1. 题目

LC108.将有序数组转换为二叉树

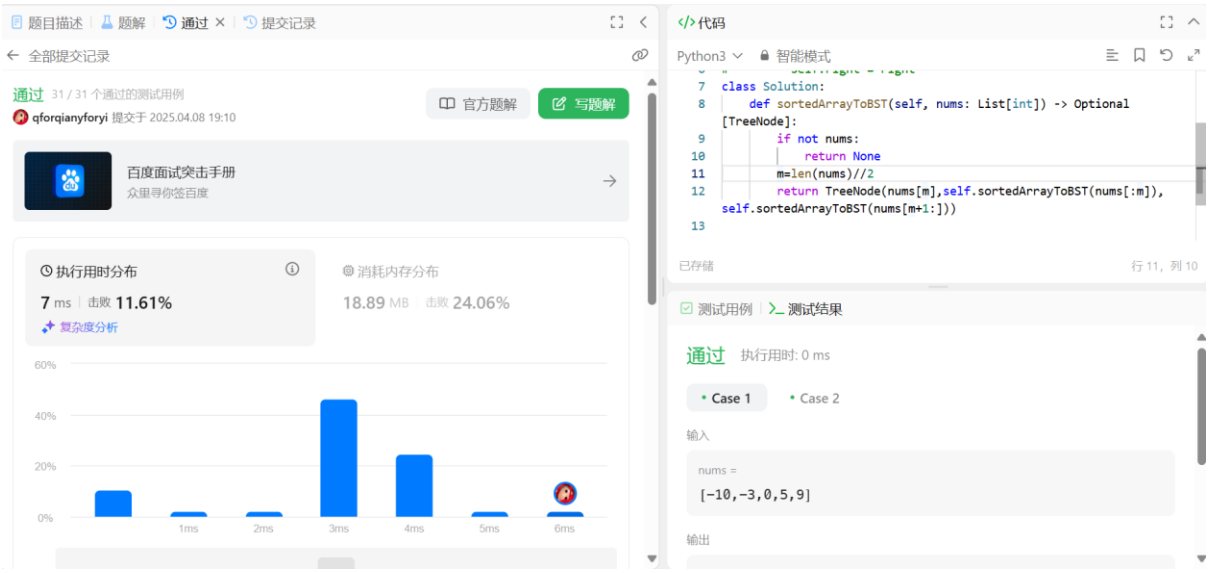
dfs, https://leetcode.cn/problems/convert-sorted-array-to-binary-search-tree/

思路：先取列表中间值作为当前节点，并递归左子树&右子树，每次返回递归的当前节点直到 nums 为空。

代码：

```
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
        if not nums:
            return None
        m=len(nums)//2
        return TreeNode(nums[m],self.sortedArrayToBST(nums[:m]),self.sortedArrayToBST(nums[m+1:]))
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



大约用时：45 分钟

M27928:遍历树

adjacency list, dfs, http://cs101.openjudge.cn/practice/27928/

思路：将输入的每行的第一个根节点加入节点集合，子节点加入子节点集合同时加入节点集合（子节点集合不包括整个树的根节点），并将每节的根节点和子节点加存入字典中；之后将节点集合-子节点集合，剩下的就是整棵树的根节点（root），然后用 dfs（根节点），获取其子节点列表和根节点本身然后排序（先是子节点，后是根（当前）节点），确保子节点按照从小到大的顺序遍历，之后遍历这个排序后的列表并递归遍历子节点，如果找到当前节点（子节点）就打印，否则继续递归，直到找到最后的总根节点（结束）。

代码：

```
n=int(input())
nodes=set()
tree={}
sub=set()
for _ in range(n):
    parts=list(map(int, input().split()))
    main=parts[0]
```

```
    nodes.add(main)
    tree[main]=parts[1:]

    for i in parts[1:]:
        sub.add(i)
        nodes.add(i)

root=(nodes-sub).pop()
# print(root)
def dfs(node):
    sorted_nodes=sorted(tree.get(node,[])+[node])
    for x in sorted_nodes:
        if x == node:
            print(x)
        else:
            dfs(x)

