From bisect import bisect\_left

Bisect\_left(列表，待查值）返回第一个比待查值大的数的索引

Set1.update(set2)把set2的元素加入set1（会自动去重）

new\_set=set1.union(set2)取set1，set2的并集且原集不变

def subsets(nums):

result = []

def backtrack(start, path):

result.append(path.copy()) # 保存当前子集

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

path.append(nums[i]) # 选择元素

backtrack(i + 1, path) # 递归

path.pop() # 回溯

backtrack(0, [])return result

**在非类的函数（即普通函数或全局函数）中，递归调用自身时不需要添加任何前缀**

**类中的函数调用时需要写self.函数名**

也可以用current=current.left加上while的方法遍历

且和None的联动使用is

Defaultdict用法

from collections import defaultdict

class Solution:

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

n = len(nums)

total = n \* (n - 1) // 2

diff\_map = defaultdict(int)（默认值是0）

good = 0

for j in range(n):

current\_diff = nums[j] - j

# 直接访问，利用defaultdict的自动初始化特性

good += diff\_map[current\_diff]

diff\_map[current\_diff] += 1

return total - good

j - i != nums[j] - nums[i]

转化为j-nums[j]!=i-nums[i]，用这样的形式处理储存哈希表好

Str.isdigits()是否只含有数字

.isalpha()是否只有字母.Isalnum是否只有数字字母

Islower(),isupper,istitle,lower,upper

Startswith,endswith

If str1.stratwith(str2):

此函数用于判断一个字符串是否是另一个的前缀

List(xx)

Set(xx)可以直接列表和集合之间互化

字典中用.get(键,无索引时返回的默认值)

来避免产生keyerror

Math.comb(n,k) 生成组合数Cnk

Math.perm(n,k) 生成排列数Ank

Itertools.combinations//permutation为排列数

Combos=list(combinations(elements,k))从elemnets数组中选k个用于组合

round（number,位数）四舍五入精确至第几位

卡特兰数Cn=（2n）!/((n+1)! \* n!)

C0：1，c1：1，c2：2，5，14，42

n对括号的有效排列数目

n个元素进栈出栈的合法序列数目

n个节点构成的完全/满 二叉树数目

n个节点构成的二叉搜索树数目

将一个n+2边形用n-1条对角线化为n个三角形

由（0，0）到（2n，0）每次动（1，1）或（1，-1），且不会穿x轴的路径总数

n个表达式构成的合法表达式树数目

当需要列举时，可分别往左往右扫一遍，得到前缀和和后缀和。那么某一处的值即为prefix[i]\*nums【i】\*suffix[i+1]

字符串及其反向字符串[::-1]之间最长公共子串为其最长回文子串。

字符串长度-其最长子串得到 插入字符使其编程回文字符串的最少次数。

用临时变量模拟栈，获得得到平衡栈最小交换次数

def minSwaps(self, s: str) -> int:

        p, q = 0, len(s) - 1

        cnt1 = ans = 0

        while p < q:

            cnt1 += 1 if s[p] == '[' else -1

            p += 1

            if cnt1 < 0:

                cnt2 = 0

                while cnt2 >= 0:

                    cnt2 += -1 if s[q] == '[' else 1

                    q -= 1

                ans += 1

                cnt1 = 1  # 模拟交换后的效果

        return ans

分解因书种类数的递归写法

def decode(n,minfactor):

if n==1:

return 1

count=0

for i in range(minfactor,n+1):

if n%i==0:

#递归，只找更大的因数，避免重复

count+=decode(n//i,i)

##重新限制最小因数条件，

##使得生成的因数列是非递减的

return count

n=int(input())

for \_ in range(n):

x=int(input())

print(decode(x,2))

##逐项操作的写法# 计算前缀按位或数组

        prefix = [0] \* n

        for i in range(1, n):

            prefix[i] = prefix[i-1] | nums[i-1]

        # 计算后缀按位或数组

        suffix = [0] \* n

        for i in range(n-2, -1, -1):

            suffix[i] = suffix[i+1] | nums[i+1]

