

class Solution:

def findTargetSumWays(self, nums, S):

if not nums:

return 0

dp = [0] \* len(nums)

for i in range(len(dp)):

dp[i] = [0] \* 2001

dp[0][nums[0] + 1000] = 1

dp[0][-nums[0] + 1000] += 1

for i in range(1, len(nums)):

for s in range(-1000, 1001):

if dp[i - 1][1000 + s] > 0:

dp[i][s + nums[i] + 1000] += dp[i-1][1000 + s]

dp[i][s - nums[i] + 1000] += dp[i-1][1000 + s]

return dp[len(nums) - 1][S + 1000] if S <= 1000 else 0

Time complexity(L \*n() where L is range of sum

Memory is O(n)

Longest Substring with At Most K Distinct Characters

1.Use 2 pointers to go through chars in a string and move the left pointer.

from collections import defaultdict  
  
class Solution:  
 def lengthOfLongestSubstringKDistinct(self, s, k):  
 if not s or k == 0:  
 return 0  
 chars = defaultdict(lambda: 0)  
 i = 0  
 j = 0  
 max\_len = 0  
 while i <= j:  
 if j < len(s):  
 chars[s[j]] += 1  
 j += 1  
 while len(chars.keys()) > k and i < len(s) and i <= j:  
 chars[s[i]] -= 1  
 if chars[s[i]] == 0:  
 del chars[s[i]]  
 i += 1  
 max\_len = max(max\_len, j - i)  
 else:  
 break  
 return max\_len

Longest Arithmetic Subsequence of Given Difference

1.Use DP but use number as dp index and for each number to check if number – difference exists in dp and it it’s so let’s dp[arr[i]] = dp[arr[i] – difference] + 1

2. if arr[i] may be less 0 then add prefix as 10000

class Solution:  
 def longestSubsequence(self, arr, difference):  
 if not arr:  
 return 0  
 dp = [-1] \* (3 \* (10 \*\* 4) + 1)  
 max\_len = 1  
 dp[10000 + arr[0]] = 1  
 for i in range(1, len(arr)):  
 a = arr[i]  
 if dp[10000 + a - difference] > - 1:  
 dp[10000 + a] = dp[10000 + a - difference] + 1  
 else:  
 dp[10000 + a] = 1  
 max\_len = max(max\_len, dp[10000 + a])  
 return max\_len

 Play with Chips

1. count odd chips and compare it to even one. If number of odds are greater then find first odd position and calculate cost others. In other case it uses even position.

class Solution:  
 def calc\_cost(self, chips, pos):  
 cost = 0  
 for chip in chips:  
 cost += abs(pos - chip) % 2  
 return cost  
  
 def minCostToMoveChips(self, chips):  
 if not chips:  
 return 0  
 odds = sum(map(lambda x: 1 if x % 2 == 1 else 0, chips))  
 evens = len(chips) - odds  
 for pos,\_ in enumerate(chips):  
 if odds > evens:  
 if pos % 2 == 1:  
 return self.calc\_cost(chips, pos)  
 else:  
 if pos % 2 == 0:  
 return self.calc\_cost(chips, pos)  
 return self.calc\_cost(chips, chips[0])

Count Vowels Permutation

1.Consider some string with length j: we create table where dp rows are ending chars and columns are length from 0 to n. So dp[i][j] contains number of strings ending by char i and having length j. From char i we can move to specific chars so we know possible strings for strings[k][j+1] but where k is char which we can add to j. So dp[k][j+1] is increased by dp[i][j] because dp[i][j] is number strings ending by i.

class Solution:  
 def countVowelPermutation(self, n):  
 if n == 0:  
 return 0  
 chars = ['a', 'e', 'i', 'o', 'u']  
 moves = {0: [1], 1: [0, 2], 3: [2, 4], 4: [0], 2: [1, 0, 3, 4]}  
 dp = [0] \* len(chars)  
 for i in range(len(dp)):  
 dp[i] = [0] \* (n + 1)  
  
 for i in range(len(dp)):  
 dp[i][1] = 1  
 for j in range(2, len(dp[0])):  
 for i in range(len(chars)):  
 to\_chars = moves[i]  
 for char in to\_chars:  
 dp[char][j] += dp[i][j-1]  
 count = 0  
 base = 10 \*\* 9 + 7  
 for j in range(len(dp)):  
 count += (dp[j][n] % base)  
 return count % base

Verifying an Alien Dictionary

1.Map char to integer to define new char ordering

2. Convert word to array number respecting to order.  
3. Compare all words

class Solution:

def isAlienSorted(self, words, order):

if not words or not order:

return False

map = {char: i for i,char in enumerate(order)}

new\_words = []

for word in words:

new\_word = [map[ch] for ch in word]

new\_words.append(new\_word)

for i in range(1, len(new\_words)):

if new\_words[i-1] > new\_words[i]:

return False

return True

 Binary Tree Right Side View

1.Use stack to store node with depth where first node is root with 0.

2. Pop up node and if it’s not None add it to map where depth is key and node value is value if this depth is not in map.

class Solution:  
 def rightSideView(self, root):  
 if not root:  
 return []  
 max\_depth = 0  
 rightmost\_node = {}  
 stack = [(root, 0)]  
 while stack:  
 node,depth = stack.pop()  
  
 if node is not None:  
 max\_depth = max(max\_depth, depth)  
 if depth not in rightmost\_node:  
 rightmost\_node[depth] = node.val  
 stack.append((node.left, depth + 1))  
 stack.append((node.right, depth + 1))  
 return [rightmost\_node[depth] for depth in range(max\_depth + 1)]

Permutation in String

1.Use sliding window: add new chars to frequency map and remove leftmost one.

2 .if 2 maps are the same then it returns true

class Solution:  
 def rightSideView(self, root):  
 if not root:  
 return []  
 max\_depth = 0  
 rightmost\_node = {}  
 stack = [(root, 0)]  
 while stack:  
 node,depth = stack.pop()  
  
 if node is not None:  
 max\_depth = max(max\_depth, depth)  
 if depth not in rightmost\_node:  
 rightmost\_node[depth] = node.val  
 stack.append((node.left, depth + 1))  
 stack.append((node.right, depth + 1))  
 return [rightmost\_node[depth] for depth in range(max\_depth + 1)]

**Capacity to ship packages within D days**

1. Use binary search from 0 to max sum of weights.

2. go through all weights to check if it’s enough capacity conveyer belt.

class Solution:  
 def enough\_belt\_capacity(self, weights, max\_w, d):  
 days = 0  
 capacity = max\_w  
 for weight in weights:  
 if weight > max\_w:  
 return False  
 if weight > capacity:  
 #move to the next day to carry the left packages  
 days += 1  
 capacity = max\_w - weight  
 else:  
 capacity -= weight  
 days += 1  
 return days <= d  
  
 def shipWithinDays(self, weights, D):  
 if not weights or D == 0:  
 return 0  
 lo = 0  
 hi = 50000 \* 500  
 while lo < hi:  
 max\_w = lo + (hi - lo) // 2  
 if self.enough\_belt\_capacity(weights, max\_w, D):  
 hi = max\_w  
 else:  
 lo = max\_w + 1  
 return lo

**Split Array Largest Sum**

1. Use binary search to look up the min max sum of a subarray.

2. Count subarray number that are less the given sum

3. If that numbers is less the then hi = mid – 1and in other case it has to be lo = mid + 1

class Solution:  
 def splitArray(self, nums, m):  
 if not nums or m == 0:  
 return 0  
 lo = 0  
 hi = 0  
 for i in range(len(nums)):  
 hi += nums[i]  
 if nums[i] > lo:  
 lo = nums[i]  
 res = hi  
 while lo <= hi:  
 f = lo + (hi - lo) // 2  
 sum\_nums = 0  
 cnt = 1  
 for i in range(len(nums)):  
 if sum\_nums + nums[i] > f:  
 cnt += 1  
 sum\_nums = nums[i]  
 else:  
 sum\_nums += nums[i]  
 if cnt <= m:  
 res = min(res, f)  
 hi = f - 1  
 else:  
 lo = f + 1  
 return res

Koko Eating Bananas

class Solution:  
 def minEatingSpeed(self, piles, H):  
 if not piles or H == 0:  
 return 0  
 k = max(piles)  
 lo = 1  
 hi = sum(piles)  
 while lo <= hi:  
 speed = lo + (hi - lo) // 2  
 hours = 0  
 for i in range(len(piles)):  
 if piles[i] <= speed:  
 hours += 1  
 else:  
 #eating piles  
 hours += (piles[i] // speed)  
 if piles[i] % speed != 0:  
 hours += 1  
 if hours <= H:  
 k = min(speed, k)  
 hi = speed - 1  
 else:  
 lo = speed + 1  
 return k

Subsets II

1.Sort input array

2. Use combination recursive function

3. Sort result array by length and append items to new array while appending new item the last one is not the same.

class Solution:  
 def subsetsWithDup(self, nums):  
 if not nums:  
 return []  
 nums.sort()  
 res = []  
 def subsets(i, cur):  
 if i == len(nums):  
 res.append(cur)  
 return  
 subsets(i + 1, cur)  
 new\_cur = cur[:]  
 new\_cur.append(nums[i])  
 subsets(i + 1, new\_cur)  
 subsets(0, [])  
 res.sort(key=lambda x:[x, len(x)])  
 new\_res = []  
 for a in res:  
 if len(new\_res) == 0 or new\_res[-1] != a:  
 new\_res.append(a)  
 return new\_res

#### [Sort Items by Groups Respecting Dependencies](https://leetcode.com/problems/sort-items-by-groups-respecting-dependencies/)

1. Build graph for items and groups using beforeItems

2. Order group graph with topological sort

3. for every group in group graph sort group’s items with topological sort

4. Check if all items in result follow the beforeItems.

from collections import defaultdict  
  
class Solution:  
 def sortItems(self, n, m, group, beforeItems):  
 if not group or not beforeItems:  
 return []  
 node\_graph = defaultdict(list)  
 group\_graph = defaultdict(list)  
 group\_to\_node = defaultdict(list)  
  
 for node in range(n):  
 group\_to\_node[group[node]].append(node)  
  
 for node,before\_items in enumerate(beforeItems):  
 for before\_item in before\_items:  
 node\_graph[before\_item].append(node)  
 if group[before\_item] != group[node]:  
 group\_graph[group[before\_item]].append(group[node])  
  
 visited\_group = [False] \* (m + 1)  
  
 def group\_top\_sort(g, visited, adj\_list, res):  
 visited[g + 1] = True  
 for neighbour in adj\_list[g]:  
 if not visited[neighbour + 1]:  
 group\_top\_sort(neighbour, visited, adj\_list, res)  
 res.insert(0, g)  
  
 def item\_top\_sort(node, g, visited, adj\_list, res):  
 visited[node] = True  
 if group[node] == g:  
 for neighbour in adj\_list[node]:  
 if group[neighbour] != g:  
 continue  
 if not visited[neighbour]:  
 item\_top\_sort(neighbour, g, visited, adj\_list, res)  
 res.insert(0, node)  
  
 res\_group = []  
 for g in range(-1, m):  
 if not visited\_group[g + 1]:  
 group\_top\_sort(g, visited\_group, group\_graph, res\_group)  
 res = []  
 visited\_item = [False] \* n  
 for i in range(len(res\_group) - 1, -1, -1):  
 g = res\_group[i]  
 for node in group\_to\_node[g]:  
 if not visited\_item[node]:  
 item\_top\_sort(node, g, visited\_item, node\_graph, res)  
 pos\_to\_node = {node:i for i,node in enumerate(res)}  
 #check if all nodes follow to restrictions  
 for i,node in enumerate(res):  
 for before\_item in beforeItems[node]:  
 if pos\_to\_node[before\_item] > i:  
 return []  
 return res

 Smallest Subtree with all the Deepest Nodes

1. Calculate max depth and then if node has such depth let consider such cases:

- left subtree and right subtree return nodes then return parent of them

- return non empty subtree.

class Solution:  
 def subtreeWithAllDeepest(self, root):  
 if not root:  
 return []  
  
 def depth(node):  
 if not node:  
 return 0  
 left\_depth = 1 + depth(node.left)  
 right\_depth = 1 + depth(node.right)  
 return max(left\_depth, right\_depth)  
  
 def is\_leaf(node):  
 return node and node.right is None and node.left is None  
  
 def sub\_tree(node, d):  
 if d == 0:  
 return node  
 if not node:  
 return None  
 left\_tree = sub\_tree(node.left, d - 1)  
 right\_tree = sub\_tree(node.right, d - 1)  
 if left\_tree and right\_tree:  
 return node  
 if left\_tree:  
 return left\_tree  
 return right\_tree  
  
 if is\_leaf(root):  
 return root  
  
 d = depth(root) - 1 # number of nodes  
 return sub\_tree(root, d)

 Minimum Domino Rotations For Equal Row

1. At least if one A[i] or B[i] is absent in A or B it returns -1

class Solution(object):  
 def minDominoRotations(self, A, B):  
 if not A or not B:  
 return -1  
  
 def check(x):  
 flip\_a = 0  
 flip\_b = 0  
 for i in range(len(A)):  
 if A[i] != x and B[i] != x:  
 return -1  
 if A[i] == x and B[i] != x:  
 flip\_b += 1  
 elif B[i] == x and A[i] != x:  
 flip\_a += 1  
 return min(flip\_a, flip\_b)  
 flip = check(A[0])  
 if flip == -1:  
 return check(B[0])  
 return flip

**Asteroid collision**

1. Use stack to put there asteroid

class Solution(object):  
 def asteroidCollision(self, asteroids):  
 if not asteroids:  
 return []  
 res = [asteroids[0]]  
 i = 1  
 while i < len(asteroids):  
 while res and asteroids[i] < 0 < res[-1]:  
 if res[-1] < -asteroids[i]:#  
 res.pop()  
 continue  
 elif res[-1] == -asteroids[i]: # =  
 res.pop()  
 break  
 else:  
 res.append(asteroids[i])  
 i += 1  
 return res

Longest Arithmetic Sequence

1. Use longest arithemetic sequence array only 2 dimensional with row as difference A[i] – A[j]

i > j

class Solution(object):

def longestArithSeqLength(self, A):

if not A:

return 0

las = [1] \* 20001

for i in range(len(las)):

las[i] = [1] \* len(A)

max\_len = 0

for i in range(len(A)):

for j in range(i):

d = A[i] - A[j]

las[2001 + d][i] = max(las[2001 + d][i], las[2001 + d][j] + 1)

max\_len = max(max\_len, las[2001 + d][i])

return max\_len

Check Completeness of a Binary Tree

1. Use BFS until a node is None.

2. Go to the right by evicting the None

3. If nodes is empty a tree is complete.

class Solution(object):  
 def isCompleteTree(self, root):  
 if not root:  
 return False  
 nodes = [root]  
 while nodes[0] is not None:  
 node = nodes.pop(0)  
 nodes.append(node.left)  
 nodes.append(node.right)  
 while len(nodes) > 0 and nodes[0] is None:  
 nodes.pop(0)  
 return len(nodes) == 0

**Split a String in Balanced Strings**

1. Use counter for L and for R.
2. When both of them are equal let’s increment split counter.

class Solution(object):

def balancedStringSplit(self, s):

if not s:

return 0

l = 0

r = 0

splits = 0

for i in range(len(s)):

if s[i] == 'L':

l += 1

elif s[i] == 'R':

r += 1

if l == r:

l = 0

r = 0

splits += 1

return splits

**Dice Roll Simulation**

1.Use DP to store combination number at i-th dice roll if last number was j. dp[i][j] = dp[i-1][1…6]. If number j has restriction we add todp[i][j] the combinations with dp[i - x][1,,6(no j)] i - x >= 0

class Solution(object):

def dieSimulator(self, n, rollMax):

if n == 0 or not rollMax:

return 0

dp = [[0 for \_ in range(6)] for \_ in range(n)]

for i in range(n):

for j in range(6):

for x in range(1, rollMax[j] + 1):

if i - x >= 0:

for v in range(6):

if v != j:

dp[i][j] = dp[i][j] + dp[i - x][v]

else:

dp[i][j] += 1

break

return sum(dp[n-1])

**Fraction to Recurring Decimal**

1. Use long division to get result.
2. Divide numerator by denumerator to get integer part.
3. Divide numerator % denumerator to get remainder and if it’s equal zero to return result without fractional part.
4. Until remainder is zero let’s add digits to fractional part: multiply remainder by 10 and

save remainder to map to check out if it’s met earlier. If so then we have recuring result and embrace it in curly braces.

class Solution(object):

def fractionToDecimal(self, numerator, denominator):

if denominator == 0:

return Exception('denominator is 0')

if numerator == 0:

return '0'

res = []

if (denominator < 0) ^ (numerator < 0):

res.append('-')

a = abs(numerator)

b = abs(denominator)

res.append(str(a // b))

a = a % b

if a == 0:

return ''.join(res)

res.append('.')

remainders = {}

while a != 0:

if a in remainders:

p = remainders[a]

res.insert(p, '(')

res.append(')')

break

remainders[a] = len(res)

a \*= 10

res.append(str(a // b))

a = a % b

return ‘'.join(res)

**Expression Add Operators**

1. Use recursion to solve it.
2. Create recursive function (pos, prev\_operand, cur\_operand, value, buf)
3. If we calculate multiplication we will subtract value - prev\_operand + prev\_operand \* cur\_operand

class Solution(object):

def addOperators(self, num, target):

if not num:

return []

res = []

def calc\_exp(pos, prev\_operand, cur\_operand, value, buf):

if pos == len(num):

if value == target and cur\_operand == 0:

res.append(''.join(buf[1:]))

return

cur\_operand = cur\_operand \* 10 + int(num[pos])

if cur\_operand > 0:

calc\_exp(pos + 1, prev\_operand, cur\_operand, value, buf)

str\_oper = str(cur\_operand)

# +

buf.append('+')

buf.append(str\_oper)

calc\_exp(pos + 1, cur\_operand, 0, value + cur\_operand, buf)

buf.pop()

buf.pop()

if buf:

# -

buf.append('-')

buf.append(str\_oper)

calc\_exp(pos + 1, -cur\_operand, 0, value - cur\_operand, buf)

buf.pop()

buf.pop()

# \*

buf.append('\*')

buf.append(str\_oper)

calc\_exp(pos + 1, prev\_operand \* cur\_operand, 0, value - prev\_operand + prev\_operand \* cur\_operand, buf)

buf.pop()

buf.pop()

calc\_exp(0, 0, 0, 0, [])

return res

**Before and After Puzzle**

1. Use map to store last word and phrase indices which contain the given word

from collections import defaultdict

class Solution(object):

def beforeAndAfterPuzzles(self, phrases):

if not phrases:

return []

res = set()

start\_word = defaultdict(list)

for i,phrase in enumerate(phrases):

words = phrase.split()

start\_word[words[0]].append(i)

for i,phrase in enumerate(phrases):

words = phrase.split()

last\_word = words[-1]

# act as before phrase

if last\_word in start\_word:

for j in start\_word[last\_word]:

if i == j:

continue

new\_phrase = words + phrases[j].split()[1:]

string = ' '.join(new\_phrase)

res.add(string)

new\_list = list(res)

new\_list.sort()

return new\_list

**Minimum Cost Tree From Leaf Values**

1. Use DP to solve it. Outer loop for length and inner loop is from I to i+lenth and third loop I <= k < j to

dp[i][j] = min(dp[i][j], max(arr[i:k + 1]) \* max(arr[k + 1: j + 1]) + dp[i][k] + dp[k+1][j])

class Solution(object):

def mctFromLeafValues(self, arr):

if not arr:

return float('inf')

dp = [[float('inf')] \* len(arr) for \_ in range(len(arr))]

for i in range(len(arr)):

dp[i][i] = 0

for l in range(1, len(arr)):

for i in range(len(arr)):

j = i + l

if j >= len(arr):

break

for k in range(i, j):

dp[i][j] = min(dp[i][j], max(arr[i:k + 1]) \* max(arr[k + 1: j + 1]) + dp[i][k] + dp[k+1][j])

return dp[0][-1]

**3Sum**

1. Use 2 pointers technic. If a+b == target - we just change 2 pointers.

2. For triplet uniqueness we store only tuple2.

class Solution(object):

def threeSum(self, nums):

if len(nums) < 3:

return []

res = []

nums.sort()

seen = set()

for i in range(len(nums) - 2):

target = -nums[i]

j = i + 1

k = len(nums) - 1

while j < k:

sum\_of\_two = nums[j] + nums[k]

if sum\_of\_two == target:

if (nums[i], nums[j]) not in seen:

res.append([nums[i], nums[j], nums[k]])

seen.add((nums[i], nums[j]))

j += 1

k -= 1

elif sum\_of\_two > target:

k -= 1

else:

j += 1

return res

**Flatten a Multilevel Doubly Linked List**

1. Use recursion to traverse a list
2. If we meet a node with child let call the node.child recursively and then call the function with node.next

class Solution(object):

def flatten(self, head):

if not head:

return None

first = None

last = None

def add(node, last):

if last:

last.next = Node(node.val, last, None, None)

return last.next

return Node(node.val, None, None, None)

def make\_flat(node):

nonlocal last, first

if not node:

return

last = add(node, last)

if not first:

first = last

if node.child:

make\_flat(node.child)

make\_flat(node.next)

make\_flat(head)

return first

**Number of Longest Increasing Subsequence**

1. Allocate new array count where we store count of Lis ending at position I
2. If j < I and dp[i] <= dp[j](length is less) it means we have the same number LIS at I as at j so count[i] = count[j]
3. If dp[i] == dp[j] + 1 let’s increase count[i] by count[j]

class Solution(object):

def findNumberOfLIS(self, nums):

if not nums:

return 0

dp = [0] \* len(nums)

count = [1] \* len(nums)

for i in range(len(nums)):

for j in range(0, i):

if nums[i] > nums[j]:

if dp[j] >= dp[i]:

dp[i] = dp[j] + 1

count[i] = count[j]

elif dp[j] + 1 == dp[i]:

count[i] += count[j]

max\_len = max(dp)

return sum([c for i,c in enumerate(count) if dp[i] == max\_len])

**Next Permutation**

1. Find the a[i] < a[i+1] in decreasing subsequence
2. Find the least item that is bigger then a[i], swap them
3. Reverse all a[j] from I + 1

class Solution(object):

def nextPermutation(self, nums):

if not nums:

return

i = len(nums) - 2

# find first decreasing item

while i >= 0 and nums[i + 1] <= nums[i]:

i -= 1

if i >= 0:

# find the least bigger item

j = len(nums) - 1

while j >= 0 and nums[j] <= nums[i]:

j -= 1

nums[i],nums[j] = nums[j], nums[i]

# reverse

i += 1

j = len(nums) - 1

while i < j:

nums[i],nums[j] = nums[j],nums[i]

i += 1

j -= 1

**Letter Combinations of a Phone Number**

1. Use recursive solution where is digit on which we are working on.
2. Go through letter of digits:

2.1 If res is empty add new list with one char

2.2 If last res list has length less than I + 1 we copy by taking [0:i ] and add new char

If I == len(digits) then we return result

‘

class Solution(object):

def letterCombinations(self, digits):

if not digits:

return []

num\_to\_chars = {"0": " ", "1": "", "2": "abc", "3": "def", "4": "ghi", "5": "jkl", "6": "mno", "7": "pqrs", "8": "tuv", "9": "wxyz"}

def comb(i, res):

if i == len(digits):

return

for ch in num\_to\_chars[digits[i]]:

if not res:

res.append([ch])

else:

if len(res[-1]) < (i + 1):

res[-1].append(ch)

else:

seq = res[-1][:i]

seq.append(ch)

res.append(seq)

comb(i + 1, res)

res = []

comb(0, res)

return [''.join(arr) for arr in res]

**Palindromic Substrings**

1. We have 2\*n - 1 positions for centre of polindrom
2. We calculate left and right and changes it to get number of palindromic substrings

class Solution(object):

def countSubstrings(self, s):

if not s:

return 0

res = 0

for center in range(2\*len(s) - 1):

left = center // 2

right = left + center % 2

while left >= 0 and right < len(s) and s[left] == s[right]:

res += 1

left -= 1

right += 1

return res

**Bulb Switcher**

1. Count factors for every bulb. If I bulb has odd number of factors it will be on in other case it will be off. Numbers that have squares have odd numbers of factors.

If number is not -square it has even number of factors:

If num % a == 0:

Factors +=1

If num % a != a:

Factors += 1

2. So we can just found all squared number till N.

3. Of just calculate sqrt of N.

from math import sqrt

class Solution(object):

def bulbSwitch(self, n):

if n <= 0:

return 0

bulb\_on = int(sqrt(n))

return bulb\_on

**Range Sum Query 2D - Immutable**

1.Use additional matrix to store sum of [0,0]-[i,j]

2. To store sum to additional matrix it needs to set up 1 line as original matrix and sum[i][j] = matrix[i][j] + sum[i-1][j], then sum[i][j] += sum[i][j-1]

class NumMatrix(object):

def \_\_init\_\_(self, matrix):

self.aux = [0] \* len(matrix)

for i in range(len(matrix)):

self.aux[i] = [0] \* len(matrix[0])

for j in range(len(matrix[0])):

self.aux[0][j] = matrix[0][j]

for i in range(1, len(self.aux)):

for j in range(len(self.aux[i])):

self.aux[i][j] = matrix[i][j] + self.aux[i-1][j]

for i in range(len(self.aux)):

for j in range(1, len(self.aux[i])):

self.aux[i][j] += self.aux[i][j - 1]

def sumRegion(self, row1, col1, row2, col2):

if 0 == len(self.aux) or 0 == len(self.aux[0]):

return 0

sum\_reg = self.aux[row2][col2]

if row1 - 1 >= 0:

sum\_reg -= self.aux[row1-1][col2]

if col1 - 1 >= 0:

sum\_reg -= self.aux[row2][col1 - 1]

if row1 - 1 >= 0 and col1 - 1 >= 0:

sum\_reg += self.aux[row1-1][col1-1]

return sum\_reg

**Contiguous Array**

1. Use counter that decreased if we meet 0 and increase if we meet 1.
2. Use map to store count-> index. If we meet such count in map we get the equal number of zeros and ones.

class Solution(object):

def findMaxLength(self, nums):

if not nums:

return 0

m = {0: -1}

count = 0

max\_len = 0

for i in range(len(nums)):

if nums[i] == 0:

count += -1

else:

count += 1

if count in m:

max\_len = max(max\_len, i - m[count])

else:

m[count] = i

return max\_len

**Campus Bikes**

1. **Use bucket sort where bucket is distance between worker and bike.**
2. **Appending new item to buckets is order in which it needs to get result for workers and bikes.**

class Solution(object):

def assignBikes(self, workers, bikes):

if not workers or not bikes:

return []

buckets = [[] for \_ in range(2001)]

for i in range(len(workers)):

for j in range(len(bikes)):

d = abs(workers[i][0] - bikes[j][0]) + abs(workers[i][1] - bikes[j][1])

buckets[d].append((i,j))

res = [-1 for i in range(len(workers))]

used\_bikes = set()

for bucket in buckets:

for worker,bike in bucket:

if res[worker] == -1 and bike not in used\_bikes:

res[worker] = bike

used\_bikes.add(bike)

return res

**Diagonal Traverse**

1. Declare 2 steps to change row and column
2. If row + d1 < 0 or row+d1 > len(matrix) then change col += 1 if col+1 < len(matrix[0]) or row += 1
3. If col + d2 < 0 or col + d2 > len(matrix[0]) then change row += 1 if row+1 < len(matrix) or col += 1

4. In other case we change row += d1 and col += d2

class Solution(object):

def findDiagonalOrder(self, matrix):

if not matrix or not matrix[0]:

return []

res = []

r = 0

c = 0

d1 = -1

d2 = 1

count = len(matrix) \* len(matrix[0])

while count > 0:

res.append(matrix[r][c])

if r + d1 < 0 or r + d1 == len(matrix):

if c + 1 == len(matrix[0]):

r += 1

else:

c += 1

d1 = -d1

d2 = -d2

elif c + d2 < 0 or c + d2 == len(matrix[0]):

if r + 1 == len(matrix):

c += 1

else:

r += 1

d1 = -d1

d2 = -d2

else:

r += d1

c += d2

count -= 1

return res

**Summary Range.**

1. Use 2 pointers technic

class Solution(object):

def summaryRanges(self, nums):

if not nums:

return []

left = 0

right = 0

res = []

for i in range(1, len(nums)):

if nums[right] + 1 == nums[i]:

right += 1

else:

if left < right:

res.append('{}->{}'.format(nums[left], nums[right]))

else:

res.append(nums[left])

left = i

right = i

if left < right:

res.append('{}->{}'.format(nums[left], nums[right]))

else:

res.append(nums[left])

return res

**Boundary of Binary Tree**

1. Use distinct functions for traverse left, right, leaves boundaries.
2. If we traverse left and right boundaries we start from root.left and root.right
3. Add root as first element in result list
4. Use set to put node in to set
5. Reverse right list

6. In left boundaries to go to the left subtree and go the right one if node doesn’t have the left one. The same works for the right boundaries only with left subtree.

class Solution(object):

def boundaryOfBinaryTree(self, root):

if not root:

return []

def left\_boundary(node, seen, res):

if not node:

return

if node not in seen:

res.append(node.val)

seen.add(node)

left\_boundary(node.left, seen, res)

if not node.left:

left\_boundary(node.right, seen, res)

def right\_boundary(node, seen, res):

if not node:

return

if node not in seen:

res.append(node.val)

seen.add(node)

right\_boundary(node.right, seen, res)

if not node.right:

#probably it will be included to leaf

right\_boundary(node.left, seen, res)

def leaves(node, seen, res):

if not node:

return

leaves(node.left, seen, res)

if not node.left and not node.right and node not in seen:

res.append(node.val)

seen.add(node)

leaves(node.right, seen, res)

seen = set()

lefts = []

leave\_list = []

rights = []

left\_boundary(root.left, seen, lefts)

leaves(root, seen, leave\_list)

right\_boundary(root.right, seen, rights)

rights.reverse()

return [root.val] + lefts + leave\_list + rights

**Daily Temperatures**

1. Use stack to store index of T[i]. If we add new value and we pop up all T[i] <= value and update resulting array.

class Solution(object):

def dailyTemperatures(self, t):

if not t:

return []

res = [0] \* len(t)

wait = []

for i in range(len(t)):

while wait and t[wait[-1]] < t[i]:

res[wait[-1]] = i - wait[-1]

wait.pop()

wait.append(i)

return res

**All Nodes Distance K in Binary Tree**

1. Add the field par to every node with dis
2. Use BFS starting from target and add left,right and parent to queue thus go to bottom and top from target at the same time. Add node we’ve seen to set.
3. If we meet the node with the given distance in queue it means that all nodes in queue are our result.

class Solution(object):

def distanceK(self, root, target, K):

if not root or not target:

return None

def dfs(node, par = None):

if node:

node.par = par

dfs(node.left, node)

dfs(node.right, node)

dfs(root)

q = [(target, 0)]

seen = {target.val}

while q:

if q[0][1] == K:

return [n.val for n,d in q]

node,dist = q.pop(0)

for n in (node.left, node.right, node.par):

if n and n.val not in seen:

q.append((n, dist + 1))

return []

**Course Schedule II**

1. Use topological sort and use additional rec\_stack to recognise a cycle in a graph.

class Solution(object):

def findOrder(self, numCourses, prerequisites):

if numCourses == 0:

return []

adj\_list = [[] for i in range(numCourses)]

# [a,b] b goes first, a -> b

for a,b in prerequisites:

adj\_list[b].append(a)

def top\_sort\_if\_no\_cycle(node, rec\_stack, seen, res):

seen.add(node)

rec\_stack[node] = True

for child in adj\_list[node]:

if child not in seen:

if top\_sort\_if\_no\_cycle(child, rec\_stack, seen, res):

return True

elif rec\_stack[child]:

return True

res.append(node)

rec\_stack[node] = False

return False

rec\_stack = [False] \* numCourses

res = []

seen = set()

for n in range(numCourses):

if n not in seen:

if top\_sort\_if\_no\_cycle(n, rec\_stack, seen, res):

return []

return res[::-1]

**Populating Next Right Pointers in Each Node**

1. Use 2 connections: node.left.next = node.right
2. Node.right.next = node.next.left where node is on level upper

class Solution(object):

def connect(self, root):

if not root:

return root

leftmost = root

while leftmost.left:

head = leftmost

while head:

head.left.next = head.right

if head.next:

head.right.next = head.next.left

head = head.next

leftmost = leftmost.left

return root

Rectangle Overlap

1. Consider cases when 2 rectangle can’t overlap: if one is above other or one is left then other

class Solution:

def isRectangleOverlap(self, rec1, rec2):

if not rec1 or not rec2:

return False

x1,y1,x2,y2 = rec1

x3,y3,x4,y4 = rec2

# one rect is upper then other

if y2 <= y3 or y4 <= y1:

return False

if x2 <= x3 or x4 <= x1:

return False

return True

**Partition Labels**

1. Save to map rightmost position of every char in input string
2. Declare max\_i to save the current rightmost position
3. In the loop: calculate max\_i = max(max\_i, m[char])
4. If current position equals to the rightmost position then append new part length
5. In other case continue

class Solution:

def partitionLabels(self, s):

if not s:

return []

m = {c:i for i,c in enumerate(s)}

parts = []

j = 0

max\_i = 0

last = 0

while j < len(s):

max\_i = max(max\_i, m[s[j]])

if j == max\_i:

parts.append(j - last + 1)

last = j + 1

j += 1

return parts

**Minimum Size Subarray Sum**

1. Use 2 pointers: add right number and while sum >= s let’s calculate min\_len = min(min\_len, j - I + 1)

class Solution:

def minSubArrayLen(self, s, nums):

if not s or not nums:

return 0

cur\_sum = 0

min\_len = float('inf')

j = 0

for i in range(len(nums)):

cur\_sum += nums[i]

while cur\_sum >= s:

min\_len = min(min\_len, i - j + 1)

cur\_sum -= nums[j]

j += 1

return min\_len

**Convert Sorted List to Binary Search Tree**

1. Copy increasing list to array
2. Use recursion to build tree to find middle of the range[l,r]
3. Continue this for [l,mid-1] and [mid+1, r]

class Solution:

def sortedListToBST(self, head):

if not head:

return None

arr = []

node = head

while node:

arr.append(node.val)

node = node.next

def build\_tree(arr, s, e):

if s > e:

return None

# find middle

m = (s + e) // 2

new\_node = TreeNode(arr[m])

if s == e:

return new\_node

new\_node.left = build\_tree(arr,s, m - 1)

new\_node.right = build\_tree(arr,m + 1, e)

return new\_node

return build\_tree(arr, 0, len(arr) - 1)

**132 Pattern**

1.Find all min element for the position I

2. Declare stack where we store third items.

3. Go from the end: if a item is in the stack is greater then min[i] - we will pop up the stack until it’s less min[i], then if min[i] < item[i] < stack[-1] returns true in other case push item[i] to the stack.