编程是美的，可以猜

k次操作内让数组互相的位或得到的结果最大，应该要把k次乘以2全部安排在一个数上

def countCompleteSubarrays(self, nums: List[int]) -> int:##统计完全子数组个数

        k = len(set(nums))

        cnt = defaultdict(int)  #初始时cnt长度为0

        ans = left = 0

        for x in nums:

            cnt[x] += 1

            while len(cnt) == k:

                out = nums[left]

                cnt[out] -= 1

                if cnt[out] == 0:

                    del cnt[out]##终止条件

                left += 1

            ans += left

        return ans

滑动窗口的例子（无重复字符的最长子串）（不断调整left和right）

def lengthOfLongestSubstring(s):

window = {}

left = max\_len = 0

for right in range(len(s)):

window[s[right]] = window.get(s[right], 0) + 1

# 当窗口内有重复字符时收缩

while window[s[right]] > 1:

window[s[left]] -= 1

left += 1

max\_len = max(max\_len, right - left + 1)

return max\_len

If set:

集合非空

Set名.discard(元素值)，移除元素且不会报错

归并排序求逆序对

Kadane最大子矩阵和

def kadane\_1d(arr):

一维Kadane算法kadane是一样的

max\_end\_here = max\_so\_far = arr[0]

for num in arr[1:]:

max\_end\_here = max(num, max\_end\_here + num)

max\_so\_far = max(max\_so\_far, max\_end\_here)

return max\_so\_far

def max\_submatrix\_sum(matrix):

求二维矩阵中的最大子矩阵和

if not matrix:

return 0

rows = len(matrix)

cols = len(matrix[0])

max\_sum = float('-inf')

for l in range(cols):

for r in range(l, cols):

row\_sums = [0] \* rows

for row in range(rows):

for col in range(l, r + 1):

row\_sums[row] += matrix[row][col]

current\_sum = kadane\_1d(row\_sums)

max\_sum = max(max\_sum, current\_sum)

return max\_sum

Kadane算法求最大连续子序列和

def max\_subarray\_sum(nums):

max\_sum = current\_sum = nums[0]

for num in nums[1:]:

current\_sum = max(num, current\_sum + num)

max\_sum = max(max\_sum, current\_sum)

return max\_sum

def merge\_sort\_and\_count(arr):

if len(arr) <= 1:用归并排序计算逆序对数量；

return arr, 0如果只要归并排序的代码，则其中所有带count的变量都不需要。

mid = len(arr) // 2第一个函数的return的第二项都不用

left, left\_count = merge\_sort\_and\_count(arr[:mid])

right, right\_count = merge\_sort\_and\_count(arr[mid:])

merged, cross\_count = merge\_and\_count(left, right)

return merged, left\_count + right\_count + cross\_count

def merge\_and\_count(left, right):

merged = []

i = j = 0

cross\_count = 0 # 跨左右子数组的逆序对数量

while i < len(left) and j < len(right):

if left[i] <= right[j]:

merged.append(left[i])

i += 1

else:

merged.append(right[j])

j += 1 # 统计逆序对：left[i..end] 都大于 right[j]

cross\_count += len(left) - i

merged.extend(left[i:])

merged.extend(right[j:])

return merged, cross\_count

Kadane算法求最大子序列

最小生成树不需要建树，是在图中找 在带权无向连接图中找一颗包含所有顶点的子树使得边权之和最大或最小

图的算法中往往存入（u，v，w）的元组组成列表

class UnionFind:Kruskal算法，给出了每条边的起终点和权重

def \_\_init\_\_(self, size):（边数和点数差不多时）

self.parent = list(range(size))

def find(self, x):

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x]) # 路径压缩

return self.parent[x]

def union(self, x, y):

fx = self.find(x)

fy = self.find(y)

if fx == fy:

return False

self.parent[fy] = fx

return True

def kruskal(edges, n):

edges.sort(key=lambda x: x[2]) # 按边权排序

uf = UnionFind(n)

mst\_weight = 0

edges\_added = 0

for u, v, w in edges:

if uf.union(u, v):

mst\_weight += w

edges\_added += 1

if edges\_added == n - 1:

break

return mst\_weight if edges\_added == n - 1 else -1

Prim算法 import heapq（给出了各个节点的相邻节点和边权）

def prim(graph, start):（边数远大于点数）

n = len(graph)

visited = [False] \* n

mst\_weight = 0

edges = 0

min\_heap = []