class Solution:

def find132pattern(self, nums):

if not nums:

return False

min\_arr = [0] \* len(nums)

min\_arr[0] = nums[0]

for i in range(1, len(min\_arr)):

min\_arr[i] = min(min\_arr[i - 1], nums[i])#decreasing order

stack = [] # for third element

for i in range(len(nums) - 1,-1, -1):

if nums[i] > min\_arr[i]:

while stack and stack[-1] <= min\_arr[i]:

stack.pop()

if stack and stack[-1] < nums[i]:

return True

stack.append(nums[i])# nums[i] <= stack[-1]

return False

Cheapest flights within stops K

1. Use Dejkstra algorithm searching minimal node with min weight.
2. Use BFS adding the (weigh, step, node) to min binary heap and pull min item.
3. If node== dest return weight, in other case if we have step > 0 add new adjacent nodes of this node.

from collections import defaultdict

from heapq import heappop,heappush

class Solution:

def findCheapestPrice(self, n, flights, src, dst, K):

if n == 0 or not flights:

return -1

adj\_list = defaultdict(list)

for s,e,w in flights:

adj\_list[s].append((w, e))

q = [(0,K+1,src)]

#seen = set()

while q:

w,step,u = heappop(q)

if u == dst:

return w

if step > 0:

for w1,v in adj\_list[u]:

heappush(q, (w + w1, step - 1, v))

return -1

**Greatest Common Divisor of Strings**

1. Use smaller string as first gcd and further to try smallest substring.

class Solution:

def has\_gcd(self, str1, gcd):

if len(str1) < len(gcd) or len(str1) % len(gcd) != 0:

return False

i = 0

while i < len(str1):

for j in range(len(gcd)):

if str1[i + j] != gcd[j]:

return False

i += len(gcd)

return True

def gcdOfStrings(self, str1, str2):

if not str1 or not str2:

return ""

if len(str1) < len(str2):

t = str1

str1 = str2

str2 = t

for i in range((len(str1) // 2 + 1), 0, -1):

gcd = str1[0: i]

if self.has\_gcd(str1, gcd) and self.has\_gcd(str2, gcd):

return gcd

return “”

**Buddy Strings**

1. If A == B we search 2 equal chars to swap them.
2. If A != B we just find all different chars and if its count != 2 it returns False

class Solution:

def buddyStrings(self, A, B):

if not A and not B:

return False

if not A or not B:

return False

A = list(A)

B = list(B)

if A == B:

chars = set()

for char in A:

if char in chars:

return True

chars.add(char)

return False

else:

res = []

for i in range(len(A)):

if A[i] != B[i]:

res.append(i)

if len(res) != 2:

return False

i,j = res[0],res[1]

return A[i] == B[j] and A[j] == B[i]

Check valid string

1. Store index of ( and \* to 2 array.
2. If it meets ) it checks if either left parenthesis or star array is not empty and pop it, otherwise it returns False

class Solution:

def checkValidString(self, s):

if not s:

return True

left = []

star = []

for i in range(len(s)):

if s[i] == '(':

left.append(i)

elif s[i] == ')':

if left:

left.pop()

elif star:

star.pop()

else:

return False

else:

star.append(i)

while len(left) > 0 and len(star) > 0:

if star[-1] > left[-1]:

left.pop()

star.pop()

else:

break

return len(left) == 0

**Remove Outermost Parentheses**

1. Use counter if encounter ( let increment ,otherwise decrement it.
2. If counter == 0 to memorise first outer parenthesise, if counter == 1 and letter is ) then save new pair of indices of outer ()

class Solution:

def removeOuterParentheses(self, S):

if not S:

return S

parn = 0

outers = []

left = None

for i in range(len(S)):

if S[i] == ')':

if parn == 1:

outers.append([left, i])

parn -= 1

else:

if parn == 0:

left = i

parn += 1

res = []

for a,b in outers:

res.append(S[a:b+1])

return ‘'.join(res)

**Remove Duplicate Letters**

1. Use stack to store them leftmost char. If current char > top of stack we try to pop up it.

from collections import Counter

class Solution:

def removeDuplicateLetters(self, s):

if not s:

return s

cnt = Counter(s)

used = set()

stack = []

for i in range(len(s)):

cnt[s[i]] -= 1

if s[i] in used:

continue

while stack and s[i] < stack[-1] and cnt[stack[-1]] > 0:

used.discard(stack[-1]) # to allow to add at second time

stack.pop()

stack.append(s[i])

used.add(s[i])

return ‘'.join(stack)

**Rotting Oranges**

1.Use BFS by append rotting fruits

class Solution:

def orangesRotting(self, grid):

if not grid:

return -1

rotten = []

cnt = 0

for i in range(len(grid)):

for j in range(len(grid[0])):

if grid[i][j] == 2:

rotten.append((i, j, 0))

elif grid[i][j] == 1:

cnt += 1

elapsed = 0

while rotten:

r, c, minutes = rotten.pop(0)

elapsed = minutes

if r + 1 < len(grid) and grid[r + 1][c] == 1:

grid[r + 1][c] = 2

cnt -= 1

rotten.append((r + 1, c, minutes + 1))

if r - 1 >= 0 and grid[r - 1][c] == 1:

grid[r - 1][c] = 2

cnt -= 1

rotten.append((r - 1, c, minutes + 1))

if c + 1 < len(grid[0]) and grid[r][c + 1] == 1:

grid[r][c + 1] = 2

cnt -= 1

rotten.append((r, c + 1, minutes + 1))

if c - 1 >= 0 and grid[r][c - 1] == 1:

grid[r][c - 1] = 2

cnt -= 1

rotten.append((r, c - 1, minutes + 1))

if cnt == 0:

return elapsed

return -1

**Accounts Merge**

1. Use DSU to map email to parent by encoding every email as integer
2. Use first email as to connect it with the rest email in account
3. Use map to collect result: key is parent and value is email list.

from collections import defaultdict

class Dsu:

def \_\_init\_\_(self, n):

self.parent = [i for i in range(n)]

def find(self, x):

if self.parent[x] != x:

p = self.find(self.parent[x])

self.parent[x] = p

return p

return self.parent[x]

def union(self, x1, x2):

p1 = self.find(x1)

p2 = self.find(x2)

self.parent[p1] = p2

class Solution:

def accountsMerge(self, accounts):

if not accounts:

return []

dsu = Dsu(1001)

em\_to\_id = {}

em\_to\_name = {}

i = 0

for acc in accounts:

name = acc[0]

for email in acc[1:]:

em\_to\_name[email] = name

if email not in em\_to\_id:

em\_to\_id[email] = i

i += 1

dsu.union(em\_to\_id[acc[1]], em\_to\_id[email])

res = defaultdict(list)

for email in em\_to\_name:

res[dsu.find(em\_to\_id[email])].append(email)

return [[em\_to\_name[v[0]]] + sorted(v) for v in res.values()]

**Redundant Connection**

1. Use disjoint union set storing the edges,
2. If add new edges to DSU and this edges have the same parent this edge is redundant.

class DSU:

def \_\_init\_\_(self, n):

self.parent = [i for i in range(n)]

def find(self, x):

if self.parent[x] != x:

p = self.find(self.parent[x])

self.parent[x] = p

return p

return self.parent[x]

def union(self, x1, x2):

p1 = self.find(x1)

p2 = self.find(x2)

self.parent[p1] = p2

class Solution:

def findRedundantConnection(self, edges):

if not edges:

return []

dsu = DSU(len(edges) \* 2 + 1)

for x,y in edges:

if dsu.find(x) == dsu.find(y):

return [x,y]

dsu.union(x, y)

return []

Two elements in array without duplicates

1. Xor all numbers from array
2. If xor contain 1 there is 2 numbers have differ so find rightmost 1 and sort number by this criteria.  
   class Solution:

def array\_two\_elements(self, arr):

if not arr:

return []

xor = 0

for a in arr:

xor = xor ^ a

# get rigtmost bit

xor = xor & -xor

rets = [0, 0]

for a in arr:

# look up the num with this bit is 1

if a & xor:

rets[0] = rets[0] ^ a

else:

rets[1] = rets[1] ^ a

return rets

**Cut Off Trees for Golf Event**

1. **Use BFS to resolve this problem.**
2. **When add new row,colum it’s important to add it to seen so avoid to consider it further.**

from collections import deque

class Solution(object):

def dist(self, forest, sr, sc, dr, dc):

R, C = len(forest), len(forest[0])

q = deque([(sr, sc, 0)])

seen = {(sr, sc)}

while q:

r, c, d = q.popleft()

if r == dr and c == dc:

return d

seen.add((r, c))

for r1, c1 in [(r + 1, c), (r - 1, c), (r, c + 1), (r, c - 1)]:

if 0 <= r1 < R and 0 <= c1 < C and (r1, c1) not in seen and forest[r1][c1]:

q.append((r1, c1, d + 1))

#seen.add((r1, c1))

return -1

def cutOffTree(self, forest):

if not forest:

return []

heights = sorted((v, r, c) for r, row in enumerate(forest) for c, v in enumerate(row) if v > 1)

sr, sc = 0, 0

total = 0

for v, r, c in heights:

d = self.dist(forest, sr, sc, r, c)

if d < 0:

return -1

total += d

sr = r

sc = c

return total

**Number of Dice Rolls With Target Sum**

1. Use DP to solve it. Allocate 2-dimensinal table dp[d][target]
2. For I = 1 dp[0][j] = 1, if j>f dp[0][j]=0
3. dp[i][j] = sum of dp[i-1][j-1] + d[i-1][j-2’… + dp[i-1][j-1] if j < f otherwise j = f

class Solution:

def numRollsToTarget(self, d, f, target):

if not d or not target or not f:

return 0

dp = [[0] \* (target + 1) for i in range(d)]

for j in range(target+1):

for i in range(d):

if i == 0:

if 0 < j <= f:

dp[i][j] = 1

else:

ways = 0

for k in range(1, min(j, f + 1)):

ways += dp[i-1][j-k]

dp[i][j] = ways

return dp[d-1][target] % (10 \*\* 9 + 7)

**Sentence Similarity**

1. Add all pair in set
2. Check if word1 == word2 or (word1,word2) or (word2.word1) contained in set

**Sentence Similarity II**

1. Use DSU to union pairs and encode word as integer

2. Go through words in words1 and word2: if they are equal or they have the same parent it returns true otherwise it returns false

**Bus Routes**

1. We use BSF to create bus graph.
2. A bus may connect another bus with bus stop.

from collections import defaultdict

class Solution:

def numBusesToDestination(self, routes, S, T):

if not routes:

return -1

stop\_to\_bus = defaultdict(set)

g = defaultdict(list)

for bus,route in enumerate(routes):

for bs in route:

if bus not in stop\_to\_bus[bs]:

for n in list(stop\_to\_bus[bs]):

g[n].append(bus)

g[bus].append(n)

stop\_to\_bus[bs].add(bus)

# set up 1 if we go by this bus but if S==T we don't ride by bus.

q = [[b,1] for b in list(stop\_to\_bus[S])]

seen = set(stop\_to\_bus[S])

dest = stop\_to\_bus[T]

while q:

bus,count = q.pop(0)

if bus in dest:

if S == T:

return 0

return count

for nei in g[bus]:

if nei not in seen:

seen.add(nei)

q.append([nei, count + 1])

return -1

**Longest Common Subsequence**

1. Use dynamic programming: dp[i][j] = max(dp[i-1][j],dp[i][j-1],1+dp[i-1][j-1])

class Solution:

def longestCommonSubsequence(self, text1, text2):

if not text1 or not text2:

return 0

dp = [[0] \* len(text1) for \_ in range(len(text2))]

for i in range(len(text2)):

for j in range(len(text1)):

if text1[j] == text2[i]:

dp[i][j] = 1

if i > 0 and j > 0:

dp[i][j] += dp[i-1][j - 1]

if i > 0:

dp[i][j] = max(dp[i-1][j], dp[i][j])

if j > 0:

dp[i][j] = max(dp[i][j-1], dp[i][j])

return dp[len(text2) - 1][len(text1) - 1]

**Number of Digit One**

1. Use such algorithm:

class Solution:

def countDigitOne(self, n):

if n <= 0:

return 0

i = 1

count = 0

while i <= n:

divider = i \* 10

count += (n // divider) \* i

count += min(max(n % divider - i + 1, 0), i)

i \*= 10

return count

**Digit Count in Range**

1. Use f(hi) - f(low-1)

class Solution:

def count\_digit(self, d, n):

if n < 0 and n < d:

return 0

count = 0

i = 1

while i <= n:

divider = i \* 10

count += (n // divider) \* i

if d > 0:

count += min(max(n % divider - d \* i + 1, 0), i)

else:

if n // divider > 0:

if i > 1:

count -= i

count += min(n % divider + 1, i)

i \*= 10

return count

**Minimum Time to Build Blocks**

1. Use Huffman code algorithm to build tree but with cost for every split: more frequently using code have shorter encoding in Huffman tree and here most weight block tase shorter path.
2. Join tow nodes with function = split + max(blocks[i-1],blocks[i])

from heapq import heappop,heapify,heappush

class Solution:

def minBuildTime(self, blocks, split):

heapify(blocks)

while len(blocks) > 1:

bl1 = heappop(blocks)

bl2 = heappop(blocks)

new\_bl = split + bl2

heappush(blocks, new\_bl)

return blocks[0]

**Maximum Sum of Two Non-Overlapping Subarrays**

1.Consider 2 options: sum L+M or M+L

2. Save Lmax before I and Mmax before, res is sum[L+M]

3. User prefix sum

4. For every I calculate max(Lmax+ sum[i-M,i], Mmax+sum[i-L,i],max\_sum)

class Solution:

def maxSumTwoNoOverlap2(self, A, L, M):

prefix = [0] \* len(A)

prefix[0] = A[0]

for i in range(1, len(A)):

prefix[i] = prefix[i - 1] + A[i]

l\_max = prefix[L - 1]

m\_max = prefix[M - 1]

max\_sum = prefix[L + M - 1]

for i in range(L + M, len(A)):

# L,M

l\_max = max(l\_max, prefix[i - M] - prefix[i - L - M])

# M,L

m\_max = max(m\_max, prefix[i - L] - prefix[i - L - M])

max\_sum = max(max\_sum, l\_max + prefix[i] - prefix[i - M], m\_max + prefix[i] - prefix[i - L])

return max\_sum

**Set Matrix Zeroes**

1. Use m+n memory to mark row or col should be zeros.

class Solution:

def setZeroes(self, matrix):

if not matrix:

return

rows = [0] \* len(matrix)

cols = [0] \* len(matrix[0])

for i in range(len(matrix)):

for j in range(len(matrix[0])):

if matrix[i][j] == 0:

rows[i] = 1

cols[j] = 1

for i in range(len(matrix)):

for j in range(len(matrix[0])):

if rows[i] == 1 or cols[j] == 1:

matrix[i][j] = 0

**Corporate Flight Bookings**

1. Allocate array in length n. Add w for bookings[i][0] and subtract w for bookings[i][1].

2. Through result array and sum up all number from previous item: item[i] -= item[i-1]

class Solution:

def corpFlightBookings(self, bookings, n):

res = [0] \* n

for a, b, w in bookings:

res[a - 1] += w

if b < n:

res[b] -= w

for i in range(1, n):

res[i] += res[i-1]

return res

Single number II

1. Use 2 numbers:
2. Seen1 = ~seen2 & (seen1 ^num)
3. Seen2 = ~ seen1 & (seen2 ^ num)
4. In seen1 will be result

class Solution:

def singleNumber(self, nums):

if not nums:

return None

seen1 = 0

seen2 = 0

for num in nums:

seen1 = ~seen2 & (seen1 ^ num)

seen2 = ~seen1 & (seen2 ^ num)

return seen1

Single Number III

1. Use bit mask to store diff of 2 number don’t have duplicate.
2. Find rightmost bit 1 diff = bitmask & (-bitmask)
3. Go through array and xor number with 1 in righmost position
4. Result is xor result and result xor bitmask

class Solution:

def singleNumber(self, nums):

if not nums:

return []

bitmask = 0

for num in nums:

bitmask ^= num

#get rightmost bit

bitmask = bitmask & (-bitmask)

x = 0

for num in nums:

if x & bitmask:

x = x ^ num

return [x, x ^ bitmask]

**Grumpy Bookstore Owner**

1. Use window size by X to count unsatisfied members.
2. Count all satisfied members and keep up max of unsatisfied in window

3. Deduct from window the left part grumpy[i-X] \* customer[i-X]

class Solution:

def maxSatisfied(self, customers, grumpy, X):

win = 0

max\_satisfied = 0

satisfied = 0

for i,x in enumerate(customers):

if grumpy[i] == 0:

satisfied += customers[i]

else:

win += customers[i]

if i >= X:

win -= grumpy[i - X] \* customers[i - X]

max\_satisfied = max(max\_satisfied, win)

return satisfied + max\_satisfied

**Candy Crush**

1. Check 3 cols to right and 3 rows to the bottom if they are the same and if so it sets up their values to negative ones.
2. Go by column and starts from bottom row:

Having 2 pointers to crush equal candles.

class Solution:

def candyCrush(self, board):

if not board:

return board

rows = len(board)

cols = len(board[0])

changes = 0

for j in range(cols):

for i in range(rows):

a = abs(board[i][j])

if a == 0:

continue

if j + 2 < cols:

if abs(board[i][j + 1]) == a and abs(board[i][j + 2]) == a:

board[i][j + 1] = -abs(board[i][j + 1])

board[i][j + 2] = -abs(board[i][j + 2])

changes += 1

if board[i][j] > 0:

board[i][j] = -board[i][j]

if i + 2 < rows:

if abs(board[i + 1][j]) == a and abs(board[i + 2][j]) == a:

changes += 1

board[i + 1][j] = -abs(board[i + 1][j])

board[i + 2][j] = -abs(board[i + 2][j])

if board[i][j] > 0:

board[i][j] = -board[i][j]

if changes == 0:

return board

for j in range(cols):

l = rows - 1

i = rows - 1

while i >= 0:

while l >= 0 and board[l][j] < 0:

l -= 1

if l >= 0:

board[i][j] = board[l][j]

l -= 1

else:

board[i][j] = 0

i -= 1

return self.candyCrush(board)

**Minimum Increment to Make Array Unique**

1. Sort array
2. Store max\_so\_far value, if we met equal values we would increase max\_so\_far and increase move by different max\_so\_far+1 - value

class Solution:

def minIncrementForUnique(self, A):

if not A or len(set(A)) == len(A):

return 0

A = sorted(A)

moves = 0

max\_so\_far = A[0]

for i in range(1, len(A)):

if A[i] <= max\_so\_far:

moves += (max\_so\_far + 1 - A[i])

A[i] = max\_so\_far + 1

max\_so\_far = max(max\_so\_far, A[i])

return moves

**Valid Triangle Number**

1. This can be solver as 3Dsum, but we use binary search by looking up the righmost K

a[i] + a[j] > a[K]

import bisect

class Solution:

def triangleNumber(self, nums):

if not nums:

return 0

nums.sort()

res = 0

for i in range(len(nums) - 2):

k = i + 2

for j in range(i+1, len(nums) - 1):

p = bisect.bisect\_left(nums, nums[i] + nums[j], k, len(nums))

if p > j:

res += p - j - 1

k = p

return res

**Critical Connections in a Network**

1.Use Tarjan SCC to find bridge in graph: low[v] > disc[u], where u is parent of v

from collections import defaultdict

class Solution:

def criticalConnections(self, n: int, connections):

low = [-1] \* n

disc = [-1] \* n

time = 1

adj\_list = defaultdict(list)

for s,e in connections:

adj\_list[s].append(e)

adj\_list[e].append(s)

def dfs(u, low, disc, pre, res):

nonlocal time

low[u] = time

disc[u] = time

time += 1

for v in adj\_list[u]:

if pre == v:

continue

if disc[v] == -1:

dfs(v,low, disc, u, res)

low[u] = min(low[u], low[v])

if low[v] > disc[u]:

res.append([u,v])

else:

low[u] = min(low[u], disc[v])

res = []

for i in range(n):

if disc[i] == -1:

dfs(i,low, disc,i,res)

return res

**Unique Paths**

1. Use dynamic programming: dp[i][j] = dp[i-1][j] + dp[i][j-1]

class Solution:

def uniquePaths(self, m: int, n: int):

#m is column number

dp = [[0] \* m for \_ in range(n)]

for i in range(m):

dp[0][i] = 1

for i in range(n):

dp[i][0] = 1

for i in range(1, n):

for j in range(1, m):

dp[i][j] = dp[i][j-1] + dp[i-1][j]

return dp[-1][-1]

**Find Peak Element**

**1 Use unequality nums[i] > nums[i+1] then nums[i] is peak**

**2. Can use binary seach**

class Solution:

def findPeakElement(self, nums):

if not nums:

return None

for i in range(len(nums)):

if nums[i] > nums[i+1]:

return i

return len(nums) - 1

**Minimum Remove to Make Valid Parentheses**

1. First remove ), let go from the left to right by increment balance if it meets ( and otherwise it decrements balance. If balance is 0 continue flow
2. Second remove (.

class Solution:

def minRemoveToMakeValid(self, s):

if not s:

return []

open\_seen = 0

balance = 0

chars = []

# remove ')'

for c in s:

if c == '(':

open\_seen += 1

balance += 1

elif c == ')':

if balance == 0:

continue

balance -= 1

chars.append(c)

keep\_open = open\_seen - balance

res = []

# remove '('

for c in chars:

if c == '(':

if keep\_open == 0:

continue

keep\_open -= 1

res.append(c)

return ‘'.join(res)

**Delete Operation for Two Strings**

1. Use DP to calculate min delete operations: dp[i][j] = dp[i-1][j-1] if s1[i-1]==s2[j-1]
2. Otherwise dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1])

class Solution:

def minDistance(self, word1, word2):

if not word1 and not word2 or word1 == word2:

return 0

if not word1:

return len(word2)

if not word2:

return len(word1)

dp = [[0] \* (len(word1) + 1) for \_ in range(len(word2) + 1)]

for i in range(len(word2) + 1):

for j in range(len(word1) + 1):

if i == 0 or j == 0:

dp[i][j] = i + j

elif word2[i - 1] == word1[j - 1]:

dp[i][j] = dp[i - 1][j - 1]

else:

dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1])

return dp[-1][-1]

**Smallest Subsequence of Distinct Characters**

**Use Time window!**

1. Use stack to push there characters.
2. If current char is greater then last one is in stack and char in stack has rightmost position is greater then current let’s pop it and push the current character.

class Solution:

def smallestSubsequence(self, text):

chars = [0] \* 26

for i,c in enumerate(text):

chars[ord(c) - ord('a')] = i

res = []

seen = set()

for i,c in enumerate(text):

if c in seen:

continue

while len(res) and res[-1] > c and chars[ord(res[-1]) - ord('a')] > i:

res.pop()

seen.add(c)

res.append(c)

return ‘'.join(res)

1.Next Greater Element III

1. Find the rightmost arr[i-1] < arr[i] ,then find rightmost arr[j] > arr[i-1] and swap them
2. Reverse substring from i

class Solution:

def to\_int(self, arr):

k = 1

n = 0

for i in range(len(arr) - 1, -1, -1):

n += arr[i] \* k

k \*= 10

return n

def nextGreaterElement(self, n):

if not n:

return -1

max\_n = 2 \*\* 31 - 1

digits = []

while n > 0:

d = n % 10

digits.insert(0, d)

n = n // 10

p = -1

i = len(digits) - 1

while i > 0 and p == -1:

if digits[i] > digits[i-1]:

p = i - 1

break

i -= 1

if p == -1:

return -1

r = -1

i = len(digits) - 1

while i > p and r == -1:

if digits[i] > digits[p]:

r = i

break

i -= 1

digits[p],digits[r] = digits[r],digits[p]

tail = digits[p+1:]

tail.reverse()

res = self.to\_int(digits[0:p+1] + tail)

if res > max\_n:

return -1

return res

**Expressive Words**

1. Encode S as array [char,count of consecutive char in S]
2. For every word in words:

- encode word

- if length of encoded word differs from length encoded S it continues

- otherwise for even chars from words should execute: group of S chars is greater than word group and lesser than 3 or word group is greater than S group

class Solution:

def enocode(self, s):

res = []

last = s[0]

count = 1

for c in s[1:]:

if last != c:

res.append([last, count])

count = 1

last = c

else:

count += 1

res.append([c, count])

return res

def expressiveWords(self, S, words):

chars1 = self.enocode(S)

count = 0

for word in words:

chars2 = self.enocode(word)

if len(chars1) != len(chars2):

continue

next\_word = False

i = 0

j = 0

while i < len(word) and j < len(chars1):

if chars1[j][0] != chars2[i][0] or chars2[i][1] < chars1[j][1] < 3 or chars1[j][1] < chars2[i][1]:

next\_word = True

break

i += 1

j += 1

if next\_word:

continue

count += 1

return count

Minimum ASC delete sum for 2 strings:

1. Use dp to solve it: dp[i][j] = dp[i+1][j+1] if s1[i] == s2[j]
2. Otherwise dp[i][j] = min(dp[i+1][j]+s2[i], dp[i][j+1] + s1[j])

class Solution:

def minimumDeleteSum(self, s1, s2):

dp = [[0] \* (len(s1) + 1) for \_ in range(len(s2) + 1)]

for i in range(len(s2) - 1, - 1, -1):

dp[i][len(s1)] = ord(s2[i]) + dp[i + 1][len(s1)]

for i in range(len(s1) - 1, -1, -1):

dp[len(s2)][i] = ord(s1[i]) + dp[len(s2)][i + 1]

for i in range(len(s2) - 1, -1, -1):

for j in range(len(s1) - 1, -1, -1):

if s2[i] == s1[j]:

dp[i][j] = dp[i + 1][j + 1]

else:

dp[i][j] = min(dp[i+1][j] + ord(s2[i]), dp[i][j+1] + ord(s1[j]))

return dp[0][0]

**MazeII**

1.Use BFS, but it had better use Dejkstra with PriorityQueue

2. Declare distance where we store distance from (i,j) and it works as indicator if we need to append (i,j) to queue.

3. In inner cycle before it starts increment r1,c1 by dr,dc but increment count inside cycle

class Solution:

def shortestDistance(self, maze, start, destination):

INF = 10001

q = []

q.append([start[0], start[1]])

dist = [[INF] \* len(maze[0]) for \_ in range(len(maze))]

dist[start[0]][start[1]] = 0

while q:

r, c = q.pop(0)

for dr, dc in [[-1, 0], [1, 0], [0, 1], [0, -1]]:

count = 0

r1, c1 = r + dr, c + dc

while 0 <= r1 < len(maze) and 0 <= c1 < len(maze[0]) and maze[r1][c1] == 0:

r1 += dr

c1 += dc

count += 1

r1 -= dr

c1 -= dc

if dist[r][c] + count < dist[r1][c1]:

dist[r1][c1] = dist[r][c] + count

q.append([r1, c1])

return -1 if dist[destination[0]][destination[1]] == INF else dist[destination[0]][destination[1]]

**Find the Closest Palindrome**

1. If a is 23456 then 235xx 236xx could be polindromes. If number is less after increasing or decreasing a then it could be 99….99 or 100…001.

class Solution:

def nearestPalindromic(self, S):

k = len(S)

candidates = [str(10 \*\* l + d) for l in (k - 1, k) for d in (-1, 1)]

p = int(S[:(k+1) // 2])

for prefix in map(str, [p - 1, p, p + 1]):

candidates.append(prefix + (prefix[:-1] if k % 2 != 0 else prefix)[::-1])

def diff(x):

return abs(int(x) - int(S))

ans = None

for cand in candidates:

if cand != S and not cand.startswith('0'):

if ans is None or diff(cand) < diff(ans) or diff(cand) == diff(ans) and cand == ans:

ans = cand

return ans

MinimumCostToMergeStones

import collections

class Solution(object):

def mergeStones(self, stones, K):

n = len(stones)

if (n - 1) % (K - 1) != 0:

return -1

cached = collections.defaultdict(dict)

for length in range(1, n + 1):

for i in range(n - length + 1):

j = i + length - 1

if length < K:

cached[i][j] = (sum(stones[i:j + 1]), 0, length)

elif length == K:

sums = sum(stones[i:j + 1])

cached[i][j] = (sums, sums, 1)

else:

min\_cost = float("Inf")

min\_cost\_len, min\_cost\_sum = -1, 0

for k in range(i, j):

a, b = cached[i][k], cached[k + 1][j]

if a[2] + b[2] < K:

cost = a[1] + b[1]

if cost < min\_cost:

min\_cost = cost

min\_cost\_sum = a[0] + b[0]

min\_cost\_len = a[2] + b[2]

elif a[2] + b[2] == K:

cost = a[1] + b[1] + a[0] + b[0]

if cost < min\_cost:

min\_cost = cost

min\_cost\_sum = a[0] + b[0]

min\_cost\_len = 1

cached[i][j] = (min\_cost\_sum, min\_cost, min\_cost\_len)

return cached[0][n - 1][1]

**Insufficient Nodes in Root to Leaf Paths**

1. Use DFS but distinguish the case when node doesn’t have the child

class TreeNode:

def \_\_init\_\_(self, x):

self.val = x

self.left = None

self.right = None

class Solution:

def sufficientSubset(self, root, limit):

if not root:

return root

def dfs(node, sum\_so\_far):

if not node:

return sum\_so\_far >= limit

new\_sum = node.val + sum\_so\_far

left\_subtree = dfs(node.left, new\_sum)

right\_subtree = dfs(node.right, new\_sum)

if not left\_subtree:

node.left = None

if not right\_subtree:

node.right = None

if not left\_subtree and not right\_subtree and node.left and node.right or not left\_subtree and not node.right or not right\_subtree and not node.left:

return False

return True

if not dfs(root, 0):

root = None

return root

MazeI

1. Use dis and visited to track cells we visited

class Solution:

def hasPath(self, maze, start, dest):

if not maze or not start or not dest:

return False

visited = [[False] \* len(maze[0]) for \_ in range(len(maze))]

def dfs(r,c):

visited[r][c] = True

if [r,c] == dest:

return True

for dr,dc in [[1,0],[-1,0],[0,-1], [0,1]]:

r1 = r + dr

c1 = c + dc

while 0<= r1 < len(maze) and 0 <= c1 < len(maze[0]) and maze[r1][c1] == 0:

r1 = r1 + dr

c1 = c1 + dc

r1 -= dr

c1 -= dc

if not visited[r1][c1]:

if dfs(r1,c1):

return True

return False

dfs(start[0], start[1])

return visited[dest[0]][dest[1]]

**Boundary of Binary Tree**

1. Use separate to find left\_boundary(node is at the left side not leaves), leaves, right boundary are nodes at the right side not leaves

class TreeNode:

def \_\_init\_\_(self, x):

self.val = x

self.left = None

self.right = None

class Solution:

def boundaryOfBinaryTree(self, root):

if not root:

return []

def is\_leaf(node):

return not node.right and not node.left

def leaves(node, nodes):

if not node:

return

if is\_leaf(node):

nodes.append(node.val)

leaves(node.left, nodes)

leaves(node.right, nodes)

res = [root.val]

t = root.left

while t:

if not is\_leaf(t):

res.append(t.val)

if t.left:

t = t.left

else:

t = t.right

leaves(root, res)

st = []

t = root.right

while t:

if not is\_leaf(t):

st.append(t.val)

if t.right:

t = t.right

else:

t = t.left

while st:

a = st.pop()

res.append(a)

return res

**Search Suggestions System**

1. Use Binary search to find substring in products

class Solution:

def search\_first\_min\_word(self, products, word):

s = 0

e = len(products) - 1

while s <= e:

m = s + (e - s)// 2

min\_len = min(len(word), len(products[m]))

prefix = products[m][:len(word)]

if prefix == word:

#return index of starting word

l = m - 1

while l >= 0 and len(products[l]) >= len(word) and products[l][:len(word)] == word:

l -= 1

l += 1

return l

if prefix < word:

s = m + 1

else:

e = m - 1

return -1

def suggestedProducts(self, products, searchWord):

if not products or not searchWord:

return []

products.sort()

res = []

for i in range(len(searchWord)):

sub = searchWord[:i+1]

j = self.search\_first\_min\_word(products, sub)

if j == -1:

res.append([])

else:

words = []

while j < len(products):

if len(words) < 3 and len(products[j]) >= len(sub) and products[j][:len(sub)] == sub:

words.append(products[j])

j += 1

else:

break

res.append(words)

return res

Search Suggestions System

1. Sort products
2. Cycle for each substring from 0..i, use bs to find index of min substring
3. Add result

class Solution:

def search\_first\_min\_word(self, products, word):

s = 0

e = len(products) - 1

while s <= e:

m = s + (e - s)// 2

min\_len = min(len(word), len(products[m]))

prefix = products[m][:len(word)]

if prefix == word:

#return index of starting word

l = m - 1

while l >= 0 and len(products[l]) >= len(word) and products[l][:len(word)] == word:

l -= 1

l += 1

return l

if prefix < word:

s = m + 1

else:

e = m - 1

return -1

def suggestedProducts(self, products, searchWord):

if not products or not searchWord:

return []

products.sort()

res = []

for i in range(len(searchWord)):

sub = searchWord[:i+1]

j = self.search\_first\_min\_word(products, sub)

if j == -1:

res.append([])

else:

words = []

while j < len(products):

if len(words) < 3 and len(products[j]) >= len(sub) and products[j][:len(sub)] == sub:

words.append(products[j])

j += 1

else:

break

res.append(words)

return res

**Number of Islands**

1. Use UnionFind:path compression and union by rank
2. Implement UnionFind
3. Go through cells: if cell is 1 and neighbours are 1 then union celll neighbours
4. In UnionFind it has component count: when it unions it will decrease component count.

class UnionFind:

def \_\_init\_\_(self, grid):

n = len(grid)

m = len(grid[0])

self.count = 0

self.parent = [0 for i in range(m \* n)]

for i in range(n):

for j in range(m):

if grid[i][j] == '1':

self.parent[i \* m + j] = i \* m + j

self.count += 1

self.rank = [0 for \_ in range(m \* n)]

def find(self, x):

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x])

return self.parent[x]

def union(self, x1, x2):

p1 = self.find(x1)

p2 = self.find(x2)

if p1 == p2:

return

if self.rank[p1] > self.rank[p2]:

self.parent[p2] = p1

elif self.rank[p2] > self.rank[p1]:

self.parent[p1] = p2

else:

self.parent[p1] = p2

self.rank[p2] += 1

self.count -= 1

class Solution:

def numIslands(self, grid):

if not grid:

return 0

n = len(grid)

m = len(grid[0])

uf = UnionFind(grid)

def to\_index(r, c):

return r \* m + c

for r in range(n):

for c in range(m):

if grid[r][c] == '0':

continue

if r - 1 >= 0 and grid[r - 1][c] == '1':

uf.union(to\_index(r, c), to\_index(r - 1, c))

if r + 1 < n and grid[r + 1][c] == '1':

uf.union(to\_index(r, c), to\_index(r + 1, c))

if c - 1 >= 0 and grid[r][c - 1] == '1':

uf.union(to\_index(r, c), to\_index(r, c - 1))

if c + 1 < m and grid[r][c + 1] == '1':

uf.union(to\_index(r, c), to\_index(r, c + 1))

grid[r][c] = '0'

return uf.count

**Reorder Data in Log Files**

1. Declare 2 items in result array
2. Go through logs and split log by space, compare ‘0’'<= arr[1][0]<= ‘9' and place it in respective result array item
3. Sort result array item with

Next Greater ElementII

1. Use stack to store confirmed next greater sequence
2. Run cycle for 2 \*n to fill up absent greater elements
3. If stack top <= current item let’s pop them
4. If stack not empty then stack top is next greater element for current item
5. Push current item

class Solution:

def nextGreaterElements(self, nums):

if not nums:

return []

st = []

res = [-1] \* len(nums)

for i in range(2 \* len(nums) - 1,-1,-1):#2\*n because we have to consider greater max from the left

j = i % len(nums)

num = nums[j]

while st and nums[st[-1]] <= num: #pop items <= current item

st.pop()

if st:

res[j] = nums[st[-1]]

# add next greater for the next element

st.append(j)

return res

**Next Greater Element I**

1.Use stack to keep up the decreasing order to items of nums2

2. If stack top < nums2[i] pop up stack items < nums2[i] - it is next greater element of them and put them in map: element->next greater element

3. Go through nums1 and if it has next greater element in map then add it to result array.

**Design Hit Counter**

1. Allocate 2 arrays: hits and seconds
2. Add to hits count of hits if (timestamp - 1) % 300 == seconds[j] otherwise hits[j] = 1
3. When getHits min\_t = max(0, timestamp - 300)
4. Sum up hits if min\_t < seconds[j] <= max\_t

**Frog Jump**

1. Use dynamic programming: store for each stone the jump size that leads to this stone.
2. On every iteration update stones[j] = stones[i] + k,stones[i] + k - 1,stones[i] +k -1 .
3. If last stone contains some jumpsize let’s return True

[Minimum Cost to Connect Sticks](https://leetcode.com/problems/minimum-cost-to-connect-sticks)

1. Connect sticks with minimal lengths
2. Use minimum priority queue
3. Until sticks contains greater than 1 pop sticks with min length and connect, push them back.

**Theasure Island:**

1.Use BFS, declare the array visited, queue

2. Append row,col,step in queue, neighbours if it’s ‘0’.

3. Dequeue row,col,step and it it’s treausure it returns step.

**Theasure IslandII:**

1.Use BFS where few source and few treasures, but with array path\_len that hold minimal path Len to [i,j] even though the paths will be overlaped.

2. Add all source to queue

3. Dequeue cell and if it’s treasure then continue otherwise append their neighbours

**Two Sum**

1. Use map to store element to index
2. Go through the numbers and if target-nums[i] in map then return index pair

[Copy List with Random Pointer](https://leetcode.com/problems/copy-list-with-random-pointer)

1. Store old node and respective new node in map to avoid node duplication
2. If node is in map we just return this node
3. Go through each node and clone node.next and node.random
4. Go to the next node in initial list

class Solution:

def \_\_init\_\_(self):

self.visited = {}

def copy\_node(self, node):

if node:

if node in self.visited:

return self.visited[node]

self.visited[node] = Node(node.val, None, None)

return self.visited[node]

return None

def copyRandomList(self, head: 'Node') -> 'Node':

if not head:

return head

p = head

new\_head = self.copy\_node(p)

while p:

new\_head.next = self.copy\_node(p.next)

new\_head.random = self.copy\_node(p.random)

new\_head = new\_head.next

p = p.next

return self.visited[head]

**Lowest Common Ancestor of a Binary Tree**

1. Use DFS to fill in the map parent:node reference to parent reference
2. Add all parent nodes to set for the node p
3. Go for q and if it has reference in parent of p then return it.

**Word Ladder**

1. Use BFS - we can try all paths with length K.
2. Encode every word by replace every char with ‘\*’: abc -> \*bc, a\*c, ab\* and put them to map as key and word as next candidate for transition
3. Dequeu word and generate all \*bc,a\*c,ab as key to look for next words.
4. In the end of every external cycle to reset all\_dict[temp] = [] to reduce working time.

**Find Median from Data Stream**

1. Use 2 priority queues: left is max PQ and right is min PQ
2. Len of left PQ = Len of right PQ+1
3. If left max is empty or num <= left max[0] let append it to left PQ otherwise append it to right PQ
4. Balance 2 PQS: if Left PQ > rightPQ+1 lets pop item from Left PQ and append it to right PQ. If Right PQ > left PQ let’s pop right PQ and append it to left PQ.
5. Return median = (left[0]+ right[0]/)2Len of left PQ + Len of right PQ == 0 otherwise return left[0]

 Subtree of Another Tree

1. Use 2 recursive functions: to find node is equal to subtree node and compare theirs nodes.
2. find\_subtree: if node is null return False, otherwise try to check if it’s subtree with current node or node.left or node.right
3. is\_subtree: if both trees are empty let’s return true, if either one node is empty let’s return false if values are the same call is\_subtree for both left and right subtrees of trees.

Search a 2D Matrix II

1. Start from left bottom corner and if cell[i][j] < target then j += 1 otherwise if cell[i][j] > target then I -= 1
2. If cel[i][j] == target then return true

Critical Connections in a Network

1. Use Tarjan algorithm, disc is time when we meet node at first time,
2. Low is minimum disc time of node from which we can reach current node
3. If disc[p] < low[child] p->child is bridge
4. Use dis to go: disc[p] = timer,low[p] = timer(time is incremented)
5. For each p neihgborh to run:

If v == pre: continue

If disc[v] == -1: - we is at first time here

Call dfs(v, prev,res)

Update low[p] = min(low[p],low[v])

If low[v] > disc[p]:

p-> v is critical connection!

Else:

low[p] = min(low[p],disc[v])

Run this for every node if disc[node] == -1

Favorite Genres

1.Map song to genre

2. Go through user songs and count songs belonging to genre

3. Filter out genres with count < max\_count

Spiral MatrixII

1.dx=0,dy = 1 use (I + dx) % n and (j+dy) % n assign A to I

2. If A[i][j] is not 0 let update dx = dy, dy= -dx

3. Update I and j by dx, dy => I += dx, j += dy

**Subarrays with K Different Integers**

1. Use slide window,
2. Count subarray for K different integer - count subarray for ( k-1)is result

3. Declare l and r indices, r go to the right and l = 0, map of char to its count

4. Update the char count by 1

5.if number of unique chars > K then increment l: if count of a char is 0 remove it from map

6. Res = r-l+1

**Sieve of Eratosthenes**

1. Use min heap to store [multiple, prime].
2. Start from 2, add [xˆ2,x] to min heap and output x as prime.
3. If I == minheap[0][0] then minheap[0][0] is composite and we pop, update it and push.

**Longest Word in Dictionary through Deleting**

1. Put string chars to map: char to index list with this char in string
2. Sort words by length, word if lengths are the same in reverse order
3. Run for each word from words:
4. For word’s char we look for index in map greater then last considered.
5. If no we return False otherwise try it for the rest of words

**Minimum Path Sum**

1. Use DP : start from dp[-1][-1]:
2. For every I,j go:

dp[i][j] = min(dp[i+1][j]+grid[i][j], dp[i][j+1]+grid[i][i]

dp[i][j] =dp[i+1][j]+grid[i][j] if j == len(grid[0])-1

dp[i][j] = dp[i][j+1]+grid[i][i] if i == len(grid[0])

dp[i][j] = grid[i][j]

3. Return dp[0][0]

**LongestPolindromicSubstring**

1. Consider every centre around which can be palindrom. 2n-1
2. For every char and two char we call extend\_aroun\_center
3. Choose maximum and update start and end of maximum palindrome

Start = i - (len-1)/2

End = I + len /2

class Solution:

def longestPalindrome(self, s: str) -> str:

if not s:

return s

def expand\_around\_center(l, r):

while l >= 0 and r < len(s) and s[l] == s[r]:

l -= 1

r += 1

return r - l - 1

start = 0

end = 0

for i in range(len(s)):

len1 = expand\_around\_center(i, i) # char is center

len2 = expand\_around\_center(i, i + 1)

max\_len = max(len1, len2)

if max\_len > end - start:

start = i - (max\_len - 1) // 2

end = i + max\_len // 2

return s[start:end + 1]

**Longest Palindromic Subsequence**

1. Use DP to store polindrom length for subsequence [i,j]
2. Set up dp[i][i] to 1
3. For length from 2 to len of s: if s[i] == s[j]: dp[i][j] = dp[i+1][j-1] + 2
4. Otherwise dp[i][j] = max(dp[i+1][j],dp[i][j-1]) and update maximum length

Substrings of size K with K distinct chars

1. Use two pointers technic
2. Allocate set for words and char\_freq[27] for chars
3. Increment char\_freq[ch]+=1 if char\_freq[ch] == 1 then increment uniq\_char
4. If j - I + 1 == k let do: if uniq\_char equals k add s[j:i+1] to result set and decrement char\_freq by 1, if char\_freq[ch] == 0 let decrement uniq\_char

**K Closest Points to Origin**

1. Use quick select
2. Choose pivot random and place items < pivot < items.
   1. If mid - i+1 > K then take (i, mid-1,K) otherwise if mid-i+1 < K take

(mid+1, j, K-(mid-i+1))

3. Partition items around pivot:

def sort(i, j, k):

if i > j:

return

mid = random.randint(i, j)

points[i], points[mid] = points[mid], points[i]

mid = partition(i, j)

if mid - i + 1 < k:

sort(mid + 1, j, k - (mid - i + 1))

elif mid - i + 1 > k:

sort(i, mid - 1, k)

def partition(i, j):

i1 = i

pivot = dist(i1)

i += 1

while True:

while i < j and dist(i) < pivot:

i += 1

while i <= j and dist(j) > pivot:

j -= 1

if i >= j:

break

points[i], points[j] = points[j], points[i]

points[i1], points[j] = points[j], points[i1]

return j

4.First K items are our answer

**Generate Parentheses**

1. Use recursive solution open and close are respective numbers of opening and closing parentheses.
2. If number in string = 2\*n let’s add this to result,

If open < n call function open+1, close, string + ‘(‘

If close < open let’s call function open,close+1, string +’)

**Connecting Cities With Minimum Cost**

1.Use Minimum spaning tree Kruskal algorithms

2. Sort edges by weight in increasing order

3. Use Disjoint Set Union to track if vertices are connected

4. Try to add every edge to graph if these vertices are not connected yet.

5. If so decrease component count by 1

6. If component count is equal 1 then we return common weight added edges.

**Prison Cells After N Days**

1. The code is repeating in 14 iterations, so apply such approach:
2. If N > 14 let’s N = N %14 +14
3. Else N = N % 14

class Solution:

def prisonAfterNDays(self, cells, N):

new\_cells = [0] \* 8

if N > 14:

N = N % 14 + 14

else:

N = N % 14

for day in range(N):

for i in range(1, 7):

new\_cells[i] = 0

if cells[i - 1] == cells[i + 1]:

new\_cells[i] = 1

new\_cells[0] = 0

new\_cells[7] = 0

cells, new\_cells = new\_cells, cells

if N % 2 == 0:

return cells

return cells

**Maximum Average Subtree**

1. Use post-order to traverse tree
2. If node is null let’s return [0,0,0] - [count,sum,average]
3. Call traverse for left and right subtrees, then recongnize witch average id maximum: current node, left or right ones.
4. Return [count,sum, average]

**Load balancer**

1. Use 2 pointers technic: if left sum <= right\_sum increase left pointer and left\_sum, otherwise increase right\_sum and right pointer
2. If mid\_sum lesser then left\_sum or right\_sum return False
3. If mid\_sum == left\_sum == right\_sum let’s return true
4. mid\_sum = total\_sum - left\_sum - right\_sum - arr[left] - arr[right]

Point of Lattice

1. Calculate dx = bx - ax, dy = by - by as consider it rotate triangle in counterclockwise
2. Rx = dy, ry = -dx rotate by 90 degrees
3. Contract rx,rx with gcd and add bx + rx, by + ry - to rotate the point B

**Merge Intervals**

1. Sort by start
2. Add item if res array is empty or if last res interval has end is lesser the interval start otherwise update res interval end = max(interval end, new interval end) - it merges 2 intervals in such way because start ALWAYS is lesser or equal

**Reorganize String**

1. Use max heap to put [count,char]
2. As long as heap is not empty , pop first and second chars and append them to result string
3. Decrease their frequencies and go on

Longest string made up of only vowels

1. Vvvv xxxx vvvv, xxxx vvvv xxxx, vvvvxxxxvvv xxxx vvvv are all cases to at most remove 2 substrings and leave vowels.
2. Start from begin as long as the char is vowel and start from end is doing the same, then find logest vowel substring from [start, end]
3. Return start + Len - start -1 + longest vowel substring

**Number of Dice Rolls With Target Sum**

1. Use rule sum to calculate way number
2. dp[sum][dice] = dp[sum-1][dice-1] + dp[sum-2][dice-1] + .. dp[sum-f][dice-1]

**Find the City With the Smallest Number of Neighbors at a Threshold Distance**

1. Use Floyd-Warshall algorithm: dist[i][j] = min(dist[i][j], dist[i][k]+dist[k][j]) i <= k<=j
2. Set up dist[i][i] = 0, dist[i][j] = w, where w is edge weight.
3. Count min cities for every city and return city with greater number id

Minimize Malware Spread II

1. Use DFS to find infected nodes
2. Create set from non-infected noded
3. Run DFS for every initial infected nodes: if node is not seen and is not-infected let’s add it to seen
4. Create map: not-infected node and list infected nodes
5. Go over this map if node has one indfected then increment map key(key is infected node)
6. Go over new map if choose the node with minimal id and maximal counter

Group Shifted Strings

1.Consider string as difference between consecutive letters, s[i]-s[i-1] so on if s[i]-s[i-1] < 0 we will add 26 here

2. Go through list and calculate key and add string with calculated key to list

**Kth Largest Element in an Array**

1. Use quick select
2. Partition array return p, where x[i] > x[p] > ..x[j]
3. If p - s + 1 == k let’s return p
4. If p - s+ 1 > k : call quick\_select(s,p-1,k), otherwise quick\_select(p+1,e, k - p + s -1)

**Basic Calculator II**

1. Use basic polish notation to discard parentethis
2. Use stack for operation and output buffer:
3. If chars is number let’s append it to output, otherwise it operation stack is not empty and top of stack is greater than current operation let’s pop it and append to output buffer
4. Iterate through output buffer: if there is number put it to stack otherwise pop two numbers from stack and make operation on them and put result to stack.
5. Return top of stack

**Concatenated Words**

1. Sort all words by length in increasing order

2. Add all words with first length to set

3. For the rest words call the dis: if word is conncatenated let’s add this word to result otherwise add this to set and add word length to array

4. DFS:

Go through array length:

Take substring with length from array and it this substring is presented in set call dis with rest part of word by passing parts number,length, array

**Last Stone Weight II**

1. Use dp to save i,j where I is stones[i] and j is sum.
2. S = S1+S2 where S1 and S2 is ether positive or negative, it needs to minimise S1 - S2 = S - S2 - S2 = S - 2 \* S2, let’s consider sum from 1 to S/2
3. dp[i][0] = 1for every stones
4. dp[i][s] = dp[i-1][s] or s>= stones[i-1](i starts from 0) and dp[i-1][s-stones[i-1]] > 0 with value is either true or false and S2 = max(S2, s)
5. Return S- 2 \* S2

**Time Based Key-Value Store**

1. Use map and value list (timestamp, value)
2. Use binary search to find value with ts <= is lesser than given one.

from collections import defaultdict

from bisect import bisect

class TimeMap:

def \_\_init\_\_(self):

self.key\_to\_timestamp = defaultdict(list)

def set(self, key, value, timestamp):

self.key\_to\_timestamp[key].append((timestamp, value))

def get(self, key, timestamp):

map = self.key\_to\_timestamp.get(key,None)

if map is None:

return ""

i = bisect(map, (timestamp, chr(127)))

return map[i-1][1] if i else “"

**Permutations:**

1.Use backtracking by exchange l with I and call function perm recursively

2. When l equals r let’s add array to result

class Solution:

def permutations(self, arr):

if not arr:

return

def perm(l, r, res):

if l == r:

res.append(arr[::])

return

for i in range(l, r + 1):

arr[i],arr[l] = arr[l],arr[i]

perm(l+1, r, res)

arr[i], arr[l] = arr[l], arr[i]

res = []

perm(0, len(arr) - 1, res)

return res

**Balanced parenthesis**

1. Use recursion to build all sequences
2. If close parent equals n let’s return new suquence
3. If close < open add ‘}’ and call function with open,close+1
4. If open < n add ‘{‘and call the function with open+1,close

**WordBreak**

1. Use dp: dp[0,i] = true if dp[j] and s[j:i] is in wordDict

2. The rest is as it’s in longest array subsequence

**Trapping Rain Water**

1. Use stack by pushing height <= then top stack
2. while current bar is higher then top of stack is doing:

- pop from stack

- if stack is empty break

- dist = current bar - left bar(top of stack) - 1

- H = min(current bar, left bar) - popped from stack

- cap += H \* dist

**Serialize and Deserialize Binary Tree**

1. Use DFS: if we meet None let’s add ‘null’ to bug

2. Append node.val, call for left and right subtrees

3. ‘[‘+ ‘,’.join(buf)+’]’

1. Deserialize with DFS:

If not but or buf[0] is ‘null’return None

Otherwise node = TreeNode(buf[0]), buf.pop(0)

node.left = dfs(buf)

node.right = dfs(buf)

**Word Break II**

1. Use DP: allocate dp = len of s
2. From 1 to len(s)+1:

dp[j] = 1 if dp[i] = true and s[i:j] in wordDict, add start position this word to map

3. Use DFS traverse from 0 to len(found words) if I == len(s) to add buffer to result array

Time complexity O(Nˆ2 + 2ˆN + W)

Memory complexity O(Nˆ2 + 2ˆN +W)

**Word Search II**

1. Use backtracking
2. Store word in trie
3. If word is added to result let’s remove this word from trie
4. Mark char as used in backtracking before, then recover it

class Trie:

def \_\_init\_\_(self):

self.chars = {}

def add(self, s):

m = self.chars

for ch in s:

m = m.setdefault(ch, {})

m['#'] = s

class Solution:

def findWords(self, board, words):

# if not words or not board:

# return []

trie = Trie()

for word in words:

trie.add(word)

def backtracking(r,c, parent, res):

letter = board[r][c]

curnode = parent[letter]

if '#' in curnode:

res.append(curnode['#'])

board[r][c] = '{}'

for dr, dc in [[0, 1], [0, -1], [1, 0], [-1, 0]]:

r1 = r + dr

c1 = c + dc

if 0 <= r1 < len(board) and 0 <= c1 < len(board[0]):

ch = board[r1][c1]

if ch in curnode:

backtracking(r1, c1, curnode, res)

board[r][c] = letter

if '#' in curnode:

curnode.pop('#')

res = []

for i in range(len(board)):

for j in range(len(board[0])):

ch = board[i][j]

if ch in trie.chars:

backtracking(i, j, trie.chars, res)

return res

**LFU Cache**

from collections import defaultdict

class Node:

def \_\_init\_\_(self, key, value):

self.key = key

self.value = value

self.freq = 1

self.prev = None

self.next = None

class DLinkedList:

def \_\_init\_\_(self):

self.sentinel = Node(None, None)

self.size = 0

self.sentinel.next = self.sentinel.prev = self.sentinel

def \_\_len\_\_(self):

return self.size

def append(self, node):

node.next = self.sentinel.next

node.next.prev = node

node.prev = self.sentinel

self.sentinel.next = node

self.size += 1

def pop(self, node=None):

if self.size == 0:

return

if not node:

node = self.sentinel.prev

node.prev.next = node.next

node.next.prev = node.prev

self.size -= 1

return node

class LFUCache:

def \_\_init\_\_(self, capacity):

self.vals = dict()

self.freq = defaultdict(DLinkedList)

self.capacity = capacity

self.min\_freq = 0

self.size = 0

def update(self, node):

freq = node.freq

self.freq[freq].pop(node)

if freq == self.min\_freq and not self.freq[freq]:

self.min\_freq += 1

node.freq += 1

freq = node.freq

self.freq[freq].append(node)

def get(self, key):

if self.size == 0 or key not in self.vals:

return -1

node = self.vals[key]

self.update(node)

return node.value

def put(self, key, value):

if self.capacity == 0:

return

if key in self.vals:

node = self.vals[key]

self.update(node)

node.value = value

else:

if self.size == self.capacity:

node = self.freq[self.min\_freq].pop()

del self.vals[node.key]

self.size -= 1

node = Node(key, value)

self.size += 1

self.vals[key] = node

self.freq[1].append(node)

self.min\_freq = 1

**Longest Substring Without Repeating Characters**

1. Use time window to put in char in map with char -> pos

2. If we meet the repetitive char to get max(start, map[char])

3. Set up max\_len = max(max\_len, j - I +1)

4. Put in new char m[char] = j + 1

**Minimum Window Substring**

1. Use min window by filling till the this window desn’t contains all chars of t. After in while cycle decrease char frequency and if m[ch] < chars[ch] then formed -= 1

from collections import Counter

from collections import defaultdict

class Solution:

def minWindow(self, s, t):

if not s and not t:

return t

if not s or not t:

return ""

min\_win = [float('inf'), None, None]

chars = Counter(t)

need = len(chars)

win = defaultdict(int)

left = 0

formed = 0

for right in range(len(s)):

ch = s[right]

win[ch] += 1

if ch in chars and win[ch] == chars[ch]:

formed += 1

while left <= right and formed == need:

if min\_win[0] > right - left + 1:

min\_win = [right - left + 1, left, right]

win[s[left]] -= 1

if s[left] in chars and win[s[left]] < chars[s[left]]:

formed -= 1

left += 1

formed, l, r = min\_win

return s[l: r + 1] if formed != float('inf') else ‘'

**Find All Anagrams in a String**

1. Use window to store chars to map where char is key and count is value.
2. If i>= len(p) then remove left side chars, s[i-len(p)] if its count is 1 otherwise decrement it
3. Add i-len(p)+1 as new index

from collections import Counter

from collections import defaultdict

class Solution:

def findAnagrams(self, s, p):

if not s or not p or len(s) < len(p):

return []

pat = Counter(p)

win = defaultdict(int)

res = []

for i in range(len(s)):

win[s[i]] += 1

# remove leftside chars

if i >= len(p):

if win[s[i - len(p)]] == 1:

del win[s[i - len(p)]]

else:

win[s[i - len(p)]] -= 1

if win == pat:

res.append(i - len(p) + 1)

return res

**Text Justification**

1. Consider 3 cases: last sentence contains one space delimiter between words and pad with space to the right

1 word sentence is padded with spaces to the right, other sentence has spaces evenly distrubuted between words,

Spaces = maxWidth - all word lengths, space/ (number of words - 1) + add by one space to every delimiter.

class Solution:

def format\_text(self, maxWidth, words, sentence, cur\_len):

if len(sentence) == 1:

word = words[sentence[0]]

return ''.join([word] + ([' '] \* (maxWidth - len(word))))

new\_sentence = []

is\_last = sentence[-1] == (len(words) - 1)

total\_spaces = maxWidth - cur\_len

num\_space = total\_spaces // (len(sentence) - 1)

space = ' ' if is\_last else (' ' \* num\_space)

rem = 0 if is\_last else total\_spaces % (len(sentence) - 1)

for j in range(len(sentence)):

if j > 0:

new\_sentence.append(space)

if rem > 0:

new\_sentence.append(' ')

rem -= 1

new\_sentence.append(words[sentence[j]])

right\_spaces = maxWidth - sum([len(w) for w in new\_sentence])

return ''.join(new\_sentence + ([' '] \* right\_spaces))

def fullJustify(self, words, maxWidth):

res = []

sentence = []

cur\_len = 0

for i in range(len(words)):

spaces = len(sentence)

if cur\_len + spaces + len(words[i]) <= maxWidth:

cur\_len += len(words[i])

sentence.append(i)

else:

res.append(self.format\_text(maxWidth, words, sentence, cur\_len))

sentence = [i]

cur\_len = len(words[i])

if sentence:

res.append(self.format\_text(maxWidth, words, sentence, cur\_len))

return res

**Maximal Rectangle**

1. Use dp by stash them heigh row by row
2. And then calculate area of this and take maximal of them

3. Go row by row and update dp if matrix[i][j] == 1 set it up dp[i][j] += 1 else dp[i][j] = 0

class Solution:

def find\_max\_hist(self, hist):

st = [-1]

max\_area = 0

for i in range(len(hist)):

while st[-1] != -1 and hist[st[-1]] >= hist[i]:

j = st.pop()

max\_area = max(max\_area, hist[j] \* (i - st[-1] - 1))

st.append(i)

while st[-1] != -1:

max\_area = max(max\_area, hist[st.pop()] \* (len(hist) - st[-1] - 1))

return max\_area

def maximalRectangle(self, matrix):

if not matrix:

return 0

n = len(matrix)

m = len(matrix[0])

dp = [0] \* m

max\_area = 0

for i in range(n):

for j in range(m):

if matrix[i][j] == '1':

dp[j] = dp[j] + 1

else:

dp[j] = 0

max\_area = max(max\_area, self.find\_max\_hist(dp))

return max\_area

**K-Similar Strings**

1.Use BFS, time complexity O(N choose K) \* (2 ^ K)

from collections import deque

class Solution:

def kSimilarity(self, A, B):

def neighbours(S):

for i in range(len(S)):

if S[i] != B[i]:

break

arr = list(S)

for j in range(i + 1, len(S)):

if S[j] == B[i]:

arr[j], arr[i] = arr[i], arr[j]

yield "".join(arr)

arr[j], arr[i] = arr[i], arr[j]

seen = {A: 0}

q = deque([A])

while q:

s = q.popleft()

if s == B:

return seen[s]

for nei in neighbours(s):

if nei not in seen:

seen[nei] = seen[s] + 1

q.append(nei)

**Largest Rectangle in Histogram**

1. Use stack to store value > a[i], until a[i] > stack[pop],otherwise if arr[stack[-1]] >= arr[i], a[i] is new centre, and pop out all previous area = height[stack[-1]]\*(i - stack[-1] - 1),/height[stack[-1]\* (len(arr) - stack[-1] - 1)

class Solution:

def largestRectangleArea(self, heights):

max\_area = 0

st = [-1]

for i in range(len(heights)):

while st[-1] != - 1 and heights[st[-1]] >= heights[i]:

max\_area = max(max\_area, heights[st.pop()] \* (i - st[-1] - 1))

st.append(i)

while st[-1] != -1:

max\_area = max(max\_area, heights[st.pop()] \* (len(heights) - st[-1] - 1))

return max\_area

**Longest String Chain**

1. Use DP as longest increasing subsequence
2. Use remove char not insert)

from collections import defaultdict, Counter

class Solution:

def longestStrChain(self, words):

if not words or len(words) == 0:

return 0

def differs\_by\_one(word, subword):

for i in range(len(word)):

cand = word[0:i] + word[i + 1:]

if cand == subword:

return True

return False

word\_by\_len = defaultdict(list)

for word in words:

word\_by\_len[len(word)].append(word)

max\_len = 1

length = {word: 1 for word in words}

for k in range(min(word\_by\_len.keys())+1, max(word\_by\_len.keys()) + 1):

for word in word\_by\_len[k]:

for subword in word\_by\_len[k - 1] :

if differs\_by\_one(word, subword) and length[word] < length[subword] + 1:

length[word] = max(length[word], length[subword] + 1)

max\_len = max(max\_len, length[word])

return max\_len

**Shortest Path with Alternating Colors**

1. Use 2 ans subarrays :[[,], [,] …[,]] 0 is for red and 1 is for blue edges.
2. In the choose min of 2 items in subarrays.

from collections import defaultdict

class Solution:

def shortestAlternatingPaths(self, n, red\_edges, blue\_edges):

if not red\_edges and not blue\_edges:

ans = [-1]\*n

ans[0] = 0

return ans

MAX = float('inf')

adj\_list = defaultdict(list)

# path\_len = [[MAX] \* n for \_ in range(n)]

# 0 is red,1 is blue,2 is gray

for x, y in red\_edges:

adj\_list[x].append([0, y])

for x, y in blue\_edges:

adj\_list[x].append([1, y])

ans = [[MAX, MAX] for \_ in range(n)]

ans[0][0] = 0

ans[0][1] = 0

q = [[0, 0], [0, 1]] # node,clr

while q:

node, clr = q.pop(0)

for nei\_clr, nei in adj\_list[node]:

if nei\_clr == 1 - clr and ans[nei][nei\_clr] > ans[node][clr] + 1:

ans[nei][nei\_clr] = ans[node][clr] + 1

q.append([nei, nei\_clr])

return [min(a1,a2) if min(a1,a2) != MAX else -1 for a1,a2 in ans]

**Construct Binary Tree from Inorder and Postorder Traversal**

1. Use postorder as node val going from the end,
2. Use ignorer to split it on 2 parts: left subtree, node, right subtree. Node is taken from postorder

class Solution:

def buildTree(self, inorder, postorder):

if not inorder and not postorder:

return None

node\_pos = {x: i for i, x in enumerate(inorder)}

last\_node = len(postorder) - 1

def traverse(node\_pos, s, e):

nonlocal last\_node

if s > e:

return None

x = postorder[last\_node]

last\_node -= 1

new\_s = node\_pos[x]

node = TreeNode(x)

node.right = traverse(node\_pos, new\_s + 1, e)

node.left = traverse(node\_pos, s, new\_s - 1)

return node

return traverse(node\_pos, 0, len(inorder) - 1)

**Minimum Window Subsequence**

1.Use DP, if S[j] == T[i] then if i== 0 dp[j] = j otherwise

new[j] = dp[j-1], if S[j]!= T[i] if j > 0 then

new[j] = new[j-1]

Where dp is start position for S at previous iteration and new is being formed new dp

class Solution:

def minWindow(self, S, T):

if not S and not T or len(T) > len(S):

return ""

dp = [i if S[i] == T[0] else -1 for i in range(len(S))]

left = -1

for i in range(len(T)):

k = -1

new = [-1] \* len(S)

for j in range(left + 1, len(S)):

if S[j] == T[i]:

if i == 0:

new[j] = j

else:

new[j] = dp[j - 1]

if k == -1:

k = j

elif j > 0:

new[j] = new[j - 1]

if k != -1:

left = k

dp = new

res = [0, float('inf')]

for j in range(0, len(S)):

if dp[j] > -1:

s, e = dp[j], j

if res[1] - res[0] > e - s:

res = [s, e]

return S[res[0]:res[1] + 1] if res[1] != float('inf') else ‘'

Knapsack:

1. Use DP, row is things and col is value/weight
2. dp[i][w] = max(dp[i-1][w],dp[i-1][w-wt[i-1]]+value[i-1]] otherwise dp[i][w] = dp[i-1][w]

def knapsack(W, val, wt):

n = len(val)

dp = [[0] \* (W + 1) for \_ in range(n + 1)]

for i in range(n + 1):#try every i in line

for w in range(W + 1):

if wt[i - 1] <= w:

dp[i][w] = max(dp[i - 1][w - wt[i - 1]] + val[i - 1], dp[i - 1][w])

else:

dp[i][w] = dp[i - 1][w]

return dp[n][W]

**Network Delay Time**

1. Use Dijkstra

from heapq import heappop, heappush

from collections import defaultdict

MAX = float('inf')

class Solution:

def networkDelayTime(self, times, N, K):

if not times or N == 0:

return -1

adj\_list = defaultdict(list)

for s, e, t in times:

adj\_list[s].append([e, t])

q = [[0, K]]

dist = [MAX] \* (N + 1)

dist[0] = 0

dist[K] = 0

while q:

cur,node = heappop(q)

for nei,t in adj\_list[node]:

if cur + t < dist[nei]:

dist[nei] = cur + t

heappush(q, [dist[nei], nei])

return max(dist) if all([True if x != MAX else False for x in dist]) else -1

**Meeting Rooms II**

0. Sort item by start time, add first end time to min-heap

1. Use min-heap to store end event to hold actual room number
2. If new meeting start >= min-heap top, then pop item, add new end time to min-heap
3. Len of min-heap is room number

**Minimum Domino Rotations For Equal Row**

1. Count not equal tiles to x : if A[i] != x: swap\_a+=1, if B[i] != x: swap\_b+=1
2. Return min(swap\_a, swap\_b)
3. Call it for A[0] and if check(A[0} == -1 let’s call it for B[0]

**Decode String**

1. Compose distinct functions word, rep, rep\_or\_word

class Solution:

def \_\_init\_\_(self):

self.pos = 0

def is\_digit(self, ch):

return '0' <= ch <= '9'

def is\_char(self, ch):

return 'a' <= ch <= 'z' or 'A' <= ch <= 'Z'

def word(self, s):

buf = []

while self.pos < len(s) and self.is\_char(s[self.pos]):

buf.append(s[self.pos])

self.pos += 1

return ''.join(buf)

def rep(self, s):

num = []

while self.is\_digit(s[self.pos]) and self.pos < len(s):

num.append(s[self.pos])

self.pos += 1

num = int(''.join(num))

self.pos += 1 # for [

# processing rest chars inside []

next\_term = self.rep\_or\_word(s)

if self.pos < len(s) and s[self.pos] == ']':

self.pos += 1

return next\_term \* num

def rep\_or\_word(self, s):

res = ''

while self.pos < len(s) and s[self.pos] != ']':

if self.is\_char(s[self.pos]):

res += self.word(s)

else:

res += self.rep(s)

return res

def decodeString(self, s):

if not str:

return ""

self.pos = 0

return self.rep\_or\_word(s)

**Maximize Distance to Closest Person**

1. Take indices where person is
2. max\_dist = max(person[0],n -person[-1] - 1)

for i in range(1, len(persons)):

dist = persons[i] - persons[i - 1] - 1

if dist % 2 == 1:

dist = (dist + 1) // 2

else:

dist = dist // 2

max\_dist = max(max\_dist, dist)

**Rotated Digits**

1. Put in rotated numbers to set
2. Put in bad numbers to other set
3. If num < 10 and in rotated set, let’s increment valid\_num
4. If num contains bad\_number skip this, if it contains rotated numbers - update valid\_num

class Solution:

def rotatedDigits(self, N: int):

valid = 0

nums = set([2,5,6,9])

bad\_nums = set([3,4,7])

for a in range(1, N+1):

if a in nums:

valid += 1

else:

rot = True

changed = False

while a > 0:

b = a % 10

if b in nums:

changed = True

if b in bad\_nums:

rot = False

break

a = a // 10

if rot and changed:

valid += 1

return valid

**Longest Absolute File Path**

class Solution:

def lengthLongestPath(self, input: str) -> int:

inp = input.split("\n")

stack = [] # will contain (dirname, level)

op = [0] # for empty result like "a"

for tm in inp:

subs = tm.split('\t')

c = len(subs)

last = subs[-1]

if c == 1 and last[:4] == ' ':

last = last[4:]

else:

while stack and stack[-1][1] >= c:

stack.pop()

if '.' in last:

tmp = ''

for k in stack:

tmp += '/' + k[0]

tmp += last

op.append(len(tmp))

else:

stack.append([last, c])

return max(op)