# 从start开始，初始边权为0

heapq.heappush(min\_heap, (0, start))

while min\_heap and edges < n - 1:

weight, u = heapq.heappop(min\_heap)

if visited[u]:

continue

visited[u] = True

mst\_weight += weight

edges += 1

# 遍历u的所有邻接边

for v, w in graph[u]:

if not visited[v]:

heapq.heappush(min\_heap, (w, v))

return mst\_weight if edges == n - 1 else -1#-1表示图不连通

BellmanFord Shuntingyard调度场

if min\_price!=INF:

            return min\_price

        else:

            return -1

def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: int, k: int) -> int:

        max\_flights=k+1

        INF=float('inf')

        dp=[[INF]\*n for x in range(max\_flights+1)]

        ##到n个目的地，k+1次

        dp[0][src]=0

        for m in range(1,max\_flights+1):

            for i in range(n):

                dp[m][i]=dp[m-1][i]

            for u,v,w in flights:

                if dp[m-1][u]!=INF:

                    if dp[m-1][u]+w<dp[m][v]:

                        dp[m][v]=dp[m-1][u]+w

        min\_price=INF

        for m in range(1,max\_flights+1):

            if dp[m][dst]<min\_price:

                min\_price=dp[m][dst]

def bellman\_ford\_with\_path(graph, start):

n = max(max(u, v) for u, v, \_ in graph) + 1

dist = [float('inf')] \* n

dist[start] = 0

prev = [-1] \* n # 记录前驱节点

for \_ in range(n - 1):

updated = False

for u, v, w in graph:

if dist[u] + w < dist[v]:

dist[v] = dist[u] + w

prev[v] = u # 记录前驱节点

updated = True

if not updated:

break

# 检测负权环

for u, v, w in graph:

if dist[u] + w < dist[v]:

return None, True, []

# 构建路径函数

def get\_path(end):

path = []

current = end

while current != -1:

path.append(current)

current = prev[current]

return path[::-1] if path[-1] == start else []

return dist, False, get\_path # 返回距离数组、是否有环、路径构建函数

栈的调度场算法，将人类习惯的中序表达式转化为后缀运算式（逆波兰）

def shunting\_yard(infix):

precedence = {'+': 1, '-': 1, '\*': 2, '/': 2, '^': 3}

associativity = {'+': 'left', '-': 'left', '\*': 'left', '/': 'left', '^': 'right'}

output = []

stack = []

for token in infix:

if token.isdigit(): # 操作数（简化处理，仅支持数字）

output.append(token)

elif token == '(': # 左括号

stack.append(token)

elif token == ')': # 右括号

while stack and stack[-1] != '(':

output.append(stack.pop())

stack.pop() # 弹出左括号

else: # 运算符

while stack and stack[-1] != '(':

top = stack[-1]

# 比较优先级和结合性

if (precedence[token] < precedence[top] or

(precedence[token] == precedence[top] and associativity[token] == 'left')):

output.append(stack.pop())

else:

break

stack.append(token)

# 处理剩余运算符

while stack:

output.append(stack.pop())

return output

后序表达式转前序思路：

使用栈存储操作数（或临时表达式）  
遍历后缀表达式的每个元素：

若遇到操作数，直接入栈。

若遇到运算符，则从栈中弹出前两个元素（注意顺序：后弹出的是左操作数，先弹出的是右操作数），将它们与运算符组合成一个带括号的中缀表达式，然后将该表达式重新入栈。

遍历结束后，栈顶元素即为最终的中缀表达式  
（若表达式合法，最终栈中只剩一个元素）。

**最短路径**

一个最简单bfs的模板

for \_ in range(T):t个样例

R, C = map(int, input().split())

grid = []

start = None

end = None

for i in range(R):

##遍历找起点终点坐标

visited = [[False for \_ in range(C)] for \_ in range(R)]

queue = deque()

queue.append((start[0], start[1], 0))

visited[start[0]][start[1]] = True

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

found = False

result = -1

while queue:

i, j, steps = queue.popleft()

if (i, j) == end:

result = steps

found = True

break

for di, dj in directions:

ni, nj = i + di, j + dj

if 0 <= ni < R and 0 <= nj < C:

if not visited[ni][nj] and grid[ni][nj] != '#':

visited[ni][nj] = True

queue.append((ni, nj, steps + 1))

print(result if found else "oop!")

用于求解图中****所有节点对之间最短路径****的经典动态规划算法。它可以处理有向图和无向图，并且允许图中存在负权边（但不能有负权环）

def floyd\_warshall(graph, n):

# 初始化距离矩阵为无穷大

dist = [[float('inf')] \* n for \_ in range(n)]

# 对角线元素设为0（节点到自身的距离）

for i in range(n):

dist[i][i] = 0

# 读取图的边信息，初始化直接相连的节点距离

for u, v, w in graph: # 假设graph是三元组列表[(u, v, weight)]

dist[u][v] = min(dist[u][v], w)

# 若为无向图，需添加反向边: dist[v][u] = min(dist[v][u], w)

# Floyd-Warshall核心更新

for k in range(n):

for i in range(n):

for j in range(n):

if dist[i][k] + dist[k][j] < dist[i][j]:

dist[i][j] = dist[i][k] + dist[k][j]

# 需要检测负权环，如果有则算法不适合

return dist

在把（u，v，w）存入后，可以用dijkstra来处理图中的路径问题，（找到最短路径的经过节点）

def Dijkstra(graph, start, end):

if start == end:

return []

distances = {place:float('inf') for place in graph.keys()}

paths = {place:[] for place in graph.keys()}

distances[start] = 0

heap = []

heapq.heappush(heap, (0, [], start))

while heap:

d, path, cur = heapq.heappop(heap)

for neighbor, nd in graph[cur].items():

if d + nd < distances[neighbor]:

distances[neighbor] = d + nd

paths[neighbor] = path + [neighbor]

heapq.heappush(heap, (distances[neighbor], paths[neighbor], neighbor))

return paths[end]

处理需要考虑不同事件之间的先后关系时用拓扑排序。

得到的拓扑序列代表可以按照给定先后次序走完的事件列表。

同时判断一张由有向边组成的图中是否存在环也可以用拓扑排序，len<n即为有环

def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:

        from collections import deque

        from collections import defaultdict

        graph=defaultdict(list)

        in\_degree=[0]\*(numCourses)

        for a in prerequisites:

            start=a[1]

            end=a[0]

            graph[start].append(end)

            in\_degree[end]+=1

        queue = deque([i for i in range(numCourses) if in\_degree[i] == 0])

        result=[]

        while queue:

            u=queue.popleft()

            result.append(u)

            for v in graph[u]:

                in\_degree[v]-=1

                if in\_degree[v]==0:

                    queue.append(v)

        if len(result)==numCourses:

            return True

        else:

            return False

单调栈查找数组中每个数右侧最大的数

def nextGreaterElements(nums):

stack = []

result = [-1] \* len(nums)

for i in range(len(nums)):

while stack and nums[i] > nums[stack[-1]]:

idx = stack.pop()

result[idx] = nums[i]

stack.append(i)

return result

查找数组中的数过多少会变大

def dailyTemperatures(temperatures):

stack = []

result = [0] \* len(temperatures)

for i in range(len(temperatures)):

while stack and temperatures[i] > temperatures[stack[-1]]:

idx = stack.pop()

result[idx] = i - idx

stack.append(i)

return result

直方图最大矩形面积

def largestRectangleArea(heights):

stack = []

max\_area = 0

heights = [0] + heights + [0] # 简化边界处理

for i in range(len(heights)):

while stack and heights[i] < heights[stack[-1]]:

h = heights[stack.pop()]

w = i - stack[-1] - 1

max\_area = max(max\_area, h \* w)

stack.append(i)

return max\_area

并查集

def find(x):

if parent[x] != x:

parent[x] = find(parent[x])

return parent[x]

def union(x, y):

root\_x = find(x)

root\_y = find(y)

if root\_x != root\_y:

parent[root\_y] = root\_x

Find查找是否同一个上级

Union合并两个小树，再按照规则设置一个总上司。

parent = list(range(n \* n)) # 一维化存储并查集

size = [1] \* (n \* n) def index(i, j):

return i \* n + j##用一个转换函数来使得二维转一维

有时进行查找操作时用集合查找更快

当题目满足以下条件时，优先考虑使用并查集：

1. 问题涉及 ****动态连通性**** 或 ****等价关系****。
2. 需要高效地进行 ****合并**** 和 ****查询**** 操作。
3. 数据结构中可能出现 ****森林****（多个不相交的树）。