**Guess the Word**

1. **Use minimal by minimise maximum word candidates**

**class Solution(object):**

**def findSecretWord(self, wordlist, master):**

**n = len(wordlist)**

**self.h = [[0] \* n for \_ in range(n)]**

**for i in range(n):**

**for j in range(n):**

**c = 0**

**for k in range(len(wordlist[0])):**

**if wordlist[i][k] == wordlist[j][k]:**

**c += 1**

**self.h[i][j] = c**

**possible = [i for i in range(n)]**

**path = set()**

**while possible:**

**guess = self.solve(possible, path)**

**match = master.guess(wordlist[guess])**

**if match == len(wordlist[0]):**

**return len(wordlist[0])**

**possible2 = []**

**for j in possible:**

**if match == self.h[guess][j]:**

**possible2.append(j)**

**possible = possible2**

**path.add(guess)**

**return -1**

**def solve(self, possible, path = ()):**

**if len(possible) <= 2:**

**return possible[0]**

**ansgrp = possible**

**ans = -1**

**for guess in range(len(self.h)):**

**if guess not in path:**

**groups = [[] for \_ in range(7)]**

**for j in possible:**

**if j != guess:**

**groups[self.h[guess][j]].append(j)**

**maxgroup = max(groups, key = len)**

**if len(maxgroup) < len(ansgrp):**

**ansgrp = maxgroup**

**ans = guess**

**return ans**

**Next Closest Time**

1. **Simulae time elapsing**

class Solution:

def nextClosestTime(self, time):

if not time:

return time

cur = 60 \* int(time[0:2]) + int(time[3:])

allowed = {int(c) for c in time if c != ':'}

while True:

cur = (cur + 1) % (24 \* 60)

digits = [cur // 60 // 10, cur // 60 % 10, cur % 60 // 10, cur % 60 % 10]

for digit in digits:

if all(digit in allowed for digit in digits):

return '{}{}:{}{}'.format(digits[0], digits[1], digits[2], digits[3])

**K Empty Slots**

1. Allocate days[bulbs-1] = I + 1(day)
2. If days[left] and days[right] are minimum, then ans = min(ans, max(days[left], days[right]))

class Solution(object):

def kEmptySlots(self, bulbs, k):

days = [0] \* len(bulbs)

for i in range(len(bulbs)):

days[bulbs[i] - 1] = i + 1

ans = float('inf')

left, right = 0, k + 1

while right < len(days):

for i in range(left + 1, right):

if days[i] < days[left] or days[i] < days[right]:

left, right = i, i + k + 1

break

else:

ans = min(ans, max(days[left], days[right]))

left, right = right, right + k + 1

return ans if ans < float('inf') else -1

**Flower Planting With No Adjacent**

1. For every garden consider how it neighbour is planted and choose free flower to take it.

from collections import defaultdict

class Solution:

def gardenNoAdj(self, N, paths):

ans = [0] \* N

adj\_list = defaultdict(list)

for x, y in paths:

adj\_list[x].append(y)

adj\_list[y].append(x)

for i in range(1, N + 1):

clr = [0] \* 4

for x in adj\_list[i]:

clr[ans[x-1]] = 1

for j in range(1,5):

if clr[j] == 0:

ans[i-1] = j

break

return ans

**Largest Time for Given Digits**

1. Use permutation of 4 digits

class Solution:

def largestTimeFromDigits(self, A):

def gen\_time(digits, start):

if start == 4:

h = 10\*digits[0] + digits[1]

m = 10\*digits[2] + digits[3]

if 0 <= h<24 and 0 <= m < 60:

return digits[::]

return None

for i in range(4):

digits[start],digits[i] = digits[i],digits[start]

fit\_time = gen\_time(digits,start + 1)

if fit\_time:

return fit\_time

digits[start],digits[i] = digits[i],digits[start]

return None

ans = gen\_time(A, 0)

if ans:

return "{}{}:{}{}".format(ans[0],ans[1],ans[2],ans[3])

return “"

**Range Sum Query - Mutable**

1. Use segment tree

class NumArray:

def \_\_init\_\_(self, nums):

self.n = len(nums)

self.tree = [0] \* (2 \*self.n)

i = self.n

j = 0

while i < 2\*self.n:

self.tree[i] = nums[j]

i += 1

j += 1

i = self.n-1

while i>= 0:

self.tree[i] = self.tree[2\*i]+self.tree[2\*i + 1]

i -= 1

def update(self, pos: int, val: int) -> None:

pos += self.n

self.tree[pos] = val

while pos > 0:

left = pos

right = pos

if pos % 2 == 0:

right = pos + 1

else:

left = pos - 1

self.tree[pos//2] = self.tree[left] + self.tree[right]

pos = pos // 2

def sumRange(self, l, r):

l += self.n

r += self.n

sum = 0

while l <= r:

if r % 2 == 0:

sum += self.tree[r]

r -= 1

if l % 2 == 1:

sum += self.tree[l]

l += 1

l = l // 2

r = r // 2

return sum

**Longest String Chain**

1. Use dp LIS:

from collections import defaultdict, Counter

class Solution:

def longestStrChain(self, words):

if not words or len(words) == 0:

return 0

def differs\_by\_one(word, subword):

for i in range(len(word)):

cand = word[0:i] + word[i + 1:]

if cand == subword:

return True

return False

words.sort(key=lambda x: len(x))

dp = [1] \* len(words)

max\_len = 1

for i in range(1, len(words)):

for j in range(i-1, -1, -1):

if len(words[i]) == len(words[j]) + 1:

if differs\_by\_one(words[i], words[j]):

dp[i] = max(dp[i], dp[j] + 1)

max\_len = max(max\_len, dp[i])

elif len(words[i]) > len(words[j]) + 1:

break

return max\_len

**Minimum Knight Moves**

1. Use BFS, but positive qaudrant x >= -2 and y >= -2 and consider abs(x), abs(y)

from collections import defaultdict

class Solution:

def minKnightMoves(self, x: int, y: int) -> int:

if x == 0 and y == 0:

return 0

seen = defaultdict(dict)

seen[0][0] = 1

q = [[0, 0, 0]]

dx = [2, -2, 2, -2, 1, -1, 1, -1]

dy = [1, 1, -1, -1, 2, 2, -2, -2]

while q:

m, x1, y1 = q.pop(0)

if x1 == abs(x) and y1 == abs(y):

return m

for i in range(8):

x2 = x1 + dx[i]

y2 = y1 + dy[i]

if abs(x2) + abs(y2) > 300 or x2 in seen and y2 in seen[x2] or x2 < -2 or y2 < -2:

continue

q.append([m + 1, x2, y2])

seen[x2][y2] = 1

**Backspace String Compare**

1. Use genetor to return char by char instead of whole string
2. Use itertools.zip\_longest

import itertools

class Solution:

def backspaceCompare(self, S: str, T: str) -> bool:

if not S and not T:

return True

if not S or not T:

return False

def process(s):

skip = 0

for ch in reversed(s):

if ch == '#':

skip += 1

elif skip:

skip -= 1

else:

yield ch

# iterate until longest iterable is exhausted by replacing shortes char with None

return all(a == b for a, b in itertools.zip\_longest(process(S), process(T)))

**Longest Line of Consecutive One in Matrix**

1. Use dp to calculate maximum distinctly for horizontal, vertical, diagonal, anti-diagonal matrix

class Solution:

def longestLine(self, M):

if not M or len(M[0]) == 0:

return 0

n = len(M)

m = len(M[0])

# hor,vert, diag, anti-diagonal

# 3 dimensional array

dp = [[[0] \* 4 for \_ in range(m)] for \_ in range(n)]

max\_len = 0

for i in range(n):

for j in range(m):

if M[i][j] == 1:

# define for horizontal

dp[i][j][0] = 1

if j > 0:

dp[i][j][0] = dp[i][j - 1][0] + 1

dp[i][j][1] = 1

if i > 0:

dp[i][j][1] = dp[i - 1][j][1] + 1

dp[i][j][2] = 1

if i > 0 and j > 0:

dp[i][j][2] = dp[i - 1][j - 1][2] + 1

dp[i][j][3] = 1

if i > 0 and j + 1 < m:

dp[i][j][3] = dp[i - 1][j + 1][3] + 1

# maximum among 4 values

max\_len = max(max\_len, dp[i][j][0], dp[i][j][1], dp[i][j][2], dp[i][j][3])

return max\_len

**Strobogrammatic Number**

1. If all chars in [0,1,8,6,9]
2. Use 2 pointers approach
3. If a== b and a is 0,1,8 or a+b == 69| 96
4. Otherwise return False
5. Check char in the middle of string is 0,1,8

from collections import Counter

class Solution:

def isStrobogrammatic(self, num: str) -> bool:

if not num:

return False

# use 2 pointers approach

i = 0

j = len(num) - 1

allowed = set(['0','1','8','6','9'])

if not all(ch in allowed for ch in num):

return False

self\_strobo = set(['0','1','8'])

while i < j:

a = num[i]

b = num[j]

# if two numbers are same or 6 == 9, 0 == 0, 8 == 8

if a == b and a in self\_strobo or a =='6'and b=='9' or a=='9' and b=='6':

i += 1

j -= 1

continue

else:

return False

# check that number in the middle is self-strobogramatic

return num[i] in self\_strobo if len(num) == 1 else True

**Wiggle Sort**

1. Sort
2. Swap items from 1 till n - 1with step 2

class Solution:

def wiggleSort(self, nums):

if not nums:

return nums

# it should have even lengt

nums.sort()

for i in range(1, len(nums)-1, 2):

#exchange neigbour with step 2

nums[i],nums[i+1] = nums[i+1],nums[i]

return nums

**Design Phone Directory**

1. Use set pool of number from 0 till maxNumber

class PhoneDirectory:

def \_\_init\_\_(self, maxNumbers: int):

self.pool = set([i for i in range(maxNumbers)])

def get(self) -> int:

return self.pool.pop() if self.pool else -1

def check(self, number: int) -> bool:

return number in self.pool

def release(self, number: int) -> None:

self.pool.add(number)

**Delete Nodes And Return Forest**

1. Go recursive to left subtree and right one
2. If node is in to\_deleted, append their node.left!= null, node.right != null to forest
3. Return None
4. Otherwise return Node

class Solution:

def delNodes(self, root: TreeNode, to\_delete):

if not root:

return None

forest = []

to\_del = set(to\_delete)

def traverse(node, to\_del, forest):

if not node:

return

# call it for left,right subtrees

node.left = traverse(node.left, to\_del, forest)

node.right = traverse(node.right, to\_del, forest)

# if node is in to\_delete

if node.val in to\_del:

# check if it's not empty node

if node.left:

forest.append(node.left)

if node.right:

forest.append(node.right)

return None

return node

traverse(root, to\_del, forest)

if root.val not in to\_del:

forest.append(root)

return forest

**Knight Probability in Chessboard**

1. Use Dp to calculate all probability: x/8/8 and so on p[r+dr][c+dc]+= p[r][c] \* 1/8

class Solution(object):

def knightProbability(self, N, K, r, c):

dp = [[0] \* N for \_ in range(N)]

dp[r][c] = 1

for \_ in range(K):

#allocate new array

dp2 = [[0] \* N for \_ in range(N)]

for r, row in enumerate(dp):

for c, val in enumerate(row):

for dr, dc in ((2,1),(2,-1),(-2,1),(-2,-1),

(1,2),(1,-2),(-1,2),(-1,-2)):

if 0 <= r + dr < N and 0 <= c + dc < N:

# calculate probality to be at [r+dr][c+dc]

# we may put here many times,so there are mutually exclusive so sum up their probalities

dp2[r+dr][c+dc] += val / 8.0

# make new one as current

dp = dp2

return sum(map(sum, dp))

**Minimum stack implementation**

1.Encode = 2 \* new\_min\_value - old\_min\_value

2. Push it, there encoded < current min\_value

3. If we pop value < current\_min\_value, encode it

As 2 \* current\_min\_val - popped value

class MinStack:

def \_\_init\_\_(self):

# store minimum value

self.min\_val = 0

self.stack = []

def push(self, x):

if len(self.stack) == 0:

#if stack is empty, our min value is x

self.stack.append(x)

self.min\_val = x

elif x < self.min\_val:

# if x is new min value, encode old min value as 2\*x - current min\_val

y = 2 \* x - self.min\_val

# push it to stack

self.stack.append(y)

# assigne new min value

self.min\_val = x

else:

# otherwise push value to stack

self.stack.append(x)

def pop(self):

if len(self.stack) == 0:

return

y = self.stack.pop()

# if y is < current min value, decode it as new min value

if y < self.min\_val:

# new min item

self.min\_val = 2 \* self.min\_val - y

def top(self):

if self.stack[-1] < self.min\_val:

return self.min\_val

return self.stack[-1]

def getMin(self):

if len(self.stack) == 0:

return None

return self.min\_val

**Moving Average from Data Stream**

1. Append val to array
2. If len(array)== given size, let pop(0)
3. Return sum(array) / len(array) if arrau is not empty,otherwise 0

**Missing Ranges**

1.if b - a > 1, add range

2. If lower < nums[0],insert (0,[lowe,nums[0]-1]

3. If upper > nums[-1], insert (nums[-1]+1,upper)

**Add Bold Tag in String**

1. Use intervals to merge them.
2. If r[0] <= new\_range[-1][1], update new\_range[-1][1] = max(r[1], new\_range[-1][1]), otherwise append r to new\_range

class Solution:

def addBoldTag(self, s: str, dict):

if not s:

return None

ranges = []

for word in dict:

i = s.find(word)

if i == -1:

continue

while i != -1 and i < len(s):

j = i + len(word)

ranges.append([i, j])

i = s.find(word, j)

ranges.sort(key=lambda x: x[0])

# try merge last range and new one

new\_ranges = []

for r in ranges:

if len(new\_ranges) > 0 and r[0] <= new\_ranges[-1][1]:

new\_ranges[-1][1] = max(r[1], new\_ranges[-1][1])

else:

new\_ranges.append(r)

res = []

start = 0

for s1, e1 in new\_ranges:

end = s1

res.append(s[start:end])

res.append(f"<b>{s[s1:e1]}</b>")

start = e1

res.append(s[start:])

return ‘'.join(res)

**Minimum Area Rectangle**

1. Add points to map[x] -> list y
2. Sort this map by x
3. Go over map:
4. Sort map[x] in y\_list
5. Go through every y and consider y1 lesser than y
6. If vert\_lines[(y,y1) exists, compare it with existing in vert\_line, if found it will calcuate new min area.

**Longest Substring with At Most K Distinct Characters**

1. Use sliding window
2. Count unique chars
3. If unique\_chars > K, increment begin

from collections import defaultdict

class Solution:

def lengthOfLongestSubstringKDistinct(self, s, k):

if not s or k == 0:

return 0

chars = defaultdict(int)

counter = 0

b = 0

max\_len = 0

for e in range(len(s)):

ch = s[e]

chars[ch] += 1

if chars[ch] == 1:

counter += 1

while counter > k:

ch = s[b]

if ch in chars:

chars[ch] -= 1

if chars[ch] == 0:

counter -= 1

b += 1

if counter <= k:

max\_len = max(max\_len, e - b + 1)

return max\_len

**Longest Repeating Character Replacement**

1. Use sliding window, max\_value is max of chars
2. If e - s + 1 - max\_value > k, increase b by 1

class Solution:

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

max\_len = 0

max\_chars = 0

chars = defaultdict(int)

b = 0

for e in range(len(s)):

ch = s[e]

chars[ch] += 1

max\_chars = max(chars[ch], max\_chars)

while e - b - max\_chars + 1 > k:

ch = s[b]

chars[ch] -= 1

b += 1

max\_len = max(max\_len, e - b + 1)

return max\_len

**Maximum Size Subarray Sum Equals k**

1. Use map to store leftmost sum,
2. Check if cur\_sum == k, or cur\_sum - k in map
3. Update max\_len

class Solution:

def maxSubArrayLen(self, nums, k):

max\_len = 0

cur\_sum = 0

m = {}

for i in range(len(nums)):

cur\_sum += nums[i]

if cur\_sum == k:

max\_len = max(max\_len, i+1)

if cur\_sum - k in m:

j = m[cur\_sum - k]

max\_len = max(max\_len, i - j)

if cur\_sum not in m:

m[cur\_sum] = i

return max\_len

**My Calendar II**

1. Store [s,e] in the array overlaps if it overlaps with easier interval
2. Store [s,e] if it’s not overlapped.
3. If [s,e] overlaps once, append [max(s,start), min(e, end)] overlapped interval

**Sum of Subarray Minimums**

class Solution:

def sumSubarrayMins(self, A: List[int]) -> int:

MOD = 10 \*\* 9 + 7

res = 0 # min arrays sum

dot = 0

st = [] #(min\_val, count) = run length encoding, count of min\_val in arrays

#consider arrrays with min\_vals: 0:i,1:i, 2:i and so on

for i,a in enumerate(A):

#array is from one [a]

count = 1

# pop old greater min\_val arrays, replace it by new min\_val - a

while st and st[-1][0] >= a:

x,c = st.pop()

# count new arrays min

count += c

#subtract old min sums

dot -= x\*c

#get count array with min a

st.append([a, count])

dot += a \* count

res += dot

return res % MOD

**Odd Even Jump**

1. Use monotonic jump

class Solution(object):

def oddEvenJumps(self, A):

N = len(A)

def next\_jumps(indexes):

# build monotonic stack

next\_jump = [None] \* N

stack = [] # invariant: stack is decreasing [4,3,2,2,1... <- top

for i in indexes:

while stack and i > stack[-1]: # i <= stack.top by keeping decreasing order

# if index is smaller, store it.

next\_jump[stack.pop()] = i

stack.append(i)

return next\_jump

# sorted A in increasing order using indexes

indexes = sorted(range(N), key=lambda i: A[i])

# there is jump where we jump from i in oddnext

oddnext = next\_jumps(indexes)

# sort in decreasin order

indexes.sort(key=lambda i: -A[i])

evennext = next\_jumps(indexes)

# for even and odd jumps: it true it can jump to i

odd = [False] \* N

even = [False] \* N

odd[N - 1] = even[N - 1] = True

# go from right to left

for i in range(N - 2, -1, -1):

# if we can jump from i, there even -> odd

if oddnext[i] is not None:

odd[i] = even[oddnext[i]]

# jump from odd to even

if evennext[i] is not None:

even[i] = odd[evennext[i]]

# count number of True(1)

return sum(odd)

**Next Greater Element I**

1. Use stack to hold there increasing order
2. Create map where store right item of a

class Solution:

def nextGreaterElement(self, nums1, nums2):

st = []

# right element of a

right = {a:-1 for a in nums1}

# go from right to left

# st holds items in increasing order, you can find right neighbour of a

for i in range(len(nums2) - 1, -1, -1):

# pop up until st.pop <= nums2[i]

while st and st[-1] <= nums2[i]:

st.pop()

if st:

right[nums2[i]] = st[-1]

st.append(nums2[i])

res = []

for a in nums1:

res.append(right[a])

return res

**Predict the Winner**

1. Choose leftmost or rightmost items
2. Player 1 score is max(a,b) where a is nums[s] - win(s+1, e, men), b is nums[e] - win(s, e - 1, men)
3. Subtract player2 scores
4. If win(0, len(nums) - 1, men) >=0 the player1 wins

class Solution:

def PredictTheWinner(self, nums):

def win(s, e, mem):

if s == e:

return nums[s]

if (s, e) in mem:

return mem[(s, e)]

# we subtract from nums[s] scores, it's related to other Player2

# pick either leftmost or rightmost item

a = nums[s] - win(s + 1, e, mem)

b = nums[e] - win(s, e - 1, mem)

best = max(a, b)

mem[(s,e)] = best

return mem[(s,e)]

mem = {}

return win(0, len(nums) - 1, mem) >= 0

**Group Shifted Strings**

1. Assume all strings belonging to on shift chain keep the same offset between chars, take it as key

class Solution:

def get\_group\_key(self, s):

key = []

for i in range(1, len(s)):

# calculate diff with previous char

diff = ord(s[i]) - ord(s[i - 1])

# diff is lesser, add 26

if diff < 0:

diff += 26

#append as string

key.append(str(diff))

return ''.join(key)

def groupStrings(self, strings):

if not strings:

return []

# diff sequence to string list

shifting\_seq = defaultdict(list)

# strings shifted for one string have the same difference between chars!

for string in strings:

k = self.get\_group\_key(string)

shifting\_seq[k].append(string)

# return map values

return shifting\_seq.values()

**Sentence Screen Fitting**

1. **Use s[pos% sz] to check if we left off inside word, if so, let shift to left until char is ‘ ‘**
2. **Update pos(last sentence position) by cols, but increase by 1 if we left off on ‘ ‘ otherwise shift to left**

class Solution:

def wordsTyping(self, sentence, rows, cols):

s = ' '.join(sentence) + ' '

# size of tied string

sz = len(s)

pos = 0 # where finish sentence in screen

for i in range(rows):

pos += cols

# check if it doesn't finish inside some word

if s[pos % sz] == ' ':

# calculate new position

pos += 1

continue

pos\_in\_s = (pos-1) % sz

while pos >= 0 and s[(pos-1) % sz] != ' ':

pos\_in\_s = (pos - 1) % sz

pos -= 1

return pos // sz

**Number of Islands II**

**from collections import defaultdict**

**class UnionFind:**

**def \_\_init\_\_(self):**

**self.parent = defaultdict(int)**

**self.rank = defaultdict(int)**

**self.comp = 0**

**def find(self, x):**

**if x not in self.parent:**

**self.parent[x] = x**

**self.rank[x] = 0**

**return x**

**if self.parent[x] != x:**

**self.parent[x] = self.find(self.parent[x])**

**return self.parent[x]**

**def union(self, x1, x2):**

**p1 = self.find(x1)**

**p2 = self.find(x2)**

**if p1 == p2:**

**return**

**if self.rank[p1] > self.rank[p2]:**

**self.parent[p2] = p1**

**elif self.rank[p2] > self.rank[p1]:**

**self.parent[p1] = p2**

**else:**

**self.parent[p1] = p2**

**self.rank[p2] += 1**

**self.comp -= 1**

**def has(self, x):**

**return x in self.parent**

**def add(self, x):**

**if x not in self.parent:**

**self.parent[x] = x**

**self.rank[x] = 0**

**self.comp += 1**

**class Solution:**

**def numIslands2(self, m: int, n: int, positions):**

**res = []**

**uf = UnionFind()**

**for x, y in positions:**

**uf.add(x \* n + y)**

**for dx,dy in [[0, 1],[0, -1], [1, 0], [-1, 0]]:**

**x1 = x + dx**

**y1 = y + dy**

**c = x1 \* n + y1**

**if 0 <= x1 < m and 0 <= y1 < n and uf.has(c):**

**uf.union(x \* n + y, c)**

**res.append(uf.comp)**

**return res**

**Exam Room**

1. Keep list of occupied student’s seats
2. Start from dist = students[0], student = 0

3. If d = (s - prev) // 2 > dist, let assign student = prep + d

4. Closest distance between 2 seats is (I - J) // 2

import bisect

class ExamRoom(object):

def \_\_init\_\_(self, N):

self.n = N

self.students = []

def seat(self):

if not self.students:

self.students.append(0)

return 0

dist = self.students[0]

# start from 0

student = 0

# go over existent students to find leftmost with max distance

for i, s in enumerate(self.students):

if i:

prev = self.students[i - 1]

d = (s - prev) // 2

if d > dist:

dist = d

student = prev + d

# if rightmost is free and may be max distance

if self.n - 1 - self.students[-1] > dist:

student = self.n - 1

bisect.insort(self.students, student)

return student

def leave(self, p):

self.students.remove(p)

IsSubsequence

1. Use 2 pointers, if s[i] == t[j], i++, then j++

class Solution:

def isSubsequence(self, s: str, t: str):

n = len(s)

m = len(t)

l = 0

r = 0

while l < n and r < m:

if s[l] == t[r]:

l += 1

r += 1

return l == n

**String Transforms Into Another String**

1. If str2 has 26 chars we can’t use other chars for intermediate transformation, so return False

class Solution:

def canConvert(self, str1: str, str2: str) -> bool:

if not str1 and not str2 or str1 == str2:

return True

if len(str1) != len(str2):

return False

# if str2 has 26 char we can't use free chars in intermidiate transformation

if len(set(str2)) == 26:

return False

m = {}

for ch1,ch2 in zip(str1,str2):

if ch1 not in m:

m[ch1] = ch2

elif m[ch1] != ch2:

return False

return True

**Edit Distance**

1. Can insert,replace, delete chars

class Solution:

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

n = len(word1)

m = len(word2)

if n \* m == 0:

return n + m

dp = [[0] \* (n + 1) for \_ in range(m + 1)]

# n chars are needed to remove

for i in range(n + 1):

dp[0][i] = i

# m chars are needed to remove

for i in range(m + 1):

dp[i][0] = i

for i in range(1, m + 1):

for j in range(1, n + 1):

# insert char to word2, add 1 to edit distance for dp[i][j]

left = dp[i][j - 1] + 1

# insert char to word1, so add 1 to dp[i][j]

down = dp[i - 1][j] + 1

# don't need to increase edit distance in dp[i][j]

left\_right = dp[i - 1][j - 1]

if word1[j - 1] != word2[i - 1]:

# if current chars differ, increase dp[i][j]

left\_right += 1

dp[i][j] = min(left, down, left\_right)

return dp[-1][-1]

**Validate Stack Sequences**

1. Push a from pushed to stack
2. If st[-1] == popped[j], st.pop() and j += 1

class Solution:

def validateStackSequences(self, pushed, popped):

st = []

j = 0

for x in pushed:

pushed.append(x)

while st and st[-1] == popped[j]:

st.pop()

j += 1

return j == len(popped)

**Binary String With Substrings Representing 1 To N**

**class Solution:**

**def queryString(self, S: str, N: int) -> bool:**

**for i in range(1, N+1):**

**subs = "{0:b}".format(i)**

**if S.find(subs) == -1:**

**return False**

**return True**

**Largest Values From Labels**

from collections import defaultdict

class Solution:

def largestValsFromLabels(self, values, labels, num\_wanted, use\_limit):

if not values or not labels:

return 0

index = [i for i in range(len(values))]

index.sort(key=lambda i: values[i], reverse=True)

res = 0

lab\_to\_count = defaultdict(int)

for i in index:

if num\_wanted == 0:

return res

l = labels[i]

v = values[i]

if l not in lab\_to\_count or lab\_to\_count[l] < use\_limit:

res += v

lab\_to\_count[l] += 1

num\_wanted -= 1

return res

**Remove Comments**

1. Process line by line and consider every chars

class Solution:

def removeComments(self, source):

res = []

in\_block = False

for line in source:

i = 0

if not in\_block:

new\_line = []

while i < len(line):

if line[i:i+2] == '/\*' and not in\_block:

in\_block = True

i += 1

elif line[i:i+2] == '\*/' and in\_block:

in\_block = False

i += 1

elif line[i:i+2] == '//' and not in\_block:

break

elif not in\_block:

new\_line.append(line[i])

i += 1

if new\_line and not in\_block:

res.append(''.join(new\_line))

return res

**Insert Interval**

**class Solution:**

**def insert(self, intervals, newInterval):**

**if not intervals:**

**return [newInterval]**

**if not newInterval:**

**return intervals**

**res = []**

**s = 0**

**new\_s,new\_e = newInterval**

**i = 0**

**# add intervals which start < newIntervals start**

**while i < len(intervals) and intervals[i][0] < new\_s:**

**res.append(intervals[i])**

**i += 1**

**# may merge added intervals with newIntervals**

**if not res or res[-1][1] < new\_s:**

**res.append(newInterval)**

**else:**

**res[-1][1] = max(new\_e, res[-1][1])**

**# add the rest of intervals if they overlap, let merge them**

**while i < len(intervals):**

**s,e = intervals[i]**

**if res[-1][1] >= s:**

**res[-1][1] = max(res[-1][1], e)**

**else:**

**res.append(intervals[i])**

**i+=1**

**return res**

**Filling Bookcase Shelves**

1. Try to place I book on shelf, height = dp[i-1] + books[i][0]

max\_height = books[i][1], cur\_width = books[i][0]

2. For every 0 … I - 1 try to place my book on the shelf with i- 1 book and recalculate then common bookcase height.

Height = min(height, dp[i-2] + max\_height) or min(height, max\_height) if I - 2 < 0

max\_height = max(max\_height, books[i-1][1])

While Cur\_width += books[j] <= shelf\_width

def minHeightShelves(self, books, shelf\_width):

if not books:

return 0

# first book at shelf

dp = [books[0][1]]

for i in range(1, len(books)):

cur\_width = books[i][0]

# height if place books[i] on new shelf

height = dp[i - 1] + books[i][1]

max\_height = books[i][1]

j = i - 1

# try to place 0 ... i -1 books on the same self where i book takes place

while j >= 0 and cur\_width + books[j][0] <= shelf\_width:

cur\_width += books[j][0]

max\_height = max(max\_height, books[j][1])

# height if place books[i] on the current shelf

# dp[j] points out shelf height of dp[i-2] and dp[j] + max\_height is height if we place i and i -1 books at the same shelf

if j - 1 >= 0:

height = min(height, dp[j - 1] + max\_height)

else:

height = min(height, max\_height)

j -= 1

dp.append(min(height, dp[i - 1] + books[i][1]))

return dp[-1]

**Sliding Puzzle**

1. Use BFS to store board and add it to set to avoid repetition

import itertools

class Solution(object):

def slidingPuzzle(self, board):

# convert 2 dim array to 1-dimension string

s = ''.join([str(x) for x in list(itertools.chain(\*board))])

target = '123450'

res = 0

seen = set()

# store the current board

seen.add(s)

q = [s]

while q:

# size of current all board variants

sz = len(q)

for \_ in range(sz):

v = q.pop(0)

#if current board is desired target, return res

if v == target:

return res

i = v.index('0')

# displacement of adjacent cells

# 3 is column size

for d in [1, -1, 3, -3]:

ni = i + d

# i and ni can't be neighbours

if ni < 0 or ni > 5 or i == 2 and ni == 3 or i == 3 and ni == 2:

continue

nv = list(v)

# swap 0 and new place

nv[i] = v[ni]

nv[ni] = '0'

ns = ''.join(nv)

if ns in seen:

continue

seen.add(ns)

q.append(ns)

res += 1

return -1

**Missing Element in Sorted Array**

**class Solution:**

**def missingElement(self, nums: List[int], k: int) -> int:**

**# return missing number of between idx and 0**

**missing = lambda idx: nums[idx] - nums[0] - idx**

**n = len(nums)**

**# if k > max missing[n - 1], return last + missing number**

**if k > missing(n - 1):**

**return nums[n-1] + k - missing(n-1)**

**left = 0**

**right = n - 1**

**# find missing[left] < k <= missing[right]**

**while left < right:**

**pivot = left + (right - left) // 2**

**if k > missing(pivot):**

**left = pivot + 1**

**else:**

**right = pivot**

**return nums[left - 1] + k - missing(left - 1)**

**Reorder Routes to Make All Paths Lead to the City Zero**

1. Use BFS and create adjacent list with bidirectional graph

from collections import defaultdict

class Solution:

def minReorder(self, n: int, connections):

new\_edges = 0

adj\_list = defaultdict(list)

edges = set()

for x,y in connections:

adj\_list[x].append(y)

adj\_list[y].append(x)

edges.add((x,y))

seen = set()

seen.add(0)

q = [0]

while q:

n = q.pop(0)

for v in adj\_list[n]:

if v in seen:

continue

if (n,v) in edges:

new\_edges += 1

q.append(v)

seen.add(v)

return new\_edges

**Split Array into Consecutive Subsequences**

1. Av is number frequency
2. Want is with number is expected for subsequences
3. If want[a] > 0, use av[a] -= 1, want[a]-=1, want[a+1] += 1
4. Otherwise av[a] > 0 and av[a+1] > 0 and av[a+2] > 0 then use them, want[a+3] += 1

from collections import defaultdict

class Solution:

def isPossible(self, nums):

av = defaultdict(int)

# count subsequence value

want = defaultdict(int)

for a in nums:

av[a] += 1

for i in range(len(nums)):

a = nums[i]

# no available values

if av[a] <= 0:

continue

# if some suqsequence needs to have a

if want[a] > 0:

av[a] -= 1

# mark that we've used a

want[a] -= 1

# request next value

want[a+1] += 1

# if it has minimal subsequence

elif av[a] > 0 and av[a+1] > 0 and av[a+2] > 0:

av[a] -= 1

av[a+1] -= 1

av[a+2] -= 1

# request next value

want[a+3] += 1

else:

return False

return True

**Open the Lock**

1. Use BFS
2. O(Nˆ2 \* A ˆN +D)

from collections import deque

class Solution(object):

def openLock(self, deadends, target):

def neighbors(node):

for i in range(4):

x = int(node[i])

for d in (-1, 1):

y = (x + d) % 10

yield node[:i] + str(y) + node[i+1:]

dead = set(deadends)

queue = deque([('0000', 0)])

seen = {'0000'}

while queue:

node, depth = queue.popleft()

if node == target:

return depth

if node in dead:

continue

for nei in neighbors(node):

if nei not in seen:

seen.add(nei)

queue.append((nei, depth+1))

return -1

**Minimum Cost to Hire K Workers**

1. **Use greedy approach: calculate quality[i]/wage[i] and sort in increasing order**
2. **Put in maxheap worker to get workers in decreasing quality.**
3. **If pool == K:**
4. **Res = min(res,sumq \* ratio)**
5. **Otherwise pop the maximal quality**

from heapq import heappop, heappush

class Solution:

def mincostToHireWorkers(self, quality, wage, K: int) -> float:

# consider with min coefficient

workers = sorted([[w/q,q,w] for q,w in zip(quality, wage)])

res = float('inf')

pool = []

sumq = 0

for r, q, w in workers:

# candidates with maximaml quality first

heappush(pool, -q)

sumq += q

if len(pool) > K:

# remove redundant

a = heappop(pool)

sumq += a # subtract a from sum

if len(pool) == K:

res = min(res, r \* sumq)

return res

**Bulls and Cows**

**class Solution:**

**def getHint(self, secret: str, guess: str) -> str:**

**h = defaultdict(int)**

**bulls = cows = 0**

**for idx, s in enumerate(secret):**

**#**

**g = guess[idx]**

**if s == g:**

**bulls += 1**

**else:**

**# bool expression is true, it's converted to int 1**

**cows += int(h[s] < 0) + int(h[g] > 0)**

**# if g is in secret, (h[g] > 0)**

**h[s] += 1**

**# if g is in secret and in guess, so it's cow**

**h[g] -= 1**

**return "{}A{}B".format(bulls, cows)**

**Evaluate Division**

**from collections import defaultdict**

**class Solution:**

**def calcEquation(self, equations, values, queries):**

**adj\_list = defaultdict(list)**

**for i in range(len(equations)):**

**a, b = equations[i]**

**# represent a / b as a -> b with weight values[i]**

**# b / a as b -> a with weight 1 / values[i]**

**adj\_list[a].append([values[i], b])**

**adj\_list[b].append([1 / values[i], a])**

**res = []**

**def find\_path(adj\_list, a, b, seen):**

**# it b == v, return k as answer**

**for k,v in adj\_list[a]:**

**if v == b:**

**return k**

**# if we considered it, go on**

**if v in seen:**

**continue**

**seen.add(v)**

**# go on considering path from v -> b with weight values[i] \* values[j]**

**r = k \* find\_path(adj\_list, v, b, seen)**

**# if r < 0.0 return -1**

**if r >= 0.0:**

**return r**

**return -1.0**

**for a, b in queries:**

**# find path from a to b**

**# use DFS, to find a path, if it can't add -1**

**if a not in adj\_list or b not in adj\_list:**

**res.append(-1.0)**

**continue**

**# use DFS to calculate an expression a /b**

**r = find\_path(adj\_list, a, b, set())**

**res.append(r)**

**return res**

**Android Unlock Patterns**

1. Use DFS

2. Encode non-adjacent number with the array skip

class Solution:

def numberOfPatterns(self, m, n):

def dfs(seen, skip, cur, rem):

if rem == 0:

return 1

if rem < 0:

return 0

seen.add(cur)

ret = 0

for i in range(1,10):

if i not in seen and (skip[i][cur] == 0 or skip[i][cur] in seen):

ret += dfs(seen, skip, i, rem - 1)

seen.discard(cur)

return ret

res = 0

skip = [[0] \* 10 for \_ in range(10)]

# encode intermidiate number

skip[1][3] = skip[3][1] = 2

skip[1][7] = skip[7][1] = 4

skip[7][9] = skip[9][7] = 8

skip[3][9] = skip[9][3] = 6

skip[1][9] = skip[9][1] = skip[2][8] = skip[8][2] = skip[3][7] = skip[7][3] = skip[4][6] = skip[6][4] = 5

seen = set()

for i in range(m, n+1):

res += (dfs(seen, skip, 1, i - 1) \* 4)

res += (dfs(seen, skip, 2, i - 1) \* 4)

res += dfs(seen, skip, 5, i - 1)

return res

**Stream of Characters**

1. Use Trie to append reversed words.
2. Use deque to append type letters
3. Go from left of deque and find substring in trie

class StreamChecker:

def \_\_init\_\_(self, words):

self.trie = {}

self.stream = deque([])

for word in set(words):

node = self.trie

for ch in word[::-1]:

if ch not in node:

node[ch] = {}

node = node[ch]

node['#'] = word

def query(self, letter: str) -> bool:

self.stream.appendleft(letter)

node = self.trie

for ch in self.stream:

if '#' in node:

return True

if ch not in node:

return False

node = node[ch]

return '#' in node

**Maximum Points You Can Obtain from Cards**

1. Use 2 pointers to calculate intermediate n - k subarray and return max of total- sum of (n-k) subsarray

class Solution:

def maxScore(self, cardPoints, k: int):

n = len(cardPoints)

dp = [[0] \* n for \_ in range(n)]

max\_score = 0

for l in range(max(n - k, 0), n + 1):

for i in range(n):

j = i + l - 1

if j >= n:

continue

if l == 2:

dp[i][j] = cardPoints[i] + cardPoints[j]

elif l > 2:

dp[i][j] = max(cardPoints[i] + dp[i + 1][j], cardPoints[j] + dp[i][j - 1])

max\_score = max(max\_score, dp[i][j])

return max\_score

**My Calendar I**

1. Use 2 intervals overlap if max(a0,b0) < min(a1,b1)

class MyCalendar:

def \_\_init\_\_(self):

self.booked = []

def book(self, start, end):

for s,e in self.booked:

if max(s, start) < min(e, end):

return False

self.booked.append([start, end])

return True

**Robot Room Cleaner**

class Solution(object):

def cleanRoom(self, robot):

def go\_back():

robot.turnRight()

robot.turnRight()

robot.move()

robot.turnRight()

robot.turnRight()

def backtrack(cell = (0, 0), d = 0):

visited.add(cell)

robot.clean()

# going clockwise : 0: 'up', 1: 'right', 2: 'down', 3: 'left'

for i in range(4):

new\_d = (d + i) % 4

new\_cell = (cell[0] + directions[new\_d][0], \

cell[1] + directions[new\_d][1])

if not new\_cell in visited and robot.move():

backtrack(new\_cell, new\_d)

go\_back()

# turn the robot following chosen direction : clockwise

robot.turnRight()

# going clockwise : 0: 'up', 1: 'right', 2: 'down', 3: 'left'

directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]

visited = set()

backtrack()

**Count Submatrices With All Ones**

1. Count height of every column row by row
2. If mat[i][j], let assign height[j] += 1 otherwise 0
3. Count sub matrices to corresponding height[j], let t = min(t, height[j])

class Solution:

def numSubmat(self, mat):

n = len(mat)

m = len(mat[0])

h = [0] \* m

res = 0

for i in range(n):

for j in range(m):

if mat[i][j] == 1:

h[j] += 1

else:

h[j] = 0

for l in range(m):

k = l

t = float('inf')

while k < m and h[k] != 0:

t = min(t, h[k])

res += t

k += 1

return res

**Serialize and Deserialize N-ary Tree**

class Codec:

def serialize(self, root: 'Node'):

if not root:

return None

buf = []

self.\_serialize(root, buf)

return ''.join(buf)

def \_serialize(self, root, buf):

if not root:

return

buf.append(chr(root.val + 48))

buf.append(chr(len(root.children) + 48))

for child in root.children:

self.\_serialize(child, buf)

def deserialize(self, data: str) -> 'Node':

if not data:

return None

pos = [0]

return self.\_deserialize(data, pos)

def \_deserialize(self, data, pos):

if len(data) == pos[0]:

return None

val = ord(data[pos[0]]) - 48

pos[0] += 1

n = ord(data[pos[0]]) - 48

node = Node(val, [])

for \_ in range(n):

pos[0] += 1

node.children.append(self.\_deserialize(data, pos))

return node

**Find in Mountain Array**

1. Find peak
2. Use binary search to find target in [0;peak] and [peak, n - 1]

class Solution:

def findInMountainArray(self, target: int, mountain\_arr: 'MountainArray') -> int:

n = mountain\_arr.length()

if n < 3:

return -1

peak = self.find\_peak(mountain\_arr, 0, n - 1)

if peak == -1:

return peak

left = self.find\_target(mountain\_arr, target, 0, peak, True)

if left != -1:

return left

return self.find\_target(mountain\_arr, target, peak, n - 1, False)

def find\_peak(self, arr, s, e):

while s < e:

m = s + (e - s)//2

a = arr.get(a)

if a < arr.get(m+1):

s = m + 1

else:

e = m

return e

def find\_target(self, arr, target, s, e, asc):

while s <= e:

m = s + (e - s)//2

a = arr.get(m)

if a == target:

return m

if asc:

if a < target:

s = m + 1

else:

e = m - 1

else:

if a < target:

e = m - 1

else:

s = m + 1

return -1

**Delete Nodes And Return Forest**

class Solution:

def delNodes(self, root: TreeNode, to\_delete):

if not root:

return []

def dfs(node, par, res, is\_left):

if not node:

return

if par and par.val in to\_delete and node.val not in to\_delete:

res.append(node)

dfs(node.left, node, res, True)

dfs(node.right, node, res, False)

if par and node.val in to\_delete:

if is\_left:

par.left = None

else:

par.right = None

res = []

if root.val not in to\_delete:

res.append(root)

dfs(root.left, root,res,True)

dfs(root.right, root,res,False)

return res

**Count Square Submatrices with All Ones**

1. Use dp[i][j] = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1, if matrix[i][j] == 1

Count += dp[i][j]

class Solution:

def countSquares(self, matrix):

if not matrix or not matrix[0]:

return 0

n = len(matrix)

m = len(matrix[0])

dp = [[0] \* (m + 1) for \_ in range(n + 1)]

count = 0

for i in range(1, n + 1):

for j in range(1, m + 1):

if matrix[i-1][j-1] == 1:

sz = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1]) + 1

count += sz

dp[i][j] = sz

return count

**Campus Bikes II**

1.Use DP with bit mask, go throug workers

class Solution:

def assignBikes(self, workers, bikes) -> int:

def assign\_bike(w, seen, mem):

if w == len(workers):

return 0

k = (w, seen)

if k in mem:

return mem[k]

p = 1

min\_sum = float('inf')

for i in range(len(bikes)):

if (p & seen) == 0:

x,y = bikes[i]

x1,y1 = workers[w]

dist = abs(x - x1) + abs(y - y1)

min\_sum = min(min\_sum, dist + assign\_bike(w + 1, seen | (1 << i), mem))

p = p << 1

mem[k] = min\_sum

return mem[k]

return assign\_bike(0, 0, {})

**Binary Tree Coloring Game**

1.Count upper subtreee node, left and right subtree to mark one of them

class Solution:

def btreeGameWinningMove(self, root: TreeNode, n: int, x: int) -> bool:

if not root or root.left is None and root.right is None:

return False

def find(node, x):

if not node:

return None

if node.val == x:

return node

l = find(node.left, x)

if l:

return l

return find(node.right, x)

def count(node):

if not node:

return 0

left = count(node.left)

right = count(node.right)

return left + right + 1

node = find(root, x)

# count x nodes

c1 = count(node)

#count upper subtree

p = n - c1

#count left subtree

l = count(node.left)

#count right subtree

r = count(node.right)

return p > l + r + 1 or l > p + r + 1 or r > p + l + 1

**Longest Repeating Character Replacement**

1. Use max char frequency and len of substring - freq > k, than increment l+=1

from collections import defaultdict

class Solution:

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

max\_len = 0

max\_chars = 0

chars = defaultdict(int)

l = 0

r = 0

while r < len(s):

ch = s[r]

chars[ch] += 1

#calculate max frequency

max\_chars = max(chars[ch], max\_chars)

r += 1

# if substring length is greater then len - max\_chars, that means we have the rest of chars requires # more tnan k operations

while r - l - max\_chars > k:

ch = s[l]

chars[ch] -= 1

l += 1

max\_len = max(max\_len, r - l)

return max\_len

**Cracking the Safe**

**Use DSF and Euler path to build de Bruini sequence.**

class Solution:

def crackSafe(self, n: int, k: int) -> str:

if n == 1 and k == 1:

return '0'

s = '0' \* (n - 1)

res = []

seen = set()

def dfs(cur, seen):

# for every edge with x

for x in range(k):

# go by edge

# new node

nei = cur + str(x)

# if we visit it, let's skip it.

if nei not in seen:

# go from the next vertex

seen.add(nei)

dfs(nei[1:], seen)

res.append(str(x))

dfs(s, seen)

return ''.join(res) + s

**Tiling a Rectangle with the Fewest Squares**

1.Use backtracking and 2 aproaches:

2 squares,

2 squares + 3 rectangles

class Solution:

def tilingRectangle(self, n, m):

n1 = max(n, m)

mem = [[-1] \* n1 for \_ in range(n1)]

return self.solve(min(n, m), max(n, m), mem)

def solve(self, n, m, mem):

if mem[n - 1][m - 1] != -1:

return mem[n - 1][m - 1]

if n == m:

return 1

if n == 0 or m == 0:

return 0

if n == 1:

return m

# find min square

nextn = min(n, m)

# find second square

nextm = max(n, m) - nextn

# first square is min(n,m) x min(n,m), let solve the rest

# solve for the rest rects

res = self.solve(min(nextm, nextn), max(nextm, nextn), mem) + 1

s = nextn - 1

while s > 0:

a = max(abs(m - s), abs(n - s))

b = min(abs(m - s), abs(n - s))

k = b

while k <= a and k <= n:

# max, min a-k,b-k

x = min(abs(a - k), abs(b - k))

y = max(abs(a - k), abs(b - k))

# solve for existant rectange

rect1 = self.solve(min(b, abs(m - k)), max(b, abs(m - k)), mem)

rect2 = self.solve(x, y, mem)

rect3 = self.solve(min(a, abs(n - k)), max(a, abs(n - k)), mem)

res = min(res, rect1 + rect2 + rect3 + 2)

k += 1

s -= 1

mem[n - 1][m - 1] = res

return mem[n - 1][m - 1]

**Minimum Swaps To Make Sequences Increasing**

1. Use DP: if A[i-1] < A[i] and B[i-1] and B[i] we can swap both i-1 and I columns, otherwise A[i-1] < B[i] and B[i-1] < A[i], we can swap either I -1 or I column

class Solution:

def minSwap(self, A, B):

s1 = 1

n1 = 0

for i in range(1, len(A)):

s2 = float('inf')

n2 = float('inf')

if A[i-1] < A[i] and B[i-1] < B[i]:

# could swap both A[i] and B[i]

n2 = min(n2, n1)

s2 = min(s2, s1+1)

if A[i-1] < B[i] and B[i-1] < A[i]:

# could swap either i-1 or i

n2 = min(n2, s1)

s2 = min(s2, n1+1)

n1 = n2

s1 = s2

return min(n1,s1)

**Move Zeroes**

1. Use slow and fast pointers:

Slow one points out last non zero items,

If fast pointers points out,let’s exchange items.

class Solution:

def moveZeroes(self, nums):

i = 0

for j in range(len(nums)):

if nums[j] != 0:

nums[i], nums[j] = nums[j], nums[i]

i += 1

return nums

**Binary Tree Vertical Order Traversal**

1. Use BFS: add node and level related to root.
2. Root is 0 level, left is -1,right is +1

class Solution:

def reverseVowels(self, s):

vowles = {'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'}

i = 0

chars = list(s)

j = len(chars) - 1

while i < j:

while i < j and chars[i] not in vowles:

i += 1

while i < j and chars[j] not in vowles:

j -= 1

chars[i], chars[j] = chars[j], chars[i]

i += 1

j -= 1

return ‘'.join(chars)

**Add to Array-Form of Integer**

1. Add K to end of array
2. A[i] = A[i] % 10, Carry = A[i] / 10
3. If I > 0: A[i-1] += carry
4. If (carry > 0)A = list(map(int, str(carry)) +A

**Matrix Block Sum**

1. Use prefix sum:

pr[i][j] = pr[i-1][j] + pr[i][j-1] - pr[i-1][j-1] + mat[i-1][j-1], I = 1, n + 1, j = 1,m + 1

Sum[i-1][j-1] = pr[r2][c2] - pr[r2][c1-1] - pr[r1-1][c2] + pr[r1-1][c1-1]

class Solution:

def matrixBlockSum(self, mat, K):

n = len(mat)

m = len(mat[0])

pr = [[0] \* (m+1) for \_ in range(n+1)]

for i in range(1, n+1):

for j in range(1, m+1):

pr[i][j] = mat[i-1][j-1] + pr[i-1][j] + pr[i][j-1] - pr[i-1][j-1]

res = [[0] \* m for \_ in range(n)]

for i in range(1, n + 1):

for j in range(1, m + 1):

r1 = max(1, i - K)

c1 = max(1, j - K)

r2 = min(n, i + K)

c2 = min(m, j + K)

res[i-1][j-1] = pr[r2][c2] - pr[r1-1][c2] - pr[r2][c1-1]+ pr[r1-1][c1-1]

return res

**Optimal Account Balancing**

1. Use backtracking
2. If item[start] is return dfs(start+1,lst)

from collections import defaultdict

class Solution:

def minTransfers(self, transactions):

balance = defaultdict(int)

# calculate balance

for s,e,m in transactions:

balance[s] -= m

balance[e] += m

# filter money differ from 0

lst = [v for k,v in balance.items() if v != 0]

def dfs(start, lst):

# if

if start == len(lst):

return 0

# if some item is 0,it needs to go the end to get min\_trans

cur = lst[start]

if cur == 0:

return dfs(start + 1, lst)

min\_trans = float('inf')

#try every start with every i, make all combination

for i in range(start + 1, len(lst)):

next = lst[i]

# if item signs are different

if cur \* next < 0:

lst[i] = next + cur

min\_trans = min(min\_trans, 1 + dfs(start+1, lst))

lst[i] = next

if cur + next == 0:

break

return min\_trans

return dfs(0, lst)

**Minimum Swaps To Make Sequences Increasing:**

1. Use DP:

If A[i-1] < A[i-1] and B[i-1] < B[i-1], we can move both i-1 and I columns only.

keep[i] = min(keep[i], keep[i-1])

swap[i] = min(swap[i], swap[i-1] + 1)

If A[i-1] < B[i] and B[i-1] < A[i], we can move either I - 1 or I columns

keep[i] = min(keep[i],swap[i-1])

swap[i[ = min(swap[i], kee[i-1] + 1)

class Solution:

def minSwap(self, A, B):

swap1 = 1

norm1 = 0

for i in range(1, len(A)):

swap2 = float('inf')

norm2 = float('inf')

if A[i - 1] < A[i] and B[i - 1] < B[i]:

# could swap both A[i] and B[i]

# don't swap on i - 1 and i

norm2 = min(norm2, norm1)

# swap on i - 1 and i

swap2 = min(swap2, swap1 + 1)

if A[i - 1] < B[i] and B[i - 1] < A[i]:

# could swap either i-1 or i

# don't swap on i but swap i - 1

norm2 = min(norm2, swap1)

# don't swap on i - 1 but swap on i

swap2 = min(swap2, norm1 + 1)

norm1 = norm2

swap1 = swap2

return min(norm1, swap1)

**Confusing Number II**

1. Use backtracking

class Solution:

def confusingNumberII(self, N):

nums = [0, 1, 6, 8, 9]

# use any number for not-confusing number

rotation = [0, 1, 100, 100, 100, 100, 9, 100, 8, 6]

count = [0]

def to\_num(arr):

s = 0

base = 1

for i in range(len(arr)-1,-1,-1):

s += base \* arr[i]

base \*= 10

return s

def is\_valid(a):

b = a

res = 0

while b > 0:

n = b % 10

res = res \* 10 + rotation[n]

b = b // 10

return a != res

def get\_confusing\_num(buf, count):

if buf:

a = to\_num(buf)

#check for 0 to avoid indefinite loop

if a == 0 or a > N:

return

if is\_valid(a):

count[0] += 1

if a \* 10 > N:

return

for i in range(len(nums)):

buf.append(nums[i])

get\_confusing\_num(buf, count)

buf.pop()

get\_confusing\_num([], count)

return count[0]

**Determine if Two Strings Are Close**

from collections import Counter

class Solution:

def closeStrings(self, word1, word2):

if len(word1) != len(word2):

return False

freq1 = Counter(word1)

lst1 = [a for k,a in freq1.items()]

freq2 = Counter(word2)

lst2 = [a for k,a in freq2.items()]

if set(freq1.keys()) != set(freq2.keys()):

return False

lst1.sort()

lst2.sort()

return lst1 == lst2

**Minimum Operations to Reduce X to Zero**

1.solve revert task to find max subsequence with sum(arr) - x

class Solution:

def minOperations(self, nums, x):

total = sum(nums)

if total == x:

return len(nums)

max\_len = -1

#find max subsequence equal to total - x

need\_sum = total - x

left = 0

cur\_sum = 0

right = 0

while right < len(nums):

cur\_sum += nums[right]

while cur\_sum >= need\_sum and left <= right:

if cur\_sum == need\_sum:

max\_len = max(max\_len, right - left + 1)

cur\_sum -= nums[left]

left += 1

right += 1

return len(nums) - max\_len if max\_len != -1 else -1

**Maximize Grid Happiness**

1. **Use dfs + dp to solve it.**
2. **Use p as counter to get column and row numbers.**

**class Solution:**

**def getMaxGridHappiness(self, m: int, n: int, introvertsCount, extrovertsCount):**

**# m is row number,n is column number**

**def cost(i, j, mask\_in, mask\_ex, d):**

**diff = 0**

**up = (1 << n - 1)**

**if j > 0 and (mask\_in & 1):**

**diff += (d - 30)**

**if i > 0 and (mask\_in & up):**

**diff += (d - 30)**

**if j > 0 and (mask\_ex & 1):**

**diff += (d + 20)**

**if i > 0 and (mask\_ex & up):**

**diff += (d + 20)**

**return diff**

**def dfs(dp, p, intr, extr, mask\_in, mask\_ex):**

**i = p // n**

**j = p % n**

**if i >= m:**

**return 0**

**if (p, intr, extr, mask\_in, mask\_ex) in dp:**

**return dp[(p, intr, extr, mask\_in, mask\_ex)]**

**new\_mask\_in = (mask\_in << 1) & 63**

**new\_mask\_ex = (mask\_ex << 1) & 63**

**res = dfs(dp, p + 1, intr, extr, new\_mask\_in, new\_mask\_ex)**

**if intr > 0:**

**d = 120 + cost(i, j, mask\_in, mask\_ex, -30)**

**res = max(res, d + dfs(dp, p + 1, intr - 1, extr, new\_mask\_in + 1, new\_mask\_ex))**

**if extr > 0:**

**d = 40 + cost(i, j, mask\_in, mask\_ex, 20)**

**res = max(res, d + dfs(dp, p + 1, intr, extr - 1, new\_mask\_in, new\_mask\_ex + 1))**

**dp[(p, intr, extr, mask\_in, mask\_ex)] = res**

**return res**

**dp = {}**

**return dfs(dp, 0, introvertsCount, extrovertsCount, 0, 0)**

**Minimum Moves to Make Array Complementary**

1. **Calculate moves for very sum from 2 to 2\*limit**

2. Define array to hold difference in move between sum[i] - sum[i-1]

1. 2 <= T < min(A, B) + 1, we need 2 operations to make both A, B smaller
2. min(A, B) + 1 <= T < A + B, we need 1 operation to make the larger one out of A and B smaller
3. T = A + B, we need 0 operation
4. A + B < T < max(A, B) + limit, we need 1 operation to make the smaller one out of A and B larger
5. max(A, B) + limit < T <= 2 \* limit, we need 2 operation to make both A, B larger

class Solution:

def minMoves(self, nums, limit):

n = len(nums)

dt = defaultdict(int) #dT[i] = T[i] - T[i-1]

for i in range(n//2):

a = nums[i]

b = nums[n - i - 1]

dt[2] += 2

dt[min(a, b) + 1] -= 1 # from min(a,b) to min(a,b)+1

dt[a + b] -= 1

dt[a + b + 1] += 1

dt[max(a, b) + limit + 1] += 1

res = math.inf

cur\_moves = 0

for s in range(2, 2\*limit + 1):

cur\_moves += dt[s]

res = min(res, cur\_moves)

return res

**Minimize Deviation in Array**

1. Use maxheap,

2. Increase all item, add it to maximal heap

3.pop max item, calculate maximal deviation,

4. Decrease item, if you can’t decrease item, then stop.

import math

from heapq import heappop, heappush,heapify

class Solution:

def minimumDeviation(self, nums):

n = len(nums)

min\_dev = math.inf

min\_val = math.inf

q = []

# increase all item till their maximal values

for i in range(n):

if nums[i] % 2 == 0:

q.append(-nums[i])

min\_val = min(min\_val, nums[i])

else:

q.append(-nums[i] \* 2)

min\_val = min(min\_val, nums[i] \* 2)

heapify(q)

#max queue

while q:

# take max value

a = -heappop(q)

# calculate minimal deviation

min\_dev = min(min\_dev, a - min\_val)

if a % 2 == 0:

# a is even, we can decrease it

min\_val = min(min\_val, a // 2)

heappush(q, -a // 2)

else:

# if max is odd, we can't decrease it, so exit

break

return min\_dev

**Find the Most Competitive Subsequence**

1. Add item to queue, if queue[-1] > a ,then pop(item) and addCount > 0, addCount is how many char we can leave.

class Solution:

def mostCompetitive(self, nums, k):

q = []

addCount = len(nums) - k

for a in nums:

while q and q[-1] > a and addCount > 0:

q.pop()

addCount -= 1

q.append(a)

# q may contain more then k items

return q[:k]

**Minimum Incompatibility**

1. Use backtract, memoize set to avoid use it again in backtracking tree.
2. Calculate impact as difference, use global min, and calculate acc if acc + impact < self.min, then add item and call backtrack

import math

class Solution:

def minimumIncompatibility(self, nums, k):

n = len(nums)

m = []

for i in range(k):

m.append(set())

self.min = math.inf

def calc\_impact(s, num):

if len(s) == 0:

return 0

if len(s) == 1:

return abs(num - list(s)[0])

min\_val = min(s)

max\_val = max(s)

if num < min\_val:

return min\_val - num

if num > max\_val:

return num - max\_val

return 0

bucket\_size = n // k

def calc\_incompatibility(m):

total = 0

for s in m:

min\_val = min(s)

max\_val = max(s)

total += (max\_val - min\_val)

return total

def backtrack(cur, m, acc):

if cur >= n:

self.min = min(acc, self.min)

return

visited = set()

for s in m:

if nums[cur] in s or len(s) == bucket\_size or tuple(s) in visited:

continue

impact = calc\_impact(s, nums[cur])

if acc + impact < self.min:

s.add(nums[cur])

backtrack(cur + 1, m, acc + impact)

s.discard(nums[cur])

visited.add(tuple(s))

backtrack(0, m, 0)

return self.min if self.min != math.inf else -1

**Concatenation of Consecutive Binary Numbers**

1. Use x & (x-1) == 0 if x is power of 2

class Solution:

def concatenatedBinary(self, n):

mod = 10 \*\* 9 + 7

res = 0

len = 0

for i in range(1, n + 1):

if i & (i - 1) == 0:

len += 1

res = (res << len) + i

res = res % mod

return res

**Partitioning Into Minimum Number Of Deci-Binary Numbers**

1. Use fact that maximum digits is how many number needs to add up to n

class Solution:

def minPartitions(self, n: str):

return int(max(n))

**Stone Game VII**

1.There is minimal problem, we can take either left or right and need maximise difference by subtracting from score - the difference

2. dp[i][j] = max(pref[j+1] - pref[i+1] - dp[i+1][j],

pref[j] - pref[i] - dp[i][j-1])

class Solution:

def stoneGameVII(self, stones):

if not stones:

return 0

pref\_sum = [0] \* (len(stones) + 1)

for i in range(len(stones)):

pref\_sum[i + 1] = pref\_sum[i] + stones[i]

n = len(stones)

dp = [[0] \* n for \_ in range(n)]

for l in range(2, n + 1):

for i in range(n):

j = i + l - 1

if j >= n:

break

sc1 = pref\_sum[j+1] - pref\_sum[i+1]

sc2 = pref\_sum[j] - pref\_sum[i]

dp[i][j] = max(sc1 - dp[i+1][j], sc2 - dp[i][j-1])

return dp[0][n-1]

**Maximum Height by Stacking Cuboids**

1. Use dp and LIS to solve it
2. Sort it by widht,length,heigth and sort final carry and preppend with [[0,0,0]]

class Solution:

def maxHeight(self, cuboids):

A = [sorted(s) for s in cuboids]

A = [[0,0,0]] + sorted(A)

dp = [0] \* len(A)

for i in range(1, len(A)):

for j in range(i):

if all(A[j][k] <= A[i][k] for k in range(3)):

dp[i] = max(dp[i], dp[j] + A[i][2])

return max(dp)

**Encode String with Shortest Length**

1. Use dp to memoize how to encode string
2. Have 3 choise: don’t encode string, having prefix[:i] and suffix[i:] and count prefix in suffix, try to encode prefix and suffix and prefix + suffix

class Solution:

def encode(self, s: str) -> str:

memo = {}

return self.\_get\_encoded\_str(s, memo)

def \_get\_encoded\_str(self, s, memo):

if s == "" or len(s) < 5:

return s

if s in memo:

return memo[s]

chosen\_str = s

min\_len = len(s)

for i in range(1, len(s) // 2 + 1):

encoded\_str = s

prefix = s[:i]

suffix = s[i:]

#count consecutive prefix in suffix

prefix\_count = self.\_get\_prefix\_count(prefix, suffix)

#encode prefix

encoded\_prefix = self.\_get\_encoded\_str(prefix, memo)

#encode suffix

encoded\_suffix = self.\_get\_encoded\_str(suffix, memo)

if prefix\_count > 0:

k = prefix\_count \* len(prefix)

#encode remaining part

encoded\_suffix\_from\_k = self.\_get\_encoded\_str(suffix[k:], memo)

#take prefix as 1 +

encoded\_str = str(prefix\_count + 1) + "[" + encoded\_prefix + "]" + encoded\_suffix\_from\_k

#

encoded\_part\_str = encoded\_prefix + encoded\_suffix

if len(encoded\_str) > len(encoded\_part\_str):

encoded\_str = encoded\_part\_str

if min\_len > len(encoded\_str):

chosen\_str = encoded\_str

min\_len = len(encoded\_str)

memo[s] = chosen\_str

return chosen\_str

"""

Helper method finds the count of prefix in suffix.

"""

def \_get\_prefix\_count(self, prefix, suffix):

count = 0

i = 0

while i < len(suffix):

j = suffix.find(prefix, i)

if i != j:

break

count += 1

i = j + len(prefix)

return count

**Jump Game VI**1

1. Use dequeue to store decreasing order, pop left the item from dequeue if there index < I - k, take leftfomst item + nums to get dp[i].

2. Evice rightmost items that <= dp[i]

class Solution:

def maxResult(self, nums, k):

n = len(nums)

res = nums[0]

dq = [[0,nums[0]]]

for i in range(1, n):

while dq and dq[0][0] < (i - k):

dq.pop(0)

res = nums[i] + dq[0][1]

while dq and dq[-1][1] <= res:

dq.pop()

dq.append([i,res])

return res

**Checking Existence of Edge Length Limited Paths**

1. Use DSU to connect nodes if it exists path.
2. Sort edgeList and quiries by distance

3. For each query to connect all nodes if their distance is less than limit

class Dsu:

def \_\_init\_\_(self, n):

self.p = [i for i in range(n)]

self.rank = [1] \* n

def find(self, x):

if self.p[x] != x:

self.p[x] = self.find(self.p[x])

return self.p[x]

def union(self, x1, x2):

p1 = self.find(x1)

p2 = self.find(x2)

if p1 == p2:

return

if self.rank[p1] > self.rank[p2]:

self.p[p2] = p1

elif self.rank[p2] > self.rank[p1]:

self.p[p1] = p2

else:

self.p[p1] = p2

self.rank[p2] += 1

class Solution:

def distanceLimitedPathsExist(self, n: int, edgeList, queries):

queries\_by\_dist = sorted([[x[0],x[1],x[2],i] for i,x in enumerate(queries)],key=lambda x:x[2])

edge\_by\_dist = sorted(edgeList, key=lambda x:x[2])

dsu = Dsu(n)

res = [False] \* len(queries)

start = 0

for s,e,limit,i in queries\_by\_dist:

while start < len(edge\_by\_dist):

p,q,dist = edge\_by\_dist[start]

if dist >= limit:

break

dsu.union(p, q)

start += 1

res[i] = dsu.find(s) == dsu.find(e)

return res

**Sliding Window Maximum**

1. Keep duque of decreasing items

class Solution:

def maxSlidingWindow(self, nums, k):

n = len(nums)

dq = []

res = []

for i in range(n):

while dq and dq[0][0] <= i - k:

dq.pop(0)

while dq and dq[-1][1] <= nums[i]:

dq.pop()

dq.append([i, nums[i]])

if i + 1 >= k:

res.append(dq[0][1])

return res

**Maximum Number of Eaten Apples**

1. Use minimum heap by appending expire days
2. Remove items if expiry day <= current\_day

from heapq import heappop, heappush, heapify

class Solution:

def eatenApples(self, apples, days):

res = 0

q = []

i = 0

n = len(apples)

#until having not empty days or i < n

while q or i < n:

#add to have most rotten apples

if i < n:

heappush(q, [i + days[i], i])

# remove rotten or empty apples

while q and (apples[q[0][1]] == 0 or q[0][0] <= i):

heappop(q)

if q:

#eat apple

res += 1

j = q[0][1]

apples[j] -= 1

i += 1

return res

**Where Will the Ball Fall**

1. If grid[r][c] == -1 and if c == 0 or grid[r][c-1] == 1:

Break otherwise c-= 1

2. If grid[r][c] == 1 and if c == m -1 or grid[r][c+1] == -1: break otherwise c += 1

3. If r == n answer[i] = c

class Solution:

def findBall(self, grid):

if not grid or len(grid[0]) <= 1:

return [-1] \* len(grid)

# row number

n = len(grid)

# column number

m = len(grid[0])

answer = [-1] \* m

# 1 from left to right

# -1 from right to left

for i in range(m):

c = i

r = 0

while r < n:

if grid[r][c] == -1:

if c == 0 or (c-1) >= 0 and grid[r][c-1] == 1:

break

c -= 1

else:

if c == m - 1 or c+1< m and grid[r][c+1] == -1:

break

c+=1

r += 1

if r == n:

answer[i] = c

return answer

**Maximum XOR With an Element From Array**

1. Sort nuts and queiries
2. Foreach for sorted quries:

Add all sorted nuts to trie

3 go by Trie to find number with inverted bits than x

If 1 -d exists, let do result or 1

lass Trie:

def \_\_init\_\_(self):

self.bin = {}

def add(self, a):

p = self.bin

for i in range(31, -1, -1):

d = (a >> i) & 1

if d not in p:

p[d] = {}

p = p[d]

def find(self, a):

if not self.bin:

return -1

p = self.bin

#get xored result

res = 0

for i in range(31, -1, -1):

d = (a >> i) & 1

if (1 - d) in p:

# this is 1 because 1-d is opposite bit in x

res |= (1 << i)

p = p[1 - d]

else:

p = p[d]

return res

class Solution:

def maximizeXor(self, nums, queries):

sorted\_nums = sorted(nums)

#sort nums

sorted\_indices = [i for i in range(len(queries))]

#sort by second m

sorted\_indices = sorted(sorted\_indices, key=lambda i: queries[i][1])

last = 0

trie = Trie()

res = [-1] \* len(queries)

for i in sorted\_indices:

x,m = queries[i]