问题是否涉及 ****动态连通性**** 和 ****集合合并****。

# 初始化并查集数据结构 秩优化

def init\_union\_find(size):

parent = list(range(size)) # 每个元素的父节点初始为自身

rank = [1] \* size # 每个集合的秩初始为1

count = size # 连通分量数量

return parent, rank, count

# 查找元素x的根节点（带路径压缩）

def find(parent, x):

if parent[x] != x:

parent[x] = find(parent, parent[x]) # 路径压缩

return parent[x]

# 合并x和y所在的集合（按秩合并）

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

root\_x = find(parent, x)

root\_y = find(parent, y)

if root\_x == root\_y:

return False # 已在同一集合

if rank[root\_x] < rank[root\_y]:

parent[root\_x] = root\_y

elif rank[root\_x] > rank[root\_y]:

parent[root\_y] = root\_x

else:

parent[root\_y] = root\_x

rank[root\_x] += 1

return True

# 判断x和y是否连通

def is\_connected(parent, x, y):

return find(parent, x) == find(parent, y)

# 获取连通分量数量

def get\_count(count):

return count

def build\_tree\_from\_leaves(self, levels):MSTleaves建树

# 反转阶段，从最后删除的叶子开始构建（最深层节点）

levels = levels[::-1]

nodes = set() # 记录所有节点

parent\_map = {} # 记录每个节点的父节点

for level in levels:

current\_leaves = list(level)

for leaf in current\_leaves:

nodes.add(leaf)

# 寻找当前叶子的父节点：BST中最邻近的前驱或后继

smaller = [n for n in nodes if n < leaf and n != leaf]

larger = [n for n in nodes if n > leaf and n != leaf]

parent = None

if smaller:

# 前驱：最大的较小节点

parent\_candidate = max(smaller)

parent = parent\_candidate

if larger:

# 后继：最小的较大节点

successor\_candidate = min(larger)

# 若同时存在前驱和后继，选择较小的父节点（确保BST结构）

if parent is None or successor\_candidate < parent:

parent = successor\_candidate

parent\_map[leaf] = parent# 可能为None（根节点）或有效父节点

# 确定根节点：没有父节点的节点

root = None

for node in nodes:

if parent\_map[node] is None:

root = node

break

# 生成先序遍历（非递归）

preorder = []

stack = [(root, False)] # (节点, 是否已访问)

while stack:

node, visited = stack.pop()

if visited:

preorder.append(node)

continue

# 先访问根节点，再压入右子节点和左子节点（确保左子节点先处理）

stack.append((node, True))

# 查找左右子节点（BST中左子节点 < 父节点，右子节点 > 父节点）

left\_child = None

right\_child = None

for child in parent\_map:

if parent\_map[child] == node:

if child < node:

left\_child = child

else:

right\_child = child

# 右子节点后压栈，确保左子节点先被处理

if right\_child:

stack.append((right\_child, False))

if left\_child:

stack.append((left\_child, False))

return ''.join(preorder)

import heapqfrom collections import defaultdict

##由各个节点的频率生成huffman编码树

如果已知各各点的权值（建最小堆），且各个点位置自由安排，则每次取最小的相加，值加到总计数中，然后再把新职加入堆

class Node:

def \_\_init\_\_(self, val, left=None, right=None):

self.val = val # 权值（频率）

self.left = left

self.right = right

def \_\_lt\_\_(self, other):#用于优先队列比较（按权值从小到大）

return self.val < other.val

def build\_huffman\_tree(freq):

heap = []

for char, count in freq.items():# 叶子节点存储字符

heapq.heappush(heap, Node(count, char=char)) while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

parent = Node(left.val + right.val, left, right)

heapq.heappush(heap, parent)

return heap[0] # 返回根节点

def generate\_codes(root, code="", huffman\_codes={}):

if root.char: # 叶子节点，记录编码

huffman\_codes[root.char] = code

return

generate\_codes(root.left, code + "0", huffman\_codes)

generate\_codes(root.right, code + "1", huffman\_codes)

return huffman\_codes

树转化为链表

def flatten(self, root: Optional[TreeNode]) -> None:

        """

        Do not return anything, modify root in-place instead.