#add number until sorted nums is <= m

while last < len(sorted\_nums) and sorted\_nums[last] <= m:

trie.add(sorted\_nums[last])

last += 1

res[i] = trie.find(x)

return res

**Maximum XOR of Two Numbers in an Array**

1. Use Trie to store number from L-1 to 0

class Trie:

def \_\_init\_\_(self):

self.bin = {}

def add(self, a):

p = self.bin

for i in range(31,-1,-1):

d = (a >> i) & 1

if d not in p:

p[d] = {}

p = p[d]

def find(self, a):

if not self.bin:

return -1

p = self.bin

res = 0

for i in range(31, -1, -1):

d = (a >> i) & 1

if (1 - d) in p:

res |= (1 << i)

p = p[1 - d]

else:

p = p[d]

return res

class Solution:

def findMaximumXOR(self, nums):

max\_xor = 0

trie = Trie()

trie.add(nums[-1])

for i in range(len(nums)-1,-1,-1):

xor = trie.find(nums[i])

max\_xor = max(xor, max\_xor)

trie.add(nums[i])

return max\_xor

**Minimize Hamming Distance After Swap Operations**

1. **Use Union-Find to store components with items which may be swapped**
2. Go through all source and make map[component] = items.
3. For items count source[i] and -target[i],to update dist by item number > 0

from collections import defaultdict

class Dsu:

def \_\_init\_\_(self, n):

self.p = [i for i in range(n)]

self.rank = [0] \* n

def find(self, x):

if self.p[x] != x:

self.p[x] = self.find(self.p[x])

return self.p[x]

def union(self, x1, x2):

p1 = self.find(x1)

p2 = self.find(x2)

if p2 == p1:

return

if self.rank[p1] > self.rank[p2]:

self.p[p2] = p1

elif self.rank[p1] < self.rank[p2]:

self.p[p1] = p2

else:

self.p[p1] = p2

self.rank[p2] += 1

class Solution:

def minimumHammingDistance(self, source, target, allowedSwaps):

dsu = Dsu(len(source))

for i, j in allowedSwaps:

dsu.union(i, j)

dist = 0

par\_to\_members = defaultdict(list)

for i in range(len(source)):

p = dsu.find(i)

par\_to\_members[p].append(i)

freq = defaultdict(int)

for \_,lst in par\_to\_members.items():

freq.clear()

for i in lst:

freq[source[i]] += 1

freq[target[i]] -= 1

dist += sum([v for \_, v in freq.items() if v > 0])

return dist

**Smallest String With Swaps**

1. Use DSU to store items from pairs
2. Make m[components] = items and sort them.
3. Go through s and get first char from m[component]

from collections import defaultdict

class Dsu:

def \_\_init\_\_(self, n):

self.p = [i for i in range(n)]

self.rank = [0] \* n

def find(self, x):

if self.p[x] != x:

self.p[x] = self.find(self.p[x])

return self.p[x]

def union(self, x1, x2):

p1 = self.find(x1)

p2 = self.find(x2)

if p2 == p1:

return

if self.rank[p1] > self.rank[p2]:

self.p[p2] = p1

elif self.rank[p1] < self.rank[p2]:

self.p[p1] = p2

else:

self.p[p1] = p2

self.rank[p2] += 1

class Solution:

def smallestStringWithSwaps(self, s: str, pairs):

dsu = Dsu(len(pairs))

for i,j in pairs:

dsu.union(i, j)

#component to its items

m = defaultdict(list)

for i in range(len(str)):

p = dsu.find(i)

m[p].append(i)

#sort all items inside component

for \_,lst in m:

lst.sort(key=lambda i:s[i])

min\_s = []

#merge item one by one

for i in range(len(s)):

p = dsu.find(i)

lst = m[p]

#take minimal item

min\_s.append(lst.pop(0))

return ''.join(min\_s)

**Find Minimum Time to Finish All Jobs**

1. Use Backtrack
2. To cur branch:

- don’t consider if workload[i] + jobs[j] >= max\_min

- don’t condor if workload[i] has been seen

from math import inf

class Solution:

def minimumTimeRequired(self, jobs, k):

max\_min\_sum = inf

jobs.sort(reverse=True)

workload = [0] \* k

def dfs(cur):

nonlocal max\_min\_sum

if cur == len(jobs):

max\_min\_sum = min(max\_min\_sum, max(workload))

return

seen = set()

for j in range(k):

if workload[j] in seen or workload[j] + jobs[cur] >= max\_min\_sum:

continue

#prevent to consider another same wokrloads

seen.add(workload[j])

workload[j] += jobs[cur]

dfs(cur + 1)

#seen.discard(workload[j])

workload[j] -= jobs[cur]

dfs(0)

return max\_min\_sum

**Partition to K Equal Sum Subsets**

1. Sort nuts in reverse order
2. Find sum(nums) // k
3. Allocate subset array for k
4. Cut branches:

* if subset[i] was seen
* If subset[i] + nums[j] > sum(nums) // k

from collections import defaultdict

class Solution:

def canPartitionKSubsets(self, nums, k):

arr\_sum = sum(nums)

need\_sum = arr\_sum // k

subset = [0] \* k

def dfs(cur):

if cur == len(nums):

return True

m = defaultdict(int)

for i in range(k):

if subset[i] in m:

if m[subset[i]]:

return True

else:

continue

if subset[i] + nums[cur] > need\_sum:

continue

subset[i] += nums[cur]

res = dfs(cur + 1)

subset[i] -= nums[cur]

m[subset[i]] = res

if res:

return True

return False

return dfs(0)

**Matchsticks to Square**

1. Sort nuts
2. Solve as above task for 4 subsets

class Solution:

def makesquare(self, nums):

side = [0] \* 4

need\_len = sum(nums) // 4

def dfs(cur):

if cur == len(nums):

return True

seen = set()

for i in range(4):

if side[i] in seen or side[i] + nums[cur] > need\_len:

continue

#if we have another side with same lenght

seen.add(side[i])

#if all sides are the same

side[i] += nums[cur]

if dfs(cur + 1):

return True

# backtract length

side[i] -= nums[cur]

return False

return dfs(0)

**Consecutive Numbers Sum**

1. Use N = x\*k + 1 + 2 + .. k = x\* k + k\*(k+1)/2
2. U <= sqrt(2\*N + 0.25) - 0.5)
3. Try k from 1 to U to hold the (N - k\*(k+1)/2)% k == 0 and increment count

class Solution:

def consecutiveNumbersSum(self, N):

upper\_limit = ceil((2 \* N + 0.25) \*\* 0.5 - 0.5) + 1

count = 0

for k in range(1, upper\_limit):

if (N - k\*(k+1)//2) % k == 0:

count += 1

return count

**Minimum Number of People to Teach**

from collections import defaultdict

class Solution:

def minimumTeachings(self, n: int, languages, friendships):

languages = [set(l) for l in languages]

teach = set()

#find users that don't have common language

for u, v in friendships:

if languages[u - 1] & languages[v - 1]:

continue

teach.add(u - 1)

teach.add(v - 1)

#cont number of language users speak

know\_lang = defaultdict(int)

for u in teach:

for lang in languages[u]:

know\_lang[lang] += 1

#number users don't have common language - most usable language = number users is needed to teach

return 0 if not know\_lang else len(teach) - max(know\_lang.values())

**Decode XORed Permutation**

1. Use xor = 1 ^ 2.. ^n
2. prim[-1] = xor ^encoded[0] ^encoded[2] ^ encoded[4] …
3. From lend(encoded)-1 => prim[i] = prim[ i+ 1] ^ encoded[i]

class Solution:

def decode(self, encoded):

n = len(encoded) + 1

xor = 0

#xor from 1 to n

for i in range(1, n + 1):

xor = xor ^ i

perm = [0] \* n

perm[-1] = xor

# calculate last perm using odd encoded[0],encoded[2] and so on.

for i in range(0 ,len(encoded), 2):

perm[-1] ^= encoded[i]

# calculate the rest of perm

for i in range(len(encoded) - 1, -1, -1):

perm[i] = perm[i + 1] ^ encoded[i]

return perm

**Count Ways to Make Array With Product**

1. Use stars and bars to count subsets number which product is n
2. Pre-calucate min smallest factor for I < 10001
3. Pre-calculat choose(n,k) = (choose(n-1,k-1) + choose(n-1,k)) % 10ˆ9 +7
4. Find factor frequencies for k
5. For every frequencies it find subsets number with choose[n+f-1][f], and multiply the number for each frequencies:

from collections import defaultdict

class Solution:

def waysToFillArray(self, queries):

mod = 10 \*\* 9 + 7

#min factor number

sfn = [0] \* 10002

for i in range(1, len(sfn)):

# for prime i is factor number is i

sfn[i] = i

i = 2

# till i \* i

while (i \* i) <= 10001:

# if i is prime:

if sfn[i] == i:

# start from i ^2

j = i \* i

while j <= 10001:

if sfn[j] == j:

# set up its min factor is i

sfn[j] = i

j += i

i += 1

# pre-calculate n choose k

choose = [[0] \* 14 for \_ in range(10033)]

# i == 1 and 0 bukets is 1

for i in range(len(choose)):

choose[i][0] = 1

for i in range(1, len(choose)):

for j in range(1, len(choose[0])):

# (n,k) = (n-1,k-1) + (n-1, k)

choose[i][j] = (choose[i-1][j-1] + choose[i-1][j]) % mod

# find factor frequenceis for [2,2,1,3,3] it returns [2,1,3]

def get\_factor\_freq(x):

fact\_to\_freq = defaultdict(int)

while x > 1:

fact\_to\_freq[sfn[x]] += 1

# divie by factor

x = x // sfn[x]

return [f for f in fact\_to\_freq.values()]

res = []

for n,k in queries:

# get factor frequencies

frq = get\_factor\_freq(k)

cur = 1

# every l freqiency make h1 subsets and to get total number we multiply number subsets.

# use stars and bars method to get number subsets for f factors:

# f is non-distingushible number to needs to distribute in n buckets.

# (f + n - 1) choose n

for f in frq:

cur = (cur \* choose[n + f - 1][f]) % mod

res.append(cur)

return res

**Coin Change**

import math

class Solution:

def coinChange(self, coins, amount):

m = [[0] \* (amount + 1) for \_ in range(len(coins) + 1)]

for i in range(1, amount + 1):

# if there is no coins, then there is infinity ways to make i

m[0][i] = math.inf

# go by row

for c in range(1, len(coins) + 1):

# for every amount

for r in range(1, amount + 1):

# coin's denomination is r

if coins[c - 1] == r:

m[c][r] = 1

# if coin's denomination is greater r, than take ways from previous coin

elif coins[c - 1] > r:

m[c][r] = m[c - 1][r]

else:

# min: 1) without this coin,2) with this coin

m[c][r] = min(m[c - 1][r], 1 + m[c][r - coins[c - 1]])

return m[-1][-1] if m[-1][-1] != math.inf else -1

**Sort Colors**

class Solution:

def sortColors(self, nums) -> None:

#three ways partitioning

lo = 0

mid = 0 #first pointer

hi = len(nums) - 1 #second pointer

while mid <= hi:

if nums[mid] == 0:#red

#swap nums[lo] and nums[mid]

nums[lo],nums[mid] = nums[mid],nums[lo]

lo += 1

mid += 1

elif nums[mid] == 2:#blue

#swap nums[mid] and nums[hi]

nums[mid],nums[hi] = nums[hi],nums[mid]

hi -= 1

else:

mid += 1 # white

**Largest Submatrix With Rearrangements**

1. Calculate height of row: if matrix[i][j] == 0 => height[j] = 0 otherwise height[j] += 1
2. Sorted in increasing order keeping the original array intact
3. Go from left to right and calculate area: heigh[j] \* (m - j)

lass Solution:

def largestSubmatrix(self, matrix):

n = len(matrix)

m = len(matrix[0])

height = [0] \* m

max\_area = 0

for i in range(n):

# calculate pillar heights

for j in range(m):

if matrix[i][j] == 0:

height[j] = 0

else:

height[j] += 1

#sort in increasing order but keeping orignal array

sorted\_height = sorted(height)

# go from left to right: rightmost items are higher then leftmost

for j in range(m):

# height \* number of rightmost items.

max\_area = max(max\_area, sorted\_height[j] \* (m - j))

return max\_area

**Largest Rectangle in Histogram**

1. Try to find for every heights[i] : heights[L] < heights[i] >= heights[R], then we may calculate area:
2. height[i] \* (R - L - 1)
3. Let’s use stack for this just push heights in increasing order. If we met the item < st[top], then there is R of stack top and L is st[top - 1]

4. For rightmost R is len(heights), for leftmost L is -1 so add to stack -1 as initial value

class Solution:

def largestRectangleArea(self, heights):

max\_area = 0

# consider heights[L] ... heights[j] ... heights[R]

#where where we check if heights[i] is R for stack top element

#put L is -1

st = [-1]

#last R is len(heights)

for i in range(len(heights)):

#push items in increasing order until we get item is less stack top and it will be R for stack top

R = i

while st[-1] != - 1 and heights[st[-1]] >= heights[i]:

t = st.pop()

L = st[-1]

# for t: L is stack top and R is current heights[i]

max\_area = max(max\_area, heights[t] \* (R - L - 1))

st.append(i)

# R is len(heights)

R = len(heights)

while st[-1] != -1:

t = st.pop()

L = st[-1]

# for heights[t]: L is stack top and R is len(heights)

max\_area = max(max\_area, heights[t] \* (R - L - 1))

return max\_area

**Maximal Rectangle**

1. Use the same approach to calculate maximal histogram area:

- push items until they are in increasing order

- if we meet the item < stack top, let assign it as R for items in stack and pop stack to calculate their areas

class Solution:

def maximalRectangle(self, matrix):

if not matrix or not matrix[0]:

return 0

n = len(matrix)

m = len(matrix[0])

heights = [0] \* m

max\_rect = 0

for i in range(n):

for j in range(m):

if matrix[i][j] == '1':

heights[j] += 1

else:

heights[j] = 0

#calculate the histogram area:

# consider: heights[L] < heights[i] >= heights[R]

# for leftmost items -1 performs as L, m does as R

st = [-1]

for j in range(m):

#push itme in increasing order until we find the heights[R] < stack top.

while st[-1] != -1 and heights[st[-1]] >= heights[j]:

# l is the item we calculate the area for.

l = st.pop()

#st[-1] is L, R is current j

max\_rect = max(max\_rect, heights[l] \* (j - st[-1] - 1))

st.append(j)

#it's left to consider m as R for rest of itmes in stack

while st[-1] != -1:

l = st.pop()

#L is stack[top-1]

max\_rect = max(max\_rect, heights[l]\*(m - st[-1] - 1))

return max\_rect

**Sliding Window Maximum**

1. Use dequeue to store maximal index, value
2. If dq[0] <= I - k, let’s pop that items
3. If dq[0][1] <= nums[i],let’s pop that items
4. Push current index, value
5. If i+1 >= k, let add to rest first dq[0][1]

class Solution:

def maxSlidingWindow(self, nums, k):

n = len(nums)

dq = []

res = []

for i in range(n):

# pop items aren't in k-window now.

while dq and dq[0][0] <= i - k:

dq.pop(0)

# pop items which are less or equal current

while dq and dq[-1][1] <= nums[i]:

dq.pop()

#push current item: index and value

dq.append([i, nums[i]])

#if we slide to the right on more than k, let add dq[0][1] to res

if i + 1 >= k:

res.append(dq[0][1])

return res

**Maximum Subarray**

1. Use greedy approach
2. Choose maximum nums[i],max\_so\_far+nums[i]
3. Choose global maximum max\_so\_far and max\_sum

class Solution:

def maxSubArray(self, nums):

if not nums:

return []

#local maximum

max\_so\_far = 0

#global maximum

max\_sum = float('-inf')

for i in range(len(nums)):

# choice new maximum: current item, max\_so\_far + current

max\_so\_far = max(nums[i], max\_so\_far + nums[i])

#calculate global sum

max\_sum = max(max\_so\_far, max\_sum)

return max\_sum

**Maximum Absolute Sum of Any Subarray**

1. Find max sum for positive items
2. Find min sum for negative items
3. Take max(max\_sum, max\_pos,abs(max\_neg))

from math import inf

class Solution:

def maxAbsoluteSum(self, nums):

cur\_sum1 = 0

cur\_sum2 = 0

max\_sum = -inf

#find max\_sum for positve items

# find min\_sum for negative items

for i in range(len(nums)):

cur\_sum1 = max(nums[i], cur\_sum1 + nums[i])

cur\_sum2 = min(nums[i], cur\_sum2 + nums[i])

max\_sum = max(max\_sum, cur\_sum1, abs(cur\_sum2))

return max\_sum

**Longest Turbulent Subarray**

1. Use Kadehni’s algorithms

class Solution:

def maxTurbulenceSize(self, arr):

if len(arr) <= 1:

return len(arr)

max\_len = 0

last\_num = -1

last\_rel = -1 #0 is less, 1 is greater

cur\_len = 0

for i in range(len(arr)):

if cur\_len == -1:

cur\_len = 1

last\_num = arr[i]

else:

if last\_num == arr[i]:

last\_rel = -1

cur\_len = 1

last\_num = arr[i]

elif last\_rel == 1 and last\_num > arr[i] or last\_rel == 0 and last\_num < arr[i]:

cur\_len = 2

last\_rel = 1 if last\_num > arr[i] else 0

last\_num = arr[i]

else:

cur\_len += 1

last\_rel = 1 if last\_num > arr[i] else 0

last\_num = arr[i]

max\_len = max(max\_len, cur\_len)

return max\_len

**Maximum Subarray Sum After One Operation**

import math

class Solution:

def maxSumAfterOperation(self, nums):

#max subarray length without double item

cur\_sum1 = 0

# max subarray length with double item

cur\_sum2 = 0

#global max subarray length

max\_sum = -math.inf

for i in range(len(nums)):

# max len either current item, or previous subarray + current item

new\_cur\_sum1 = max(nums[i], cur\_sum1 + nums[i])

#max len either current item ^2 or subarray with previous double item or prevoius subarray(without item doubling) + double current item

new\_cur\_sum2 = max(nums[i] \* nums[i], cur\_sum2 + nums[i],cur\_sum1 + nums[i] \* nums[i])

# pick up the max subarray length

max\_sum = max(max\_sum, new\_cur\_sum1, new\_cur\_sum2)

# update my variables

cur\_sum1 = new\_cur\_sum1

cur\_sum2 = new\_cur\_sum2

return max\_sum

**Cat and Mouse II**

1. Use DFS + DP

from functools import lru\_cache

class Solution:

def canMouseWin(self, grid, catJump, mouseJump):

n = len(grid)

m = len(grid[0])

cat = None

mouse = None

aval = 0

#count available cells in grid

for r,line in enumerate(grid):

for c in range(len(line)):

if grid[r][c] != '#':

aval += 1

if grid[r][c] == 'C':

cat = (r,c)

elif grid[r][c] == 'M':

mouse = (r,c)

#cache a reuslt by key(cat,mouse, turn)

@lru\_cache(None)

def dfs(cat, mouse, turn):

#cell number are twice for cat and mouse moves

if turn == (2 \* aval):

return False

steps = catJump if turn % 2 == 1 else mouseJump

# go in on direction

for dr, dc in [[1, 0], [-1, 0], [0, 1], [0, -1]]:

for d in range(steps + 1):

#cat move

if turn % 2 == 1:

nr = cat[0] + dr \* d

nc = cat[1] + dc \* d

if 0 <= nr < n and 0 <= nc < m and grid[nr][nc] != '#':

if (nr, nc) == mouse or grid[nr][nc] == 'F' or not dfs((nr,nc), mouse, turn + 1):

return False

else:

#stop going

break

else:

#mouse move

nr = mouse[0] + dr \* d

nc = mouse[1] + dc \* d

if 0 <= nr < n and 0 <= nc < m and grid[nr][nc] != '#':

if grid[nr][nc] == 'F' or dfs(cat, (nr,nc), turn + 1):

return True

else:

#stop going

break

#important:

# if cat can't catch either mouse or food then mouse wins

if turn % 2 == 0:

return False

#otherwise cat wins and mouse looses

return True

return dfs(cat, mouse, 0)

**Maximum Number of Events That Can Be Attended II**

1. Sort events by start
2. Use binary search to find events which starts > prep
3. Use DP.

class Solution:

def maxValue(self, events, k):

def find(lst, x):

lo = 0

hi = len(lst)

while lo < hi:

mid = lo + (hi - lo) // 2

if x >= lst[mid][0]:

lo = mid + 1

else:

hi = mid

return lo

def calc\_value(lst, prev\_end, rest, mem):

if (prev\_end, rest) in mem:

return mem[(prev\_end, rest)]

if rest == 0:

return 0

max\_val = 0

s = find(lst, prev\_end)

for i in range(s, len(lst)):

val1 = lst[i][2] + calc\_value(lst, lst[i][1], rest - 1, mem)

max\_val = max(max\_val, val1)

mem[(prev\_end, rest)] = max\_val

return max\_val

# N log N

sorted\_by\_start = sorted(events, key=lambda x: x[0])

return calc\_value(sorted\_by\_start, -1, k, {})

**Minimum Cost For Tickets**

1. Use DP

class Solution:

def mincostTickets(self, days, costs) -> int:

dayset = set(days)

def find\_min\_cost(d, mem):

if d in mem:

return mem[d]

if days[-1] < d:

return 0

min\_cost = find\_min\_cost(d + 1, mem)

if d in dayset:

#1 day pass

min\_cost = min(costs[0] + min\_cost,costs[1] + find\_min\_cost(d + 7, mem), costs[2] + find\_min\_cost(d + 30, mem))

mem[d] = min\_cost

return mem[d]

return find\_min\_cost(0, {})

**Minimum Operations to Make Array Equal**

1. Sum of odd is n ^2
2. Our target is n
3. Every couple of odd is make (n - 1), (n - 3) … adding/subtracting
4. Sum up (n-1) + (n-3) + …

class Solution:

def minOperations(self, n):

if n <= 1:

return 0

diff\_sum = 0

# (n - i) where i is odd is number needs to add/subtract.

# find sum (n-1) + (n - 3) + (n-5)

while n > 0:

diff\_sum += n - 1

# calculate the next number

n -= 2

return diff\_sum

>>>>**Wiggle Sort II**

1. Find n//2 median using quick select.
2. Using 3-way sorting with virtual indexes

class Solution:

def wiggleSort(self, nums):

def parition(s, e):

i = s

x = nums[e]

for j in range(s, e):

if nums[j] > x:

nums[i], nums[j] = nums[j], nums[i]

i += 1

nums[e], nums[i] = nums[i], nums[e]

return i

def quick\_select(s, e, k):

# if k > 0 but [e - s+1]

if 0 < k <= e - s + 1:

# find p element

p = parition(s, e)

# if found K element, then return it.

if p - s + 1 == k:

return nums[p]

# otherwsie if k is in left part

if p - s + 1 > k:

return quick\_select(s, p - 1, k)

# otherwise it is in right part

# calculate delta (k - p - 1)

return quick\_select(p + 1, e, s + (k - p - 1))

def mapInd(i):

return (2 \* i + 1) % (len(nums) | 1)

n = len(nums)

#find median

median = quick\_select(0, n - 1, n // 2)

#use 3-way sorting

left = 0

i = 0

right = n - 1

#from 0 to left > median, left to i - unsorted, i to right is less than median

while i <= right:

if nums[mapInd(i)] > median:

#put it at the begining

nums[mapInd(left)],nums[mapInd(i)] = nums[mapInd(i)],nums[mapInd(left)]

left += 1

i += 1

elif nums[mapInd(i)] < median:

#put it at the end

nums[mapInd(right)],nums[mapInd(i)] = nums[mapInd(i)],nums[mapInd(right)]

right -= 1

else:

# nothing to change

i += 1

return nums

**Design Most Recently Used Queue**

1. Use square root decomposition: split array on sqrt(n) blocks.
2. Find block with item and remove this number, than shift number from begging of right block to the end of left block.
3. import math
4. from collections import defaultdict

import math

from collections import defaultdict

class MRUQueue:

def \_\_init\_\_(self, n: int):

self.n = n

#size of one block

self.sz = int(math.sqrt(n))

#block list

self.blocks = defaultdict(list)

#append number to block list

for i in range(1, n + 1):

self.blocks[(i - 1) // self.sz].append(i)

def fetch(self, k):

block\_num = (k - 1) // self.sz

#position inside block

j = (k-1) % self.sz

res = self.blocks[block\_num][j]

#pop item from block list

self.blocks[block\_num].pop(j)

#number of last block

last = (self.n - 1) // self.sz

#append number to last block

self.blocks[last].append(res)

# shift blocks

for i in range(last, block\_num, -1):

x = self.blocks[i].pop(0)

self.blocks[i - 1].append(x)

return res

**Tuple with Same Product**

1. Use map to store product of nums[i],nums[j]
2. If we meet this product res += 8 \* m[product]
3. m[product] += 1

from collections import defaultdict

class Solution:

def tupleSameProduct(self, nums):

m = defaultdict(int)

tuples = 0

for i in range(len(nums) - 1):

for j in range(i+1, len(nums)):

#calculate the product

product = nums[i] \* nums[j]

# if we meet this product earlier,let count them as 8 cases

if product in m:

tuples += (8 \* m[product])

#increase count of this product

m[product] += 1

return tuples

**Largest Submatrix With Rearrangements**

1. Calculate row height for every column
2. If column is 0,reset height to 0
3. Sorted heights
4. Go from left to right and calculate maximal area: h[j]\*(m-j)

class Solution:

def largestSubmatrix(self, matrix):

n = len(matrix)

m = len(matrix[0])

# height of rows: go from to to bottom

height = [0] \* m

max\_area = 0

for i in range(n):

#if meet 0,let reset the height to 0

for j in range(m):

if matrix[i][j] == 0:

height[j] = 0

else:

height[j] += 1

#sort height to place the same number together

sorted\_height = sorted(height)

#go from left to right, calculate minimal area.

for j in range(m):

max\_area = max(max\_area, sorted\_height[j] \* (m - j))

return max\_area

**XOR Queries of a Subarray**

1. Use xor prefix
2. If l > 0 Res = pr[r] ^pr[l-1] else pr[r]

class Solution:

def xorQueries(self, arr, queries):

prefix = [0] \* len(arr)

prefix[0] = arr[0]

#make prefix of xored values

for i in range(1, len(arr)):

prefix[i] = prefix[i-1] ^ arr[i]

res = []

# values from 0 to l will be zero.

for l,r in queries:

if l == 0:

res.append(prefix[r])

else:

res.append(prefix[r] ^ prefix[l - 1])

return res

**Count Substrings with Only One Distinct Letter**

1. Use counter to save number of substring ending at I position
2. Update total by this count

class Solution:

def countLetters(self, S: str) -> int:

#count of substrings ending at i

#if one char is 1 substring

count = 1

total = count

for i in range(1, len(S)):

#if current char is as previois, let increment current count of substring ending at 1

if S[i] == S[i-1]:

count = count + 1

else:

# set number of substrings to 1

count = 1

total += count

#count total number of all substring with one character

return total

**Change Minimum Characters to Satisfy One of Three Conditions**

1.For every char from alphabet calculate char is largest than a and smaller than b and opposite way, or make both a and b with distinct char

2. For a and z don’t make sense

=class Solution:

def minCharacters(self, a, b):

ca = [0] \* 26

cb = [0] \* 26

#find a frequencies

for ch in a:

ca[ord(ch) - ord('a')] += 1

#find b frequencies

for ch in b:

cb[ord(ch) - ord('a')] += 1

m = len(a)

n = len(b)

res = m + n

# exclude 'a' and z' from alphabet

for i in range(26):

# how many operations is needed to if a and b consist of one distinct letter i:

# let subtract lena+lenb - ai - bi (i -frequencies), suppose m + n - c1[i] - c2[i] - are the same letters

res = min(res, m + n - ca[i] - cb[i])

# for all chars except 'a'

if i > 0:

#add frequencies of chars <= i in a and b

ca[i] += ca[i - 1]

cb[i] += cb[i - 1]

# for all chars except 'z'

if i < 25:

#how many operations is needed to make i the largest letter in a and smaller than the smallest one in b

# m - ca[i] is number chars > i,so we should replace them in a

# n - cb[i] is number chars > i, so we should replace them in b

# cb[i] is number chars <= i in b, we should replace in b

res = min(res, m - ca[i] + cb[i], n - cb[i] + ca[i])

return res

**Find Kth Largest XOR Coordinate Value**

1. Xor every row, xor every column
2. Put all items to minimal queue by size k
3. Return q.top

class FindKthLargestXORCoordinateValue {

public:

int kthLargestValue(vector<vector<int>> &matrix, int k) {

priority\_queue<int> pq;

int n = matrix.size();

int m = matrix[0].size();

if (m > 1) {

for (int i = 0; i < n; i++) {

for (int j = 1; j < m; j++) {

matrix[i][j] ^= matrix[i][j - 1];

}

}

}

if (n > 1) {

for (int j = 0; j < m; j++) {

for (int i = 1; i < n; i++) {

matrix[i][j] ^= matrix[i - 1][j];

}

}

}

for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

if (pq.size() == k) {

*//evict fewest item*

if (-pq.top() < matrix[i][j]) {

pq.pop();

pq.push(-matrix[i][j]);

}

} else pq.push(-matrix[i][j]);

}

}

return -pq.top();

}

};

**Partition Array into Disjoint Intervals**

1. Create max left array and min right array
2. Find maxleft[i-1] <= minright[i]

class Solution:

def partitionDisjoint(self, A: List[int]) -> int:

minRight = [0] \* len(A)

maxLeft = [0] \* len(A)

m = A[-1]

for i in range(len(A)-1, -1, -1):

m = min(m, A[i])

minRight[i] = m

m = A[0]

for i in range(len(A)):

m = max(m, A[i])

maxLeft[i] = m

for i in range(1, len(A)):

if maxLeft[i-1] <= minRight[i]:

return I

**Pancake Sorting**

**class Solution:**

**def pancakeSort(self, arr: List[int]) -> List[int]:**

**def flip(arr, k):**

**i = 0**

**j = k - 1**

**while i <= j:**

**arr[i],arr[j] = arr[j],arr[i]**

**i += 1**

**j -= 1**

**n = len(arr)**

**res = []**

**for a in range(n, 0, -1):**

**#find at which index is num (a+1)**

**j = arr.index(a)**

**#if number is at right pos, skip it**

**if (a - 1) == j:**

**continue**

**#move item to 0 pos**

**if j != 0:**

**#move a to 0 pos**

**flip(arr, j + 1)**

**res.append(j + 1)**

**#move item to original pos**

**flip(arr, a)**

**res.append(a)**

**return res**

**Bulb Switcher III**

class Solution:

def numTimesAllBlue(self, light):

res = 0

#put all bulbs have been switched on

max\_bulb = 0

for i in range(len(light)):

max\_bulb = max(max\_bulb, light[i])

#if number of switched on bulbs is i + 1 to max\_bulbs is i+1, then we've switched on all bulbs to the left,

# so max\_bulbs is blue

if max\_bulb == i + 1:

res += 1

return res

**Subarray Sum Equals K**

1. Use cumulative sum and represent k = sum[i] - sum[j] and put sum[i] to map with counter
2. Check if cur\_sum - k in map and if so update res by map[cur\_sum - k] counter

from collections import defaultdict

class Solution:

def subarraySum(self, nums: List[int], k: int) -> int:

cur\_sum = 0

count = 0

#store cumulative sum [:i]

dp = defaultdict(int)

dp[0] = 1

for i in range(len(nums)):

cur\_sum += nums[i]

# if exist sum[j] - sum[i] == k

if cur\_sum - k in dp:

count += dp[cur\_sum - k]

#update cumulative sum counter

dp[cur\_sum] += 1

return count

**Make Sum Divisible by P**

1. Use map where store sum %p -> index
2. R = sum\_array % p, so sum[i:j] % p = sum\_arr % p
3. Need to find (s[0:i] - r) % p in map, if so
4. s[0:j] % p = s[0:i] - r) % p where i > j,

class Solution:

def minSubarray(self, nums: List[int], p: int) -> int:

sum\_arr = sum(nums)

if sum\_arr % p == 0:

return 0

min\_size = len(nums)

m = defaultdict(int)

# removed subarray that is remainder of sum\_arr % p

r = sum\_arr % p

m[0] = -1

s = 0

#find subarray which sum is s - r

for i in range(len(nums)):

# module sum

s += nums[i]

# find (s - r) % p in m

k = (s - r) % p

if k in m:

#calculate min length of subarray with sum is s - r

min\_size = min(min\_size, i - m[k])

m[s % p] = i

return min\_size if min\_size!= len(nums) else -1

**Restore the Array From Adjacent Pairs**

1. Find ending vertex
2. Start from it go append next vertices

from collections import defaultdict

class Solution:

def restoreArray(self, adjacentPairs):

m = defaultdict(list)

#find number with frequency is being 1

for i in range(len(adjacentPairs)):

a,b = adjacentPairs[i]

m[a].append(i)

m[b].append(i)

#number with frequence equals 1 is ending items.

v = [v for v,l in m.items() if len(l) == 1]

q = [v[0]]

n = len(adjacentPairs) + 1

seen = set()

#find next pair use BFS

while len(q) < n:

#find last number from previous m[a] list

for i in m[q[-1]]:

if i in seen:

continue

else:

break

seen.add(i)

#find next pair and it to queue

a,b = adjacentPairs[i]

if a == q[-1]:

q.append(b)

else:

q.append(a)

return q

**Can You Eat Your Favorite Candy on Your Favorite Day?**

1. Use min days = count of candies I - 1 // dayCap
2. Max days = count of candies I - 1

class Solution:

def canEat(self, candiesCount, queries):

prefix = [0] \* len(candiesCount)

prefix[0] = candiesCount[0]

for i in range(1, len(candiesCount)):

prefix[i] = prefix[i - 1] + candiesCount[i]

prefix = [0] + prefix

ans = [False] \* len(queries)

#go through queries

for i in range(len(queries)):

candy,day,cap = queries[i]

#find min and max I can eat the number of candies

maxDays = prefix[candy + 1] - 1 #eat one candy per day, d starts from 0

minDays = prefix[candy] // cap # for days I eat candies of type candy - 1

ans[i] = minDays <= day <= maxDays

return ans

**Palindrome Partitioning II**

1. Use d as cut number from I to n - 1
2. d[i] = min(1+ d[j]) j = I to n-1, and s[i:j] is polindrom otherwise d[i] = n - i - 1

class Solution:

def minCut(self, s: str) -> int:

n = len(s)

#is polindrom for i to j

pol = [[False] \* n for \_ in range(n)]

#cut from i to n-1

# cut number in string [i:n-1]

d = [0] \* n

for i in range(n-1, -1, -1):

# every char is cut point.

d[i] = n - i - 1

#consider every j position as cut point,

#and s[i:j],s[i:j+1],s[i:j+2] and so on by updating d[i] if i:j is polindrom and [i+1:j-1], it may decrease cut number

# we go from lefty to right considering [i,j] snd [j,n-1], so we have cuts calculated for rightmost.

for j in range(i, n):

# if [i,j] is polyndrom?

if s[i] == s[j] and (j - i < 2 or pol[i+1][j-1]):

pol[i][j] = True

# if[i:n-1] is polyndrom,there is no cut points.

if j == n - 1:

d[i] = 0

# if s[j+1:n-1] has d[j+1] cuts it’s less then [j+1,n-1] cuts,let update d[i]

elif d[j + 1] + 1 < d[i]:

d[i] = d[j + 1] + 1

return d[0]