        """

        if not root:

            return

        self.flatten(root.left)

        self.flatten(root.right)

        tmp=root.right

        root.right=root.left

        root.left=None

        p=root

        while p.right:

            p=p.right

        p.right=tmp

根据有序数组建平衡二叉树

也可以对单调递增数组nums使用以生成二叉搜索树

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

        if not nums:

            return None

        n=len(nums)

        mid=n//2

        root=TreeNode(nums[mid])

        root.left=self.sortedArrayToBST(nums[:mid])

        root.right=self.sortedArrayToBST(nums[mid+1:])

        return root

expression = "(5 (4 (11 (7 () ()) (2 () ()) ) ()) (8 (13 () ()) (4 () (1 () ()) ) ) )"##处理这样格式的树，建树

def build\_tree(s):

s = s.strip()

if not s or s == "()":

return None

if s[0] == '(' and s[-1] == ')':

s = s[1:-1].strip()

first\_space = s.find(' ')

if first\_space == -1:

return TreeNode(int(s))

val = int(s[:first\_space])

remaining = s[first\_space+1:].strip()

count = 0

left\_start = 0

for i in range(len(remaining)):

if remaining[i] == '(':

count += 1

elif remaining[i] == ')':

count -= 1

if count == 0 and remaining[i] == ')':

left\_subtree = remaining[left\_start:i+1]

right\_subtree = remaining[i+1:].strip()

Break

left = build\_tree(left\_subtree)

right = build\_tree(right\_subtree)

return TreeNode(val, left, right)

root = build\_tree(expression)

动态插入构建bst,大小比较是核心

def insert(root, val):

if root is None:

return TreeNode(val)

if val < root.val:

root.left = insert(root.left, val)

else:

root.right = insert(root.right, val)

return root

最小生成树，即在树的左右子树和本身有大小关系

def buildTree(preorder):

index=0##根据扩展后带‘.’的前序数列生成树

def helper():

nonlocal index

if index>len(preorder):

return None

val=preorder[index]

index+=1

if val=='.':

return None

node=TreeNode(val)

node.left=helper()

node.right=helper()

return node

return helper()

计算树的直径：self.max\_diameter = 0

def depth(node):

if not node:

return 0

left\_depth = depth(node.left)

right\_depth = depth(node.right)

self.max\_diameter = max(self.max\_diameter, left\_depth + right\_depth)# 返回当前节点的深度

return max(left\_depth, right\_depth) + 1

depth(root)return self.max\_diameter

遍历根到叶节点所有的路径

def get\_root\_to\_leaf\_paths(root):

paths = []

def dfs(node, current\_path):

if not node:

return

# 将当前节点值加入路径

current\_path.append(node.val)

# 如果是叶子节点，记录路径

if not node.left and not node.right:

paths.append(current\_path.copy()) # 保存路径副本

else:

# 递归遍历左右子树

dfs(node.left, current\_path)

dfs(node.right, current\_path)

# 回溯：移除当前节点（避免影响其他路径）

current\_path.pop()

dfs(root, [])

return paths

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):等等

self.val = val

self.left = left

self.right = right

def bstFromPreorder(preorder):

index = 0 # 全局索引，记录当前处理的元素

Def buildtreefrompreorder(preorder):

Index=0

def helper(lower, upper):由

nonlocal index

if index >= len(preorder):

return None

val = preorder[index]

if val < lower or val > upper:

return None

index += 1

node = TreeNode(val)

node.left = helper(lower, val) # 左子树的上界为当前节点值

node.right = helper(val, upper) # 右子树的下界为当前节点值

return node

return helper(float('-inf'), float('inf'))