**Palindrome Partitioning IV**

1. Use DP to calculate polindrom for s[i][j]
2. Use 2 pointers to check if prefix + middle + suffix

class Solution:

def checkPartitioning(self, s):

n = len(s)

pol = [[False] \* n for \_ in range(n)]

for i in range(n):

pol[i][i] = True

# l is length of considering string

# find all palindromes

for l in range(2, n):

for i in range(n):

j = i + l - 1

if j == n:

break

if s[i] == s[j] and (l == 2 or pol[i+1][j-1]):

pol[i][j] = True

# go from right to left

for i in range(n - 2):

if not pol[0][i]:

continue

j = n - 1

# considering [i:j] and [j:n-1] are polindroms?

while j > i:

if pol[j][n-1] and pol[i+1][j-1]:

return True

j -= 1

return False

**Maximum Length of Repeated Subarray**

1. Use dp and consider A[i:] and B[j:]

class Solution:

def findLength(self, A, B) -> int:

dp = [[0] \* (len(A) + 1) for \_ in range(len(B) + 1)]

# go from right to left

for i in range(len(A)-1, -1,-1):

#compare A[i] to start in B

for j in range(len(B) - 1, -1, -1):

if A[i] == B[j]:

# if previous chars matched

dp[i][j] = dp[i+1][j+1] + 1

return max(max(r) for r in dp)

**Maximum Score From Removing Stones**

1. If a+b <= c, then result is a+b
2. If a+b > c, then reduce (a,c) and (b,c) until c is 0 and

a should equal b, result is c+ (a+b-c)//2

class Solution:

def maximumScore(self, a: int, b: int, c: int) -> int:

a, b, c = sorted([a, b, c])

# a + b <= c, then it results in a+b pairs

if (a + b) <= c:

return a + b

# a + b > c, then decrease (a,c) and (b,c) until c = 0 and the a== b,so it results in (a+b) / 2

return c + (a + b - c) // 2

**Largest Merge Of Two Strings**

1. Compare substring w1 and w2 if w1 >= w2 take w1[0] + merge(w1[1:] , w2)
2. Otherwise w2[0] + merge(w1,w2[1:])

class Solution:

def largestMerge(self, word1: str, word2: str) -> str:

if not word1 and not word2:

return ''

if not word1:

return word2

if not word2:

return word1

if word1 >= word2:

return word1[0] + self.largestMerge(word1[1:], word2)

return word2[0] + self.largestMerge(word1, word2[1:])

**4Sum**

1. Add all sum a+b to map as key and i,j as value
2. Declare res as set to store them sorted (i1,j1,i2,j2)

3. If exist target - s(key from map) then connect couples to get (i1,j1,i2,j2) and put them to cache.

Discrete logarithm

1 user meet in the middle technique

2. Find n: a ^n = b mod m

If represent n = p \* i - j where p = (sqrt(m) + 1) and i,j 0<= i,j <= sqrt(m)

import math

class Solution:

def powmod(self, a, n, p):

if n == 0:

return 1

if n % 2 == 0:

return self.powmod((a\*a) % p, n // 2, p)

return (a \* self.powmod((a\*a) % p, (n-1) // 2, p)) % p

def discreteLog(self, a, b, m):

# a ^k = b % m, find k

n = int(math.sqrt(m) + 1)

# create map (a ^ k) % m => k for left part

mm = {}

for i in range(n, 0, -1):

cur = self.powmod(a, i \* n, m)

mm[cur] = i

# create right part

for j in range(m):

cur = (self.powmod(a, j, m) \* b) % m

if cur in mm:

# p = i \* n - j, i \*n is in map.

r = mm[cur] \* n - j

if r < m:

return r

return -1

**Closest Subsequence Sum**

1. Use meet in the middle: divide the array by 2:

- generate for left and right parts all possible subsets sums

* sort left part and go through right past and find goal -s in left part with binary search
* using left\_search by condisering left[k] and left[k-1]

import bisect

import math

class Solution:

# fast way to generate subset sums

def generateSubsetSums(self, nums):

# 0 gives to add tne items as sums

ans = {0}

for x in nums:

ans |= {x + y for y in ans}

return ans

def minAbsDifference(self, arr, target):

# divide arr by 2 and generate sum for left and right parts

n = len(arr)

left = sorted(self.generateSubsetSums(arr[:n // 2]))

right = self.generateSubsetSums(arr[n // 2:])

# max sum is less then target

res = math.inf

for s in right:

# return x[:k] < target - s and x[k:] >= target -s

k = bisect.bisect\_left(left, target - s)

# consider two cases:

if k < len(left):

res = min(res, s + left[k] - target)

if k > 0:

res = min(res, target - s - left[k-1])

return res

**Count of Range Sum**

1. Sort prefix sum and count the range sum
2. Calculate prefix sum, use mergeSort to sort prefix sum and count the sum of subarray lying in range

class Solution:

def countRangeSum(self, nums, lower, upper):

n = len(nums)

pr = [0] \* n

pr[0] = nums[0]

for i in range(1, n):

pr[i] = pr[i-1] + nums[i]

pr = [0] + pr

#we merge and sort prefix sum

# r is exclusive

def mergeSort(pr, s, e):

if (e - s) <= 1:

return 0

mid = s + (e - s) //2

count = mergeSort(pr, s, mid) + mergeSort(pr, mid, e)

# we look at second half

k = mid # from pr[k] - pr[i] >= lower

j = mid # pr[j] - pr[i] < upper

t = mid

r = 0

cache = [0] \* (e - s)

for i in range(s, mid):

#consider [i,k,j) subarray

while k < e and (pr[k] - pr[i]) < lower:

k += 1

while j < e and (pr[j] - pr[i]) <= upper:

j += 1

# put into cache pr[t]< pr[i]

while t < e and pr[i] > pr[t]:

cache[r] = pr[t]

t += 1

r += 1

cache[r] = pr[i]

count += (j - k)

r += 1

#put to pr sorted values

j = s

for i in range(t - s):

pr[j] = cache[i]

j += 1

i += 1

return count

return mergeSort(pr, 0, len(pr))

**Minimum Limit of Balls in a Bag**

1. Try to find max number of balls to split all bag within maxOperation, then it will be possible penalty.
2. Use binary search

class Solution:

def minimumSize(self, nums, maxOperations):

lo = 1

hi = 10 \*\* 9

def canDivide(sz):

split = 0

for b in nums:

#count spilt number to spent to split this bag with balls

split += (b - 1) // sz

if split > maxOperations:

return False

return True

while lo < hi:

#calculate balls for one bag

sz = lo + (hi - lo) // 2

#find leftmost max size that can't be used to split bags

if canDivide(sz):

# use division, so for big sz, restrict upper boundv of balls.

hi = sz

else:

# too samll ball size to spilt bags,so increase size

lo = sz + 1

return lo

**Super Ugly Number**

from heapq import heappop,heappush

class Solution:

def nthSuperUglyNumber(self, n, primes) -> int:

ugly = [0] \* n

ugly[0] = 1

q = []

# put minimal heap with val, index of last ugly, prime

for i in range(len(primes)):

heappush(q, (primes[i], 1, primes[i]))

for i in range(1, n):

#take min val

a,\_,\_ = q[0]

ugly[i] = a

# until last ugly equals heap top

while q[0][0] == ugly[i]:

# val,idx, p

val,idx,p = heappop(q)

#multiply every prime on ugly[1...n] and increase index

heappush(q, (p \* ugly[idx], idx + 1, p))

return ugly[-1]

**Single Element in a Sorted Array**

1. Use binary search by noticing that part is with odd length contains item with frequency 1
2. Consider the cases:

Mid parent is to the left/right

class Solution:

def singleNonDuplicate(self, nums):

lo = 0

hi = len(nums) - 1

#array shouldn't be sorted but should being pairing

while lo < hi: # strict inequality

m = lo + (hi-lo) // 2

halves\_is\_even = (hi - m) % 2 == 0

if nums[m] == nums[m+1]:

# mid 's partner is to the right

if halves\_is\_even:

lo = m + 2

else:

hi = m - 1

elif nums[m-1] == nums[m]:

# mid partner is to the left

if halves\_is\_even:

hi = m - 2

else:

lo = m + 1

else:

return nums[m]

return nums[lo]

**Capacity To Ship Packages Within D Days**

1. Use binary search to find load for day
2. Use the function the can\_ship to find hi = m
3. Not lo

Minimum Degree of a Connected Trio in a Graph

1. Brute force

rom collections import defaultdict

import math

class Solution:

def minTrioDegree(self, n: int, edges):

vertices = []

for a, b in edges:

vertices.append(sorted([a, b]))

vertices.sort()

adj\_list = defaultdict(set)

for a, b in edges:

adj\_list[a].add(b)

adj\_list[b].add(a)

res = math.inf

for i in range(len(vertices) - 2):

a, b = vertices[i]

d1 = len(adj\_list[a])

d2 = len(adj\_list[b])

if d1 < 2 or d2 < 2:

continue

for j in range(i + 1, len(vertices)):

c, d = vertices[j]

if c != a:

break

# a == c, then try to find (b,d)

if b in adj\_list and d in adj\_list[b]:

d3 = len(adj\_list[d])

res = min(res, d1 + d2 + d3 - 6)

return res if res != math.inf else -1

**Maximum Score from Performing Multiplication Operations**

1. Use DP:

class Solution:

def maximumScore(self, nums, multipliers):

@lru\_cache(2000)

def find\_max(l, i):

if i == len(multipliers):

return 0

res1 = find\_max(l + 1, i + 1) + nums[l] \* multipliers[i]

res2 = find\_max(l, i + 1) + nums[len(nums) - (i-l) - 1] \* multipliers[i]

return max(res1, res2)

return find\_max(0, 0)

**Maximize Palindrome Length From Subsequences**

1. Concatenate word1 and word2
2. Find longest palindromic subsequence
3. Checke if every word1[i] == word2[j] and update max\_len with dp[i][j+n]

class Solution:

def longestPalindrome(self, word1: str, word2: str) -> int:

n = len(word1)

m = len(word2)

h = m + n

#find max polindromic subsequence

word = word1 + word2

dp = [[0] \* h for \_ in range(h)]

for i in range(h):

dp[i][i] = 1

for l in range(2, h + 1):

for i in range(h - l + 1):

j = i + l - 1

if word[i] == word[j]:

dp[i][j] = dp[i+1][j-1] + 2

else:

dp[i][j] = max(dp[i+1][j], dp[i][j-1])

#go through and find such dp[i][j] that has i < n and j >= n

max\_len = 0

#consider polindromic strings if one is in word1 and another is in word2

for i in range(n):

for j in range(m):

if word1[i] == word2[j]:

max\_len = max(max\_len, dp[i][j + n])

return max\_len

**Minimum Number of Operations to Move All Balls to Each Box**

1. Go from left to right and count balls and update work by balls
2. Go from right to left doing the same and set up work in res.

class Solution:

def minOperations(self, boxes: str):

res = [0] \* len(boxes)

#accumulate balls

cnt = 0

# work to move cnt balls to i-position

ops = 0

# go from the left to right

for i in range(len(boxes)):

res[i] += ops

if boxes[i] == '1':

cnt += 1

ops += cnt

ops = 0

cnt = 0

#go from right to the left

for i in range(len(boxes)-1,-1,-1):

res[i] += ops

if boxes[i] == '1':

cnt += 1

ops += cnt

return res

**Numbers At Most N Given Digit Set**

1. If first of N is less than digits, then it could be numbers d1111 to d9999 and count it len(digits) \*\* (k-1) otherwise N[0] == S[0] it’s just dp[i] += dp[i+1]

class Solution:

def atMostNGivenDigitSet(self, digits, n: int) -> int:

s = str(n)

k = len(s)

# store numbor of [i:n]

dp = [0] \* k + [1]

for i in range(k-1,-1,-1):

for d in digits:

if s[i] > d:

#add count of s[i]xxxx - num of digits \*\* k-1

dp[i] += len(digits) \*\* (k-i-1)

elif s[i] == d:

# it is only dxxx number

dp[i] += dp[i+1]

return dp[0] + sum([len(digits) \*\* i for i in range(1,k)])

**Divide Chocolate**

Use Binary search and Greedy approach

class Solution:

def maximizeSweetness(self, sweetness, K: int):

#min sweetness is sweetness of one chunk

lo = min(sweetness)

#max is total sum of all chunks

hi = sum(sweetness)

def count\_pieces(min\_piece):

#count if every piece may be not less than min\_piece

cur\_sweet = 0

c = 0

for i in range(len(sweetness)):

cur\_sweet += sweetness[i]

if cur\_sweet >= min\_piece:

c += 1

cur\_sweet = 0

return c

# increment for me

K += 1

#find max of min sweetness

while lo <= hi:

m = lo + (hi - lo) // 2

# if we can't split for K persons, then reduce min\_sweetness

if count\_pieces(m) < K:

hi = m - 1

else:

#otherwise increase it

lo = m + 1

# here is not obvious but we return hi as min(lo,hi) in the end.

return hi

**Koko Eating Bananas**

1. Use binary search

class Solution:

def minEatingSpeed(self, piles, h: int) -> int:

lo = 1

hi = sum(piles)

def can\_eat(v):

return sum([(p - 1) // v + 1 for p in piles]) <= h

while lo < hi:

m = lo + (hi - lo) // 2

if can\_eat(m):

hi = m

else:

lo = m + 1

return lo

**Minimize Max Distance to Gas Station**

1. Use binary search and with real numbers:

class Solution:

def minmaxGasDist(self, stations, k: int) -> float:

dist = []

for i in range(1, len(stations)):

dist.append(stations[i] - stations[i - 1])

lo = 0

hi = sum(dist)

def count\_stations(dist, d):

return sum([int(dst / d) for dst in dist])

while hi - lo > 10 \*\* (-6):

m = lo + (hi - lo) / 2.0

if count\_stations(dist, m) <= k:

hi = m

else:

lo = m

return hi

**Equal Sum Arrays With Minimum Number of Operations**

class Solution:

def minOperations(self, nums1, nums2):

if len(nums2) > 6 \* len(nums1) or len(nums1) > 6 \* len(nums2):

return -1

s1 = sum(nums1)

s2 = sum(nums2)

cnt = 0

nums1.sort()

nums2.sort()

if s1 >= s2:

nums1,nums2 = nums2,nums1

s1,s2 = s2,s1

#s2 > s1

i = 0

j = len(nums2) - 1

while s2 > s1:

if j < 0 or i < len(nums1) and (6 - nums1[i]) > (nums2[j] - 1):

s1 += 6 - nums1[i]

i += 1

else:

s1 += nums2[j] - 1

j -= 1

cnt += 1

return cnt

**Minimum Swaps to Make Strings Equal**

class Solution:

def minimumSwap(self, s1: str, s2: str) -> int:

n = len(s1)

x1 = 0

y1 = 0

x2 = 0

y2 = 0

for i in range(n):

if s1[i] == s2[i]:

continue

if s1[i] == 'x':

x1+=1

else:

y1+=1

if s2[i] == 'x':

x2 += 1

else:

y2 += 1

# if number of x or y is odd, return -1

if (x1+y1) % 2 != 0 or (y2+x2) % 2 != 0:

return -1

#consider case 1: xx / yy

#s1xx,s2yy

c1 = min(x1//2,y2 // 2)

#s1yy,s2xx

c2 = min(y1//2,x2 // 2)

cnt = c1 + c2

x1 -= c1\*2

y2 -= c1\*2

x2 -= c2\*2

y1 -= c2\*2

cnt += (x1 + y1)

return cnt

**Coin Path**

**class Solution:**

**def cheapestJump(self, A: List[int], B: int) -> List[int]:**

**n = len(A)**

**nxt = [-1] \* n**

**dp = [0] \* n**

**for i in range(n - 2,-1,-1):**

**min\_cost = math.inf**

**for j in range(i+1,min(i+B+1, n)):**

**if A[j] == -1:**

**continue**

**c = A[i] + dp[j]**

**if min\_cost > c:**

**nxt[i] = j**

**min\_cost = c**

**dp[i] = min\_cost**

**res = []**

**i = 0**

**while i < n and i >= 0:**

**res.append(i + 1)**

**i = nxt[i]**

**if res and res[-1] == n:**

**return res**

**return []**

**Maximum Number of Events That Can Be Attended**

**from heapq import heappop,heappush**

**c**lass Solution:

def maxEvents(self, events):

#shorter events come first

sorted\_by\_start = sorted(events,key=lambda x:x[0])

cnt = 0

pq = []

j = 0

for d in range(1, 10\*\*5+1):

#remove the events that have been finished

while pq and pq[0] < d:

heappop(pq)

#add the events that started this day

while j < len(sorted\_by\_start) and sorted\_by\_start[j][0] == d:

heappush(pq, sorted\_by\_start[j][1])

j += 1

#take the shortest event.

if pq:

heappop(pq)

cnt += 1

return cnt

Lowest Common Ancestor of a Binary Tree II

class Solution:

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

#euler index to level

depth = {}

#node to euler index

nodes = {}

#euler index to node

et = {}

self.et = 0

def visit(n, l, depth, nodes, et):

depth[self.et] = l

nodes[n] = self.et

et[self.et] = n

self.et += 1

def dfs(n, l, depth, nodes, et):

if not n:

return

visit(n, l, depth, nodes,et)

dfs(n.left, l + 1, depth, nodes,et)

visit(n, l, depth, nodes,et)

dfs(n.right, l + 1, depth, nodes,et)

visit(n, l, depth, nodes,et)

dfs(root, 0, depth, nodes,et)

if p not in nodes or q not in nodes:

return None

i = nodes[p]

j = nodes[q]

#euler indices

l = min(j, i)

r = max(i, j)

i = min(range(l,r + 1),key=lambda x:depth[x])

return et[i]

**Car Fleet II**

**class Solution:**

**def getCollisionTimes(self, cars):**

**n = len(cars)**

**ans = [-1.0] \* n**

**#left in the stack that current car can collide**

**st = []**

**for i in range(len(cars) - 1, -1, -1):**

**p,v = cars[i]**

**while st:**

**j = st[-1]**

**p2,v2 = cars[j]**

**#pop if car has higher speed**

**#or car in st[-1] bumped into previous car earlier than current car collides st[-1]**

**if cars[j][1] >= v or (p2 - p) / (v-v2) >= ans[j] and ans[j] >= 0:**

**st.pop()**

**else:**

**break**

**#we poped out all cars we can't collide, so calculate crash time)**

**if st:**

**j = st[-1]**

**p2,v2 = cars[j]**

**ans[i] = (p2 - p) / (v-v2)**

**st.append(i)**

**return ans**

**Car Fleet**

**class Solution:**

**def carFleet(self, target: int, position: List[int], speed: List[int]) -> int:**

**if not speed:**

**return 0**

**cnt = 0**

**n = len(speed)**

**index = list(range(n))**

**index.sort(key=lambda i:position[i])**

**slowest\_time = 0**

**for i in range(n - 1, -1, -1):**

**p = position[index[i]]**

**v = speed[index[i]]**

**t = (target - p) / v**

**if t > slowest\_time:**

**# new fleet**

**cnt += 1**

**slowest\_time = t**

**return cnt**

**Daily Temperatures**

**class Solution:**

**def dailyTemperatures(self, temperatures: List[int]) -> List[int]:**

**st = []**

**res = [0] \* len(temperatures)**

**for i in range(len(temperatures)):**

**while st and temperatures[st[-1]] < temperatures[i]:**

**j = st[-1]**

**st.pop()**

**res[j] = i - j**

**st.append(i)**

**return res**

**Longest increasing subsequence in logn:**

**dp holds item on which the subsequence by length** I terminates.

def longest\_increasing\_subsequence\_length(nums):

n = len(nums)

dp = [math.inf] \* (n + 1)

dp[0] = -math.inf

for i in range(n):

j = bisect.bisect\_left(dp, nums[i])

if dp[j-1] < nums[i] and nums[i] < dp[j]:

dp[j] = nums[i]

j = 0

for i in range(n + 1):

if dp[i] < math.inf:

j = i

return j

**Number of Restricted Paths From First to Last Node**

**from collections import defaultdict**

**from heapq import heappop,heappush**

**from math import inf**

**class Solution:**

**def countRestrictedPaths(self, n: int, edges):**

**adj\_list = defaultdict(list)**

**for s,e,w in edges:**

**adj\_list[s].append([w, e])**

**adj\_list[e].append([w,s])**

**dist = [inf] \* (n + 1)**

**#find short path for i**

**q = [[0, n]]**

**dist[n] = 0**

**#use Dikjistra**

**while q:**

**\_,node = heappop(q)**

**for w,nei in adj\_list[node]:**

**if dist[nei] > dist[node] + w:**

**dist[nei] = dist[node] + w**

**heappush(q,[dist[nei],nei] )**

**def dfs(dist, adj\_list, x, mem):**

**if x in mem:**

**return mem[x]**

**if x == n:**

**return 1**

**res = 0**

**for w,nei in adj\_list[x]:**

**if dist[x] > dist[nei]:**

**res = (res + dfs(dist, adj\_list, nei, mem)) % (10 \*\* 9 + 7)**

**mem[x] = res**

**return mem[x]**

**return dfs(dist, adj\_list, 1, {})**

**Maximum Score of a Good Subarray**

1. Use 2 pointers technic

**class Solution:**

**def maximumScore(self, nums: List[int], k: int) -> int:**

**if not nums:**

**return 0**

**n = len(nums)**

**i = k**

**j = k**

**min\_so\_far = nums[k]**

**score = nums[k]**

**while i > 0 or j < (n - 1):**

**if i == 0:**

**j += 1**

**elif j == n - 1:**

**i -= 1**

**elif nums[i-1] < nums[j + 1]:**

**j += 1**

**else:**

**i -= 1**

**min\_so\_far = min(min\_so\_far, nums[i], nums[j])**

**score = max(score, min\_so\_far \* (j - i + 1))**

**return score**

**Longest increasing subsequence for NlogN**

import bisect

import math

def longest\_increasing\_subsequence\_length(nums):

n = len(nums)

dp = [math.inf] \* (n + 1)

#store nums[i] at witch subsequence [0:i] breaks

dp[0] = -math.inf

for i in range(n):

#search dp[i] <= nums[i]

j = bisect.bisect\_left(dp, nums[i])

if dp[j-1] < nums[i] and nums[i] < dp[j]:

#update at leftmost dp

dp[j] = nums[i]

j = 0

for i in range(n + 1):

if dp[i] < math.inf:

#j is length off

j = i

return j

Connect nodes at same level

lass ListNode:

def \_\_init\_\_(self, v):

self.val = v

self.left = None

self.right = None

self.nextRight = None

class Solution:

def connect(self, root):

def getNextNode(node):

t = node.nextRight

while t:

if t.left:

return t.left

if t.right:

return t.right

t = t.nextRight

return None

def connect(node):

if not node:

return

if node.nextRight:

connect(node.nextRight)

if node.left:

if node.right:

node.left.nextRight = node.right

node.right = getNextNode(node)

else:

node.left = getNextNode(node)

connect(node.left)

elif node.right:

node.right.nextRight = getNextNode(node)

connect(node.right)

else:

connect(getNextNode(node))

connect(root)

**132 Pattern**

class Solution:

def find132pattern(self, nums):

if not nums:

return False

left = [0] \* len(nums)

left[0] = nums[0]

for i in range(1, len(nums)):

left[i] = min(left[i-1],nums[i])

st = []

for i in range(len(nums)-1, -1,-1):

if i and nums[i] <= left[i]:

continue

# keep increasing subsequence

while st and nums[i] > nums[st[-1]]:

k = st.pop()

if left[i] < nums[k]:

return True

st.append(i)

return False

**Maximum Average Pass Ratio**

**class Solution:**

**def maxAverageRatio(self, classes: List[List[int]], extraStudents: int) -> float:**

**if not classes:**

**return 0**

**def profit(p,t):**

**return (p+1) / (t+1) - p / t**

**q = []**

**for p,t in classes:**

**#p = 1.0 \* p**

**#t = 1.0 \* t**

**q.append([-profit(p,t), p,t])**

**heapify(q)**

**ratio = 0**

**for i in range(extraStudents):**

**\_,p,t = heappop(q)**

**p += 1**

**t += 1**

**heappush(q, [-profit(p,t), p, t])**

**for \_,p,t in q:**

**ratio += p / t**

**return ratio / len(classes)**

**Next Greater Element II**

**class Solution:**

**def nextGreaterElements(self, nums: List[int]) -> List[int]:**

**if not nums:**

**return None**

**st = []**

**res = [-math.inf] \* len(nums)**

**#find all right greater number**

**for i in range(len(nums)-1,-1,-1):**

**while st and nums[i] >= nums[st[-1]]:**

**st.pop()**

**if st:**

**res[i] = nums[st[-1]]**

**st.append(i)**

**#fill stack with items is greater than nums[-1]**

**# go from left to right by popping the items is less or equal to curent item having -1**

**# all rightmost items are less or equal then curent item is.**

**st = []**

**for i in range(len(nums)-2,-1,-1):**

**if nums[i] > nums[-1]:**

**st.append(i)**

**for i in range(len(nums)-1,-1,-1):**

**if res[i] != -math.inf:**

**continue**

**while st and nums[i] >= nums[st[-1]]:**

**st.pop()**

**if st:**

**res[i] = nums[st[-1]]**

**return [a if a != -math.inf else -1 for a in res]**

**Shortest Subarray with Sum at Least K**

**class Solution:**

**def shortestSubarray(self, nums: List[int], k: int) -> int:**

**if not nums:**

**return -1**

**min\_len = math.inf**

**pr = [0]**

**for n in nums:**

**pr.append(pr[-1] + n)**

**q = []**

**for i in range(len(pr)):**

**while q and pr[i] <= pr[q[-1]]:**

**q.pop()**

**while q and pr[i] - pr[q[0]] >= k:**

**min\_len = min(min\_len, i - q[0])**

**q.pop(0)**

**q.append(i)**

**return min\_len if min\_len != math.inf else -1**

class Solution:

def minOperationsToFlip(self, exp: str) -> int:

d = {}

def brackets(s, d):

st = []

for i,ch in enumerate(exp):

if ch == '(':

st.append(i)

elif ch == ')':

d[i] = st.pop()

def dfs(s, e, d):

if s == e:

return[int(exp[s]), 1]

# check if it's another bracket

s2 = d.get(e, e)

if s2 == s:

return dfs(s + 1,e - 1, d)

p1,c1 = dfs(s, s2-2,d)

p2,c2 = dfs(s2, e,d)

op = exp[s2 - 1]

t = {'|': lambda x,y:x|y, '&': lambda x,y: x & y}

c3 = 1 if p1 + p2 == 1 else min(c1, c2) + (p1 ^ (op == "&"))

return [t[op](p1,p2), c3]

brackets(exp, d)

return dfs(0,len(exp) - 1, d)[1]

**Online Stock Span**

class StockSpanner:

def \_\_init\_\_(self):

self.stocks = []

self.lens = []

def next(self, price: int) -> int:

cnt = 1

#keep decreasing order

while self.stocks and self.stocks[-1] <= price:

cnt += self.lens.pop()

self.stocks.pop()

self.stocks.append(price)

self.lens.append(cnt)

return cnt

**Maximum Value at a Given Index in a Bounded Array**

class Solution:

def maxValue(self, n: int, index: int, maxSum):

def can\_place(cur\_sum):

a0 = max(cur\_sum - index, 0)

an = max(cur\_sum - (n - index - 1), 0)

s1 = (a0 + cur\_sum) \* (cur\_sum - a0 + 1) // 2

s2 = (cur\_sum - 1 + an) \* (cur\_sum - an) // 2

return s1 + s2 <= maxSum

# below row is filled by 1

maxSum -= n

hi = maxSum

lo = 0

while lo < hi:

# max at index

mid = (lo + hi + 1) // 2

if can\_place(mid):

lo = mid

else:

hi = mid - 1

# + 1 to count the below row.

return lo + 1

**Count Pairs With XOR in a Range**

**class TrieNode:**

**def \_\_init\_\_(self, bitsPerNum=16):**

**self.table = [None,None]**

**self.cnt = 0**

**class Solution:**

**def insertTrie(self, node, n, bitsPerNum=16):**

**for i in range(bitsPerNum,-1,-1):**

**x = (n >> i) & 1**

**if not node.table[x]:**

**node.table[x] = TrieNode()**

**node.table[x].cnt += 1**

**node = node.table[x]**

**def countSmallerPair(self, node, n, limit,bitsPerNum=16):**

**i = bitsPerNum**

**cnt = 0**

**for i in range(bitsPerNum,-1,-1):**

**if not node:**

**break**

**x = (n >> i) & 1**

**y = (limit >> i) & 1**

**if y == 0:**

**#consider next bit, current bits don't matter**

**node = node.table[x]**

**continue**

**#y is 1**

**if node.table[x]:**

**#if trie bit is as x so it gives 0 and it's less 1**

**cnt += node.table[x].cnt**

**# go on with another bit**

**node = node.table[1 - x]**

**return cnt**

**def countPairs(self, nums, low: int, high: int) -> int:**

**if not nums or len(nums) < 2:**

**return 0**

**trie = TrieNode()**

**cnt = 0**

**bitsPerNum = 16**

**for i in range(len(nums)):**

**cnt += self.countSmallerPair(trie, nums[i], high + 1, bitsPerNum) - self.countSmallerPair(trie, nums[i], low, bitsPerNum)**

**self.insertTrie(trie, nums[i], bitsPerNum)**

**return cnt**

**Sum of Subarray Minimums**

**class Solution:**

**def sumSubarrayMins(self, nums: List[int]) -> int:**

**if not nums or len(nums) == 0:**

**return 0**

**#count arr with length 1**

**mod = 10 \*\* 9 + 7**

**n = len(nums)**

**#place prev least item and its default value is max length of subarray to the left**

**left = [i+1 for i in range(n)]**

**#place next least item and its default value is max length of subarray to the right**

**right = [n-i for i in range(n)]**

**st = []**

**for i in range(n):**

**while st and nums[st[-1]] > nums[i]:**

**st.pop()**

**if st:**

**left[i] = i - st[-1]**

**st.append(i)**

**st = []**

**for i in range(n):**

**while st and nums[st[-1]] > nums[i]:**

**j = st.pop()**

**right[j] = i - j**

**st.append(i)**

**sum\_of\_min = 0**

**for i in range(n):**

**sum\_of\_min = (sum\_of\_min + nums[i] \* left[i] \* right[i]) % mod**

**return sum\_of\_min**

**Online Election**

import bisect

import collections

class TopVotedCandidate(object):

def \_\_init\_\_(self, persons, times):

self.count = collections.defaultdict(int)

self.winner = []

c = 0

for t, p in zip(times, persons):

self.count[p] += 1

if self.count[p] >= c:

self.winner.append((t, p))

c = max(c, self.count[p])

def q(self, t):

j = bisect.bisect(self.winner, (t, float('inf')))

return self.winner[j - 1][1]

**Maximize Number of Nice Divisors**

**class Solution:**

**def maxNiceDivisors(self, primeFactors: int) -> int:**

**if primeFactors <= 3:**

**return primeFactors**

**def pow(x, p, mod):**

**res = 1**

**while p:**

**if p & 1 == 1:**

**res = (res \* x) % mod**

**p >>= 1**

**x = (x \* x) % mod**

**return res**

**#maximize that primeFactors number will give max product.**

**# we can achive it if we use 2 and 3 but mostly we will use 3**

**# if number is 4 we represent it as 2 + 2 if remainder is 1**

**div3 = primeFactors // 3**

**rem = primeFactors % 3**

**mod = 10 \*\* 9 + 7**

**if rem == 0:**

**return pow(3, div3, mod)**

**if rem == 1:#there is 4 and it can't represent as 3 + x so does it as 2 \* 2**

**div3 -= 1 # remove one of 3**

**rem = 4 # some of primary factors is 4**

**return (rem \* pow(3, div3, mod)) % mod**

**return (rem \* pow(3, div3, mod)) % mod**

**Check If Array Pairs Are Divisible by k**

**class Solution:**

**def canArrange(self, arr: List[int], k: int) -> bool:**

**if not arr or k == 0:**

**return False**

**count = Counter()**

**for i in range(len(arr)):**

**# count how many numbers has remainders < k**

**# if 2 numbers have the remainder a + b = k that means that number's sum is multiple of k**

**# a + b = k => (a +b) % k = k % k => a % k + b % k = 0 if a + b is multiple of k**

**count[abs(arr[i] % k)] += 1**

**for i in range(k):**

**#check edge cases**

**if (i == 0 or i == k - i) and count[i] % 2 == 0:**

**continue**

**elif count[i] == count[k - i]:**

**continue**

**else:**

**return False**

**return True**

**Integer Break**

**class Solution:**

**def integerBreak(self, n: int) -> int:**

**if n < 3:**

**return 1**

**if n == 3:**

**return 2**

**#maximal product if n is represented as sum of 3+3+3+ .. 2**

**div3 = n // 3**

**rem = n % 3**

**if rem == 0:**

**return int(3 \*\* div3)**

**if rem == 1:**

**#if n == 4, then n is 3+ 3 + .. 2 + 2. With some number of 2**

**div3 -= 1**

**rem = 4**

**return rem \* int(3 \*\* div3)**

**return rem \* int(3 \*\* div3)**

**Sum of Subsequence Widths**

**class Solution:**

**def sumSubseqWidths(self, nums: List[int]) -> int:**

**len\_of\_nums = len(nums)**

**nums.sort()**

**pow2 = [0] \* len\_of\_nums**

**pow2[0] = 1**

**mod = 10 \*\* 9 + 7**

**for i in range(1, len\_of\_nums):**

**pow2[i] = (pow2[i-1] \* 2) % mod**

**subseq\_sum = 0**

**for i in range(len\_of\_nums):**

**subseq\_sum = (subseq\_sum + (pow2[i] - pow2[len\_of\_nums - i - 1]) \* nums[i]) % mod**

**return subseq\_sum**

**Minimum Absolute Sum Difference**

**import bisect**

**class Solution:**

**def minAbsoluteSumDiff(self, nums1, nums2) -> int:**

**sum\_of\_diff = 0**

**gain = 0**

**#maximize the difference between abs(nums1[i] - nums2[i]) and abs(other nums1[i] - nums2[i])**

**sorted\_nums1 = sorted(nums1)**

**for i in range(len(nums1)):**

**diff = abs(nums1[i] - nums2[i])**

**sum\_of\_diff += diff**

**if gain > diff:**

**continue**

**j = bisect.bisect\_right(sorted\_nums1, nums2[i])**

**if j == len(nums1):**

**j -= 1**

**gain = max(gain, diff - abs(sorted\_nums1[j] - nums2[i]))**

**gain = max(gain, diff - abs(sorted\_nums1[j-1] - nums2[i]))**

**return (sum\_of\_diff - gain) % (10 \*\* 9 + 7)**

**Number of Different Subsequences GCDs**

from collections import defaultdict

class Solution:

def countDifferentSubsequenceGCDs(self, nums) -> int:

if not nums:

return 0

def gcd(a, b):

if b == 0:

return a

return gcd(b, a % b)