如果给的是后序序列，则index从最后一位开始移动，i逐渐减一，在indeed<0时退出

给的是中序序列，则逐渐二分递归

def bstFromInorder(inorder):

if not inorder:

return None

mid = len(inorder) // 2

root = TreeNode(inorder[mid])

root.left = bstFromInorder(inorder[:mid]) # 左子树：小于根节点的元素

root.right = bstFromInorder(inorder[mid+1:]) # 右子树：大于根节点的元素 return root

判断是否有效二叉搜索树，递归的写法

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

        def validate(node,low=float('-inf'),high=float('inf')):

            if not node:

                return True

            if node.val<=low or node.val>=high:

                return False

            return validate(node.left,low,node.val) and validate(node.right,node.val,high)

        return validate(root)

def parse\_tree(s):括号嵌套格式的二叉树

if s == '\*': # 处理空树

return None

if '(' not in s: # 只有单个根节点

return TreeNode(s)

root\_value = s[0] # 根节点值

subtrees = s[2:-1] # 去掉根节点和外层括号

# 使用栈找到逗号位置

stack = []

comma\_index = None

for i, char in enumerate(subtrees):

if char == '(':

stack.append(char)

elif char == ')':

stack.pop()

elif char == ',' and not stack:

comma\_index = i

break

left\_subtree = subtrees[:comma\_index] if comma\_index is not None else subtrees

right\_subtree = subtrees[comma\_index + 1:] if comma\_index is not None else None

root = TreeNode(root\_value)

root.left = parse\_tree(left\_subtree) # 解析左子树

root.right = parse\_tree(right\_subtree) if right\_subtree else None # 解析右子树

return root

class TreeNode:##由1到n构建完全二叉树，且可以索引父节点

def \_\_init\_\_(self, value=0, left=None, right=None, parent=None):

self.value = value

self.left = left

self.right = right

self.parent = parent

def build\_complete\_binary\_tree(n):

if n <= 0:

return None

nodes = [TreeNode(i) for i in range(1, n + 1)]

for i in range(n // 2):

nodes[i].left = nodes[2 \* i + 1]

nodes[i].right = nodes[2 \* i + 2]

nodes[2 \* i + 1].parent = nodes[i]

nodes[2 \* i + 2].parent = nodes[i]

# 如果节点数为奇数，单独处理最后一个节点的左子节点

if n % 2 == 1:

nodes[n // 2].left = nodes[n - 1]

nodes[n - 1].parent = nodes[n // 2]

return nodes[0]

from collections import deque由层序遍历建树

def build\_tree(levelorder):

if not levelorder:

return None

root = TreeNode(levelorder[0])

queue = deque([root])

idx = 1

while queue and idx < len(levelorder):

node = queue.popleft()

if levelorder[idx] is not None:

node.left = TreeNode(levelorder[idx])

queue.append(node.left)

idx += 1

if idx < len(levelorder) and levelorder[idx] is not None:

node.right = TreeNode(levelorder[idx])

queue.append(node.right)

idx += 1 return root

def invert\_tree(root):二叉树镜像反转

if root is None:

return None

root.left, root.right = invert\_tree(root.right), invert\_tree(root.left)

return root

from collections import deque层序遍历

def levelorder(root):

result = []

if root is None:

return result

queue = deque([root])

while queue:

level\_size = len(queue)

level = []

for \_ in range(level\_size):

node = queue.popleft()

level.append(node.val)

if node.left:

queue.append(node.left)

if node.right:

queue.append(node.right)

result.append(level)

return result

def preorder(root):前序遍历，val在前面其余同理

if root is None:

return []

return [root.val] + preorder(root.left) + preorder(root.right)

def build\_tree(preorder, inorder):前序和中序遍历建树

if not preorder or not inorder:

return None

root\_val = preorder[0]

root = TreeNode(root\_val)

root\_idx = inorder.index(root\_val)

root.left = build\_tree(preorder[1:root\_idx+1], inorder[:root\_idx])

root.right = build\_tree(preorder[root\_idx+1:], inorder[root\_idx+1:])

return root

def count\_leaves(root):二叉树叶子节点树

if root is None:

return 0

if root.left is None and root.right is None:

return 1

return count\_leaves(root.left) + count\_leaves(root.right)

def count\_nodes(root):二叉树节点总数

if root is None:

return 0

return 1 + count\_nodes(root.left) + count\_nodes(root.right)

def max\_depth(root):二叉树深度，根到最远叶子

if root is None:

return 0

left\_depth = max\_depth(root.left)

right\_depth = max\_depth(root.right)

return max(left\_depth, right\_depth) + 1