#fact = [0] \* 200001

fact = defaultdict(int)

for i in range(len(nums)):

for k in range(1, int(nums[i] \*\* 0.5) + 1):

if nums[i] % k == 0:

fact1 = k

fact2 = nums[i] // fact1

fact[fact1] = gcd(fact[fact1], nums[i])

fact[fact2] = gcd(fact[fact2], nums[i])

ans = 0

for f,v in fact.items():

if f == v:

ans += 1

return ans

**Longest Palindromic Substring**

**class Solution:**

**def longestPalindrome(self, s: str) -> str:**

**if not s:**

**return s**

**def expand\_around\_center(l, r):**

**while l >= 0 and r < len(s) and s[l] == s[r]:**

**l -= 1**

**r += 1**

**return r - l - 1**

**start = 0**

**end = 0**

**for i in range(len(s)):**

**len1 = expand\_around\_center(i, i) # char is center**

**len2 = expand\_around\_center(i, i + 1)**

**max\_len = max(len1, len2)**

**if max\_len > end - start:**

**start = i - (max\_len-1) // 2**

**end = i + max\_len // 2**

**return s[start:end + 1]**

**Remove Duplicate Letters**

**from collections import Counter**

**class Solution:**

**def removeDuplicateLetters(self, s):**

**if not s:**

**return s**

**cnt = Counter(s)**

**used = set()**

**stack = []**

**for i in range(len(s)):**

**cnt[s[i]] -= 1**

**if s[i] in used:**

**continue**

**while stack and s[i] < stack[-1] and cnt[stack[-1]] > 0:**

**used.discard(stack[-1]) # to allow to add at second time**

**stack.pop()**

**stack.append(s[i])**

**used.add(s[i])**

**return ‘’.join(stack)**

**Count Unique Characters of All Substrings of a Given String**

**class Solution:**

**def uniqueLetterString(self, s: str) -> int:**

**if not s:**

**return 0**

**#last pos of char**

**last\_pos = [0] \* 27**

**# number of substr ending with char**

**contrib = [0] \* 27**

**uniq\_char = 0**

**cur = 0**

**for i,ch in enumerate(s):**

**code = ord(ch) - ord('A')**

**#subtract previous value**

**cur -= contrib[code]**

**num\_of\_substr\_ending\_here = i+1**

**#if there is else such char then don't count num of uniq chars ending at last\_pos**

**contrib[code] = num\_of\_substr\_ending\_here - last\_pos[code]**

**cur += contrib[code]**

**uniq\_char += cur**

**last\_pos[code] = i + 1**

**return uniq\_char**

**Shortest Subarray with Sum at Least K**

**class Solution:**

**def shortestSubarray(self, nums: List[int], k: int) -> int:**

**if not nums:**

**return -1**

**min\_len = math.inf**

**pr = [0]**

**for n in nums:**

**pr.append(pr[-1] + n)**

**q = []**

**for i in range(len(pr)):**

**while q and pr[i] <= pr[q[-1]]:**

**q.pop()**

**while q and pr[i] - pr[q[0]] >= k:**

**min\_len = min(min\_len, i - q[0])**

**q.pop(0)**

**q.append(i)**

**return min\_len if min\_len != math.inf else -1**

**Sum of Subarray Minimums**

**class Solution:**

**def sumSubarrayMins(self, nums: List[int]) -> int:**

**if not nums or len(nums) == 0:**

**return 0**

**#count arr with length 1**

**mod = 10 \*\* 9 + 7**

**n = len(nums)**

**#place prev least item**

**left = [i+1 for i in range(n)]**

**#place next least item**

**right = [n-i for i in range(n)]**

**st = []**

**for i in range(n):**

**while st and nums[st[-1]] > nums[i]:**

**st.pop()**

**if st:**

**left[i] = i - st[-1]**

**else:**

**left[i] = i + 1**

**st.append(i)**

**st = []**

**for i in range(n):**

**while st and nums[st[-1]] > nums[i]:**

**j = st.pop()**

**right[j] = i - j**

**st.append(i)**

**sum\_of\_min = 0**

**for i in range(n):**

**sum\_of\_min = (sum\_of\_min + nums[i] \* left[i] \* right[i]) % mod**

**return sum\_of\_min**

**Remove K Digits**

**class Solution:**

**def removeKdigits(self, num: str, k: int) -> str:**

**if len(num) <= k:**

**return '0'**

**res = []**

**for i in range(len(num)):**

**while res and res[-1] > num[i] and k > 0:**

**res.pop()**

**k -= 1**

**res.append(num[i])**

**#remove leading 0**

**i = 0**

**while i < len(res) and res[0] == '0':**

**res.pop(0)**

**res = res[:-k] if k else res**

**return ''.join(res) if res else '0'**

**. Max Chunks To Make Sorted II**

**class Solution:**

**def maxChunksToSorted(self, arr) -> int:**

**if not arr:**

**return 0**

**if len(arr) == 1:**

**return 1**

**left = [0] \* len(arr)**

**right = [0] \* len(arr)**

**left[0] = arr[0]**

**for i in range(1, len(arr)):**

**left[i] = max(left[i-1],arr[i])**

**right[-1] = arr[-1]**

**for i in range(len(arr) - 2, -1,-1):**

**right[i]=min(right[i+1],arr[i])**

**chunks = 0**

**for i in range(len(arr)-1):**

**if left[i] <= right[i+1]:**

**chunks += 1**

**return chunks+1**

**Minimum Sideway Jumps**

**class Solution:**

**def minSideJumps(self, obstacles):**

**jump = 0**

**i = 0**

**cur = 2**

**while i + 1 < len(obstacles):**

**if obstacles[i + 1] == cur:**

**jump += 1**

**obs = []**

**while i < len(obstacles):**

**obs.append(obstacles[i])**

**#if we have all 3 lanes, then we will found farest lane.**

**if 1 in obs and 2 in obs and 3 in obs:**

**break**

**i += 1**

**#farest lane**

**cur = obs[-1]**

**i -= 2**

**i += 1**

**return jump**

**Finding MK Average**

import java.util.Iterator;

import java.util.LinkedList;

import java.util.Queue;

import java.util.TreeMap;

public class MKAverage {

private int m,k;

private Queue<Integer> nums = new LinkedList<>();

private TreeMap<Integer,Integer> tree = new TreeMap<>();

private int sum\_of\_items;

private int result\_sum;

public MKAverage(int m, int k) {

this.m = m;

this.k = k;

}

private int calcSum(Iterator<Integer> keys, int k){

int result = 0;

for(;keys.hasNext() && k > 0;){

Integer key = keys.next();

Integer freq = tree.get(key);

if(k - freq >= 0) {

result += key \* freq;

} else {

result += key \* k;

}

k -= freq;

}

return result;

}

public void addElement(int num) {

this.nums.add(num);

sum\_of\_items += num;

Integer freq = tree.getOrDefault(num, 0);

tree.put(num, freq + 1);

if(nums.size() > m) {

int removed = nums.remove();

sum\_of\_items -= removed;

freq = tree.get(removed);

freq -= 1;

if(freq > 0)

tree.put(removed, freq);

else

tree.remove(removed);

}

if(nums.size() == m) {

int sum\_to\_remove = calcSum(tree.keySet().iterator(), k) + calcSum(tree.descendingKeySet().iterator(), k);

result\_sum = sum\_of\_items - sum\_to\_remove;

}

}

public int calculateMKAverage() {

if(nums.size() < m)

return -1;

return result\_sum / (nums.size() - 2 \* k);

}

}

**Create Maximum Number**

**class Solution(object):**

**def maxNumber(self, nums1,nums2, k):**

**n = len(nums1)**

**m = len(nums2)**

**res = [0] \* k**

**for l1 in range(k+1):**

**l2 = k - l1**

**if l1 > n or l2 > m:**

**continue**

**left = self.findMaxNumber(nums1, l1)**

**right = self.findMaxNumber(nums2, l2)**

**merged = self.merge(left, right)**

**if merged > res:**

**res = merged**

**return res**

**def findMaxNumber(self, nums, need\_len):**

**n = len(nums)**

**if need\_len > n:**

**return [-1]**

**res = [-1]**

**while need\_len > 0:**

**l = res[-1] + 1**

**r = n - need\_len + 1**

**res.append(max(range(l,r),key=nums.\_\_getitem\_\_))**

**need\_len -= 1**

**return [nums[i] for i in res[1:]]**

**def merge(self, n1, n2):**

**n = len(n1)**

**m = len(n2)**

**res = []**

**#important to compare whole numbers not separete numbers!**

**while n1 or n2:**

**if n1 > n2:**

**res.append(n1[0])**

**n1 = n1[1:]**

**else:**

**res.append(n2[0])**

**n2 = n2[1:]**

**return res**

**Single Thread CPU**

from heapq import heappop,heappush

class Solution:

def getOrder(self, tasks):

sorted\_tasks = [[i, tasks[i][0], tasks[i][1]] for i in range(len(tasks))]

sorted\_tasks.sort(key=lambda x: x[1])

next\_task = []

res = []

time = 0

i = 0

while len(res) < len(tasks):

#put new task

while i < len(tasks) and sorted\_tasks[i][1] <= time:

heappush(next\_task, (sorted\_tasks[i][2], sorted\_tasks[i][0]))

i += 1

if not next\_task:

time = sorted\_tasks[0][1]

continue

\_,j = heappop(next\_task)

res.append(j)

time += tasks[j][1]

return res

**Find XOR Sum of All Pairs Bitwise AND**

1. Use & as \*
2. Use xor as +,
3. (a+b)\*(c+d)

class Solution:

def getXORSum(self, arr1: List[int], arr2: List[int])

res = 0

xor1 = 0

for i in range(len(arr1)):

xor1 ^= arr1[i]

xor2 = 0

for i in range(len(arr2)):

xor2 ^= arr2[i]

return xor1 & xor2

**Minimum Cost Tree From Leaf Values**

1. **Calculate cost to remove min items from arr:**

**arr[i] \* min(max left of arr[i], max right of arr[i])**

**class Solution:**

**def mctFromLeafValues(self, arr: List[int]) -> int:**

**st = [math.inf]**

**res = 0**

**for i in range(len(arr)):**

**while st and st[-1] <= arr[i]:**

**cur = st.pop()**

**res += cur \* min(arr[i],st[-1])**

**st.append(arr[i])**

**for i in range(2, len(st)):**

**res += st[i-1] \* st[i]**

**return res**

**Increasing Triplet Subsequence**

**class Solution:**

**def increasingTriplet(self, nums):**

**if not nums or len(nums) < 3:**

**return False**

**# find a < b < c**

**a = inf**

**b = inf**

**for i in range(len(nums)):**

**if nums[i] <= a:**

**a = nums[i]**

**elif nums[i] < b:**

**b = nums[i]**

**elif a < b < nums[i]:**

**return True**

**return False**

**Score of Parentheses**

**class Solution(object):**

**def scoreOfParentheses(self, S):**

**st = [0]**

**for ch in S:**

**if ch == '(':**

**st.append(0)**

**else:**

**v = st.pop()**

**st[-1] += max(2\*v, 1)**

**return st.pop()**

**Sliding Window Maximum**

**class Solution:**

**def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:**

**if not nums or k <= 1:**

**return nums**

**l = [0] \* len(nums)**

**r = [0] \* len(nums)**

**l[0] = nums[0]**

**for i in range(1, len(nums)):**

**if i % k == 0:**

**l[i] = nums[i]**

**else:**

**l[i] = max(nums[i],l[i-1])**

**r[-1] = nums[-1]**

**for i in range(len(nums) - 2, -1,-1):**

**if i % k == 0:**

**r[i] = nums[i]**

**else:**

**r[i] = max(nums[i],r[i+1])**

**res = [0] \* (len(nums) - k + 1)**

**for i in range(len(nums) - k + 1):**

**res[i] = max(l[i+k-1],r[i])**

**return res**

**Number of Visible People in a Queue**

**class Solution:**

**def canSeePersonsCount(self, heights: List[int]) -> List[int]:**

**n = len(heights)**

**res = [0] \* n**

**st = []**

**for i in range(n-1, -1, -1):**

**while st and st[-1] < heights[i]:**

**st.pop()**

**res[i] +=1**

**if st:**

**res[i] += 1**

**st.append(heights[i])**

**return res**

**132 Pattern**

**class Solution:**

**def find132pattern(self, nums):**

**if not nums:**

**return False**

**left = [0] \* len(nums)**

**left[0] = nums[0]**

**for i in range(1, len(nums)):**

**left[i] = min(left[i-1],nums[i])**

**st = []**

**for i in range(len(nums)-1, -1,-1):**

**if i and nums[i] <= left[i]:**

**continue**

**# keep increasing subsequence**

**while st and nums[i] > nums[st[-1]]:**

**k = st.pop()**

**if left[i] < nums[k]:**

**return True**

**st.append(i)**

**return False**

**Create Maximum Number**

**class Solution(object):**

**def maxNumber(self, nums1,nums2, k):**

**n = len(nums1)**

**m = len(nums2)**

**res = [0] \* k**

**for l1 in range(k+1):**

**l2 = k - l1**

**if l1 > n or l2 > m:**

**continue**

**left = self.findMaxNumber(nums1, l1)**

**right = self.findMaxNumber(nums2, l2)**

**merged = self.merge(left, right)**

**if merged > res:**

**res = merged**

**return res**

**def findMaxNumber(self, nums, need\_len):**

**n = len(nums)**

**if need\_len > n:**

**return [-1]**

**res = [-1]**

**while need\_len > 0:**

**l = res[-1] + 1**

**r = n - need\_len + 1**

**res.append(max(range(l,r),key=nums.\_\_getitem\_\_))**

**need\_len -= 1**

**return [nums[i] for i in res[1:]]**

**def merge(self, n1, n2):**

**n = len(n1)**

**m = len(n2)**

**res = []**

**#important to compare whole numbers not separete numbers!**

**while n1 or n2:**

**if n1 > n2:**

**res.append(n1[0])**

**n1 = n1[1:]**

**else:**

**res.append(n2[0])**

**n2 = n2[1:]**

**return res**

**Describe the Painting**

**class Solution:**

**def splitPainting(self, segments):**

**#sum offset**

**sums = defaultdict(int)**

**# point where segment is starting/ending**

**end = defaultdict(int)**

**for s,e,c in segments:**

**sums[s] += c**

**sums[e] -= c**

**end[s] = True**

**end[e] = True**

**res = []**

**r = max(sums)**

**cur\_sum = 0**

**last\_i = 0**

**for i in range(1, r + 1):**

**if cur\_sum > 0 and end[i] > 0:**

**res.append([last\_i,i,cur\_sum])**

**if end[i] > 0:**

**last\_i = i**

**cur\_sum += sums[i]**

**return res**

**Determine Whether Matrix Can Be Obtained By Rotation**

**class Solution:**

**def findRotation(self, mat: List[List[int]], target: List[List[int]]) -> bool:**

**def rotate(mat):**

**rev\_matrix = []**

**for row in zip(\*mat[::-1]):**

**rev\_matrix.append(list(row))**

**return rev\_matrix**

**def mat\_equal(mat1, mat2):**

**for row1,row2 in zip(rotated\_mat, target):**

**if row1 != row2:**

**return False**

**return True**

**rotated\_mat = rotate(mat)**

**if mat\_equal(rotated\_mat,mat):**

**return True**

**for i in range(3):**

**rotated\_mat = rotate(rotated\_mat)**

**if mat\_equal(rotated\_mat, rotated\_mat):**

**return True**

**return False**

**Frequency of the Most Frequent Element**

**class Solution:**

**def maxFrequency(self, nums: List[int], k: int) -> int:**

**nums.sort()**

**cur\_sum = k**

**l = 0**

**for i in range(len(nums)):**

**cur\_sum += nums[i]**

**if cur\_sum < nums[i]\* (i - l + 1):**

**cur\_sum -= nums[l]**

**l += 1**

**return i - l + 1**

**Maximum Building Height**

**class Solution:**

**def maxBuilding(self, n: int, restrictions: List[List[int]]) -> int:**

**if not restrictions:**

**return n - 1**

**restrictions.extend([[1,0],[n,n-1]])**

**restrictions.sort(key=lambda x:x[0])**

**def calc\_height(lst):**

**max\_height = 0**

**for i in range(1, len(lst)):**

**h1 = lst[i-1][1]**

**h2 = lst[i][1]**

**# calculate second point(lst[i][0], h) that starts at (lst[i-1][0],h1)**

**h = h1 + abs(lst[i][0] - lst[i-1][0])**

**if h > h2:**

**h = h2 + (h-h2)//2**

**max\_height = max(max\_height, h)**

**lst[i][1] = min(h,h2)**

**return max\_height**

**#go from left to right**

**calc\_height(restrictions)**

**#go from right to left**

**return calc\_height(restrictions[::-1])**

**Maximum Product of the Length of Two Palindromic Substrings**

class Solution:

def maxProduct(self, s):

def manachers(S):

A = '@#' + '#'.join(S) + '#$'

Z = [0] \* len(A)

center = 0

right = 0

for i in range(1, len(A) - 1):

if i < right:

Z[i] = min(right - i, Z[2 \* center - i])

while A[i + Z[i] + 1] == A[i - Z[i] - 1]:

Z[i] += 1

if i + Z[i] > right:

center = i

right = i + Z[i]

return Z[2:-2:2]

def helper(s):

man = manachers(s)

n = len(s)

intervals = [(i - man[i] // 2, i + man[i] // 2) for i in range(n)]

arr = [0] \* n

for a, b in intervals:

arr[b] = max(arr[b], b - a + 1)

for i in range(n - 2, -1, -1):

arr[i] = max(arr[i], arr[i + 1] - 2)

max\_int = [0] \* len(arr)

max\_int[0] = arr[0]

for i in range(1, len(arr)):

max\_int[i] = max(max\_int[i - 1], arr[i])

return max\_int

t1 = helper(s)

t2 = helper(s[::-1])[::-1]

t2 = t2[1:] + [0]

max\_prod = 0

for i in range(len(t1)):

max\_prod = max(max\_prod, t1[i]\*t2[i])

return max\_prod

**Minimum Total Space Wasted With K Resizing Operations**

from math import inf

class Solution:

def minSpaceWastedKResizing(self, nums, k: int) -> int:

n = len(nums)

dp = [[-1] \* (k+1) for \_ in range(n)]

def find\_min\_wasted(dp, i, k):

if i == len(nums):

return 0

if k < 0:

return inf

if dp[i][k] >= 0:

return dp[i][k]

sum\_of\_num = 0

max\_so\_far = nums[i]

ans = inf

for j in range(i, len(nums)):

max\_so\_far = max(max\_so\_far, nums[j])

sum\_of\_num += nums[j]

#try to resize the at max\_so\_far

wasted = (j - i + 1) \* max\_so\_far - sum\_of\_num

ans = min(ans, find\_min\_wasted(dp, j + 1, k - 1) + wasted)

dp[i][k] = ans

return ans

return find\_min\_wasted(dp, 0, k)

**Regular Expression Matching**

class Solution:

def isMatch(self, s: str, p: str) -> bool:

if not p:

return not s

first\_match = s and (s[0] == p[0] or p[0] == '.')

if len(p) >= 2 and p[1] == '\*':

return self.isMatch(s,p[2:]) or first\_match and self.isMatch(s[1:], p)

else:

return first\_match and self.isMatch(s[1:],p[1:])

**Merge k Sorted Lists**

**class Solution:**

**def mergeKLists(self, lists):**

**min\_heap = []**

**for lst in lists:**

**head = lst**

**while head:**

**heappush(min\_heap, head.val)**

**head = head.next**

**head = ListNode(-1)**

**node = head**

**while min\_heap:**

**a = heappop(min\_heap)**

**node.next = ListNode(a)**

**node = node.next**

**return head.next**

**Combinations**

**class Solution(object):**

**def combine(self, n, k):**

**def comb(last\_num, buf, cur\_comb):**

**if len(cur\_comb) == k:**

**buf.append(cur\_comb[::])**

**return**

**for a in range(last\_num, n + 1):**

**cur\_comb.append(a)**

**comb(a + 1, buf, cur\_comb)**

**cur\_comb.pop()**

**buf = []**

**comb(1, buf, [])**

**return buf**

**Maximum Matrix Sum**

**class Solution:**

**def maxMatrixSum(self, matrix: List[List[int]]) -> int:**

**neg\_cnt = 0**

**max\_sum = 0**

**min\_num = abs(matrix[0][0])**

**for i in range(len(matrix)):**

**for j in range(len(matrix[0])):**

**max\_sum += abs(matrix[i][j])**

**min\_num = min(min\_num, abs(matrix[i][j]))**

**if matrix[i][j] < 0:**

**neg\_cnt += 1**

**if neg\_cnt % 2 == 0:**

**return max\_sum**

**return max\_sum - 2 \* min\_num**

**Number of Ways to Arrive at Destination**

**class Solution:**

**def countPaths(self, n: int, roads: List[List[int]]) -> int:**

**adj\_lst = defaultdict(list)**

**for u,v,t in roads:**

**adj\_lst[u].append([v,t])**

**adj\_lst[v].append([u,t])**

**dist = [inf] \* n**

**dist[0] = 0**

**path = [0] \* n**

**path[0] = 1**

**q = [[0,0]]**

**mod = 10 \*\* 9 + 7**

**while q:**

**d,u = heappop(q)**

**if dist[u] < d:**

**continue**

**for nei,t in adj\_lst[u]:**

**if dist[nei] > d+ t:**

**path[nei] = path[u]**

**dist[nei] = d + t**

**heappush(q, [dist[nei], nei])**

**elif dist[nei] == d + t:**

**path[nei] = (path[nei] + path[u]) % mod**

**return path[n-1]**

**Best Time to Buy and Sell Stock III**

**class Solution:**

**def maxProfit(self, prices: List[int]) -> int:**

**buy\_first\_stock = prices[0]**

**sell\_first\_stock = 0**

**buy\_second\_stock = -prices[0]**

**sell\_second\_stock = 0**

**for price in prices:**

**buy\_first\_stock = min(buy\_first\_stock,price)**

**sell\_first\_stock = max(sell\_first\_stock, price - buy\_first\_stock)**

**#pay price min price for second stock**

**buy\_second\_stock = max(buy\_second\_stock, sell\_first\_stock - price)**

**sell\_second\_stock = max(sell\_second\_stock,buy\_second\_stock + price )**

**return sell\_second\_stock**

**Longest Consecutive Sequence**

**class Solution:**

**def longestConsecutive(self, nums: List[int]) -> int:**

**if not nums:**

**return 0**

**seen\_nums = set(nums)**

**max\_len = 0**

**for num in nums:**

**# make search only for first item**

**if num - 1 not in seen\_nums:**

**cur\_len = 1**

**next\_num = num**

**while next\_num + 1 in seen\_nums:**

**cur\_len += 1**

**next\_num += 1**

**max\_len = max(max\_len, cur\_len)**

**return max\_len**

**Copy List with Random Pointer**

class Solution:

def copyRandomList(self, head: 'Node') -> 'Node':

if not head:

return head

new\_head = None

ptr = head

# create new list A->A'->B->B'...

while ptr:

ptr\_next = ptr.next

ptr.next = Node(ptr.val,ptr\_next)

if not new\_head:

new\_head = ptr.next

ptr = ptr\_next

# set up random pointers for new list nodes

ptr = head

while ptr:

if ptr.random:

ptr.next.random = ptr.random.next

ptr = ptr.next.next

#reset intertwined lists

ptr = head

while ptr:

ptr\_next = ptr.next

if ptr.next:

ptr.next = ptr.next.next

ptr = ptr\_next

return new\_head

**Majority Element**

**class Solution:**

**def majorityElement(self, nums: List[int]) -> int:**

**cand = None**

**cnt = 0**

**for num in nums:**

**if cnt == 0:**

**cand = num**

**cnt += (1 if cand == num else -1)**

**return cand**

**Majority Element II**

**class Solution:**

**def majorityElement(self, nums: List[int]) -> List[int]:**

**# assert self.\_majorityElement([1,2,1]) == [1], 'test0'**

**# assert self.\_majorityElement([]) == [], 'test1'**

**# assert self.\_majorityElement([1,1,2,2]) == [1,2], 'test2'**

**return self.\_majorityElement(nums)**

**def \_majorityElement(self, nums: List[int]) -> List[int]:**

**if not nums:**

**return []**

**a = None**

**cnt1 = 0**

**b = None**

**cnt2 = 0**

**for num in nums:**

**if a == num:**

**cnt1 += 1**

**elif b == num:**

**cnt2 += 1**

**elif cnt1 == 0:**

**a = num**

**cnt1 += 1**

**elif cnt2 == 0:**

**b = num**

**cnt2 += 1**

**else:**

**cnt1 -= 1**

**cnt2 -= 1**

**res = []**

**for c in [a,b]:**

**if nums.count(c) > len(nums) //3:**

**res.append(c)**

**return res**

**Minimum Adjacent Swaps to Reach the Kth Smallest Number**

**lass Solution:**

**def getMinSwaps(self, num: str, k: int) -> int:**

**# assert self.\_getMinSwaps('12',1) == 1,'test0' #21**

**# assert self.\_getMinSwaps('2133',2) == 2,'test1' #3213**

**# assert self.\_getMinSwaps('099',2) == 2,'test2' #990**

**# assert self.\_getMinSwaps('6975', 8) == 4,'test3'**

**return self.\_getMinSwaps(num, k)**

**def \_getMinSwaps(self, num, k):**

**def reverse(cur\_num, s):**

**e = len(cur\_num) - 1**

**while s < e:**

**cur\_num[s],cur\_num[e] = cur\_num[e],cur\_num[s]**

**s += 1**

**e -= 1**

**def find\_next(cur\_num):**

**for i in range(len(cur\_num) - 1, 0, -1):**

**if cur\_num[i] > cur\_num[i-1]:**

**reverse(cur\_num, i)**

**for j in range(i, len(cur\_num)):**

**if cur\_num[j] > cur\_num[i-1]:**

**cur\_num[j],cur\_num[i-1] = cur\_num[i-1],cur\_num[j]**

**return**

**reverse(cur\_num, 0)**

**kth\_num = list(num)**

**for \_ in range(k):**

**find\_next(kth\_num)**

**swaps = 0**

**num = list(num)**

**for i in range(len(kth\_num)):**

**j = i**

**while kth\_num[i] != num[j]:**

**j += 1**

**while i < j:**

**num[j-1],num[j] = num[j],num[j-1]**

**swaps += 1**

**j -= 1**

**return swaps**

**Course Schedule II**

**def \_findOrder(self, n , pre):**

**if n == 0:**

**return []**

**adj\_lst = defaultdict(list)**

**ind = [0] \* n**

**for d,s in pre:**

**adj\_lst[s].append(d)**

**ind[d] += 1**

**q = []**

**for i in range(len(ind)):**

**if ind[i] == 0:**

**q.append(i)**

**vert = []**

**while q:**

**v = q.pop(0)**

**vert.append(v)**

**for nei in adj\_lst[v]:**

**ind[nei] -= 1**

**if ind[nei] == 0:**

**q.append(nei)**

**return vert if len(vert) == n else []**

**Add Two Numbers II**

**Class Solution:**

**def len\_list(self, l):**

**if not l:**

**return 0**

**c = 0**

**while l:**

**c+=1**

**l = l.next**

**return c**

**def add\_zero\_nodes(self, l, d):**

**while d > 0:**

**z = ListNode(0)**

**z.next = l**

**l = z**

**d -= 1**

**return l**

**def addTwoNumbers(self, l1, l2):**

**if not l1 or not l2:**

**return None**

**carry = 0**

**res = None**

**len1 = self.len\_list(l1)**

**len2 = self.len\_list(l2)**

**if len1 > len2:**

**l2 = self.add\_zero\_nodes(l2, len1 - len2)**

**elif len2 > len1:**

**l1 = self.add\_zero\_nodes(l1, len2 - len1)**

**def sum\_list(p1, p2):**

**nonlocal carry, res**

**if not p1:**

**return**

**sum\_list(p1.next, p2.next)**

**a = p1.val + p2.val + carry**

**carry = a // 10**

**a = a % 10**

**node = ListNode(a)**

**if res:**

**node.next = res**

**res = node**

**sum\_list(l1, l2)**

**if carry:**

**node = ListNode(carry)**

**node.next = res**

**res = node**

**return res**

**Contiguous Array**

**class Solution(object):**

**def findMaxLength(self, nums):**

**if not nums:**

**return 0**

**m = {0: -1}**

**count = 0**

**max\_len = 0**

**for i in range(len(nums)):**

**if nums[i] == 0:**

**count += -1**

**else:**

**count += 1**

**if count in m:**

**max\_len = max(max\_len, i - m[count])**

**else:**

**m[count] = i**

**return max\_len**

**Maximum Subarray Min-Product**

**class Solution:**

**def maxSumMinProduct(self, nums: List[int]) -> int:**

**# assert self.\_maxSumMinProduct([1,2,2]) == 8,'test1'**

**# assert self.\_maxSumMinProduct([1,1,1]) == 3,'test2'**

**# assert self.\_maxSumMinProduct([1,2,3]) == 10,'test3'**

**return self.\_maxSumMinProduct(nums)**

**def \_maxSumMinProduct(self, nums):**

**MOD = 10 \*\*9 + 7**

**#keep up increasing order to find closest left min item**

**pr = [0]**

**for i in range(len(nums)):**

**pr.append(pr[-1] + nums[i])**

**st = []**

**max\_prod = 0**

**for i in range(len(nums) + 1):**

**while st and (i == len(nums) or nums[st[-1]] > nums[i]):**

**#leftmos min item**

**# subarray from [st.pop+1: i)**

**#min item is at [j]**

**#if st is empty we take pr[st.pop+1] - st[0]**

**j = st.pop()**

**print('{}-{}:{}'.format((st[-1]+1 if st else 0),i, j))**

**max\_prod = max(max\_prod,nums[j]\*(pr[i] - pr[st[-1] + 1 if st else 0]))**

**st.append(i)**

**#we put start and min item for this subarray**

**return max\_prod % MOD**

**Subtree of Another Tree**

**class Solution:**

**def isSubtree(self, s, t):**

**def find\_subtree(root, node):**

**if not root:**

**return False**

**return root.val == node.val and is\_subtree(root, node) or find\_subtree(root.left, node) or find\_subtree(**

**root.right, node)**

**def is\_subtree(root, node):**

**if not root and not node:**

**return True**

**if not root or not node or root.val != node.val:**

**return False**

**return is\_subtree(root.left, node.left) and is\_subtree(root.right, node.right)**

**return find\_subtree(s, t)**

**Minimum Number of Swaps to Make the Binary String Alternating**

**def \_minSwaps(self, s: str) -> int:**

**if not s:**

**return 0**

**zeros = 0**

**s\_len = len(s)**

**for ch in s:**

**if ch == '0':**

**zeros += 1**

**if s\_len % 2 == 0 and zeros\*2 != s\_len or abs(s\_len - 2 \* zeros) > 1:**

**return -1**

**def count\_swaps(s, start):**

**swaps = 0**

**for ch in s:**

**if ch != start:**

**swaps += 1**

**start = str(ord('1') - ord(start))**

**return swaps**

**if s\_len % 2 == 0:**

**return min(count\_swaps(s, '0'),count\_swaps(s, '1'))//2**

**if zeros > s\_len // 2:**

**return count\_swaps(s, '0')//2**

**return count\_swaps(s, ‘1')//2**

**Erect the Fence**

**class Solution:**

**def outerTrees(self, trees: List[List[int]]) -> List[List[int]]:**

**#assert self.\_outerTrees([]) == [],'empty trees'**

**#assert self.\_outerTrees([[1,1],[2,2],[3,1]]) == [[1,1],[2,2],[3,1]],'empty trees'**

**return self.\_outerTrees(trees)**

**def \_outerTrees(self, trees):**

**if len(trees) < 4:**

**return trees**

**def orientation(p, q, r):**

**return (q[1] - p[1])\*(r[0] - q[0]) - (q[0] - p[0]) \* (r[1] - q[1])**

**def is\_between(p, i, q):**

**a = i[0] >= p[0] and i[0] <= q[0] or i[0] <= p[0] and i[0] >= q[0]**

**b = i[1] >= p[1] and i[1] <= q[1] or i[1] <= p[1] and i[1] >= q[1]**

**return a and b**

**left = 0**

**for i in range(len(trees)):**

**if trees[i][0] < trees[left][0]:**

**left = i**

**p = left**

**q = 0**

**hull = set()**

**while True:**

**q = (p + 1) % len(trees)**

**for i in range(len(trees)):**

**if orientation(trees[p], trees[i], trees[q]) < 0:**

**q = i**

**for i in range(len(trees)):**

**if i != p and i != q and orientation(trees[p],trees[i], trees[q]) == 0 and is\_between(trees[p],trees[i],trees[q]):**

**hull.add((trees[i][0],trees[i][1]))**

**hull.add((trees[q][0],trees[q][1]))**

**p = q**

**if p == left:**

**break**

**return hull**

**Sum of All Subset XOR Totals**

**class Solution:**

**def subsetXORSum(self, nums) -> int:**

**if not nums:**

**return 0**

**#to count the bits which are 1**

**ones = 0**

**for a in nums:**

**ones = ones | a**

**#there 2ˆ(n-1) subsets at which it set bit to 1**

**xor = 0**

**i = 0**

**while ones:**

**if ones & 1:**

**xor += 2\*\*(len(nums) - 1 + i)**

**ones = ones >> 1**

**i += 1**

**return xor**

**Find K Closest Elements**

**1.Consider two ranges arr[mid]:arr[mid+k-1] and arr[mid+1]:arr[mid+k] as common range arr[mid]:arr[mid+k]**

**By using binary search.**

**class Solution:**

**def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:**

**assert self.\_findClosestElements([1,2,3],3,1) == [1,2,3], 'ascending array'**

**assert self.\_findClosestElements([],3,1) == [],'empty array'**

**assert self.\_findClosestElements([1,2,3],3,5) == [1,2,3],'x is max value'**

**assert self.\_findClosestElements([10,12,13],3,5) == [10,12,13],'x is min value'**

**assert self.\_findClosestElements([1,2,3,4,5],4,-1) == [1,2,3,4],'x is negative'**

**return self.\_findClosestElements(arr, k, x)**

**def \_findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:**

**if not arr or len(arr) <= k:**

**return arr**

**def find\_closest(arr, k, x):**

**s = 0**

**e = len(arr) - k**

**while s < e:**

**m = s + (e - s) // 2**

**r = m + k**

**# compare x with (arr[m]+arr[m+k])//2**

**if (arr[m] + arr[m + k])//2 < x :**

**s = m + 1**

**#if x - arr[m] > arr[r] - x:**

**# go to right**

**# s = m + 1**

**else:**

**e = m**

**return s**

**#find low boundey for k range**

**i = find\_closest(arr, k, x)**

**return arr[i:i+k]**