dfs(root)
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

大约用时：1 小时 30 分钟

LC129.求根节点到叶节点数字之和

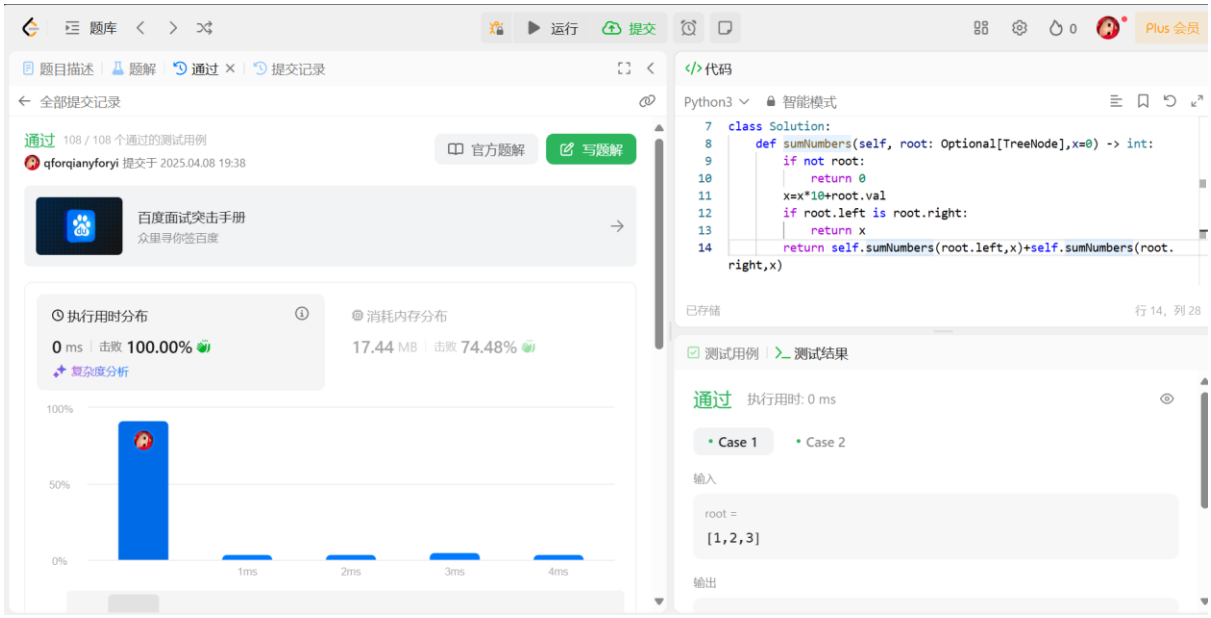
dfs, https://leetcode.cn/problems/sum-root-to-leaf-numbers/

思路：定义一个 x 用于记录当前的数字，每往下递归一层就把当前节点的数字拼到 x 中，当 root.left 和 root.left（当前节点的下一层左右节点）都为 None 时返回拼好的数字，重复递归左右子树的节点并把左右子树的汇总结果加起来。

代码：

```
class Solution:
    def sumNumbers(self, root: Optional[TreeNode],x=0) -> int:
        if not root:
            return 0
        x=x*10+root.val
        if root.left is root.right:
            return x
        return self.sumNumbers(root.left,x)+self.sumNumbers(root.right,x)
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



大约用时：45 分钟

M22158:根据二叉树前中序序列建树

tree, http://cs101.openjudge.cn/practice/24729/

思路：找到前、中序遍历的根节点（前序：根节点 -> 左子树 -> 右子树；中序：左子树 -> 根节点 -> 右子树），再找到前、中序遍历的左右子树，如何用前、中序遍历的左子树递归构建后序遍历的左子树，同样用前、中序遍历的右子树递归构建后序遍历的右子树（每次递归到下一层就有新的根节点和前、中遍历的左右子树，直到找到空），最后找到后序遍历（左子树 -> 右子树 -> 根节点）。

代码：

```
def postorder(preorder,inorder):
    if not preorder or not inorder:
        return ""

    root=preorder[0]  # 前序遍历的第一个字符是根节点
    root_index=inorder.index(root)  # 找到根节点在中序遍历中的位置

    # 切分前序和中序遍历成左右子树
```

```
    left_inorder=inorder[:root_index]
    right_inorder=inorder[root_index + 1:]

    left_preorder=preorder[1:1+len(left_inorder)]
    right_preorder=preorder[1+len(left_inorder):]

    # 递归构建左右子树的后序遍历
    left_postorder=postorder(left_preorder,left_inorder)
    right_postorder=postorder(right_preorder,right_inorder)

    # 后序遍历顺序：左子树 -> 右子树 -> 根节点
    return left_postorder + right_postorder + root

while True:
    try:
        preorder=input()
        inorder=input()
        print(postorder(preorder, inorder))
    except EOFError:
        break
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

#48859799提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
def postorder(preorder,inorder):
    if not preorder or not inorder:
        return ""

    root=preorder[0]  # 前序遍历的第一个字符是根节点
    root_index=inorder.index(root)  # 找到根节点在中序遍历中的位置

    # 切分前序和中序遍历成左右子树
    left_inorder=inorder[:root_index]
    right_inorder=inorder[root_index + 1:]

    left_preorder=preorder[1:1+len(left_inorder)]
    right_preorder=preorder[1+len(left_inorder):]

    # 递归构建左右子树的后序遍历
    left_postorder=postorder(left_preorder,left_inorder)
    right_postorder=postorder(right_preorder,right_inorder)

    # 后序遍历顺序：左子树 -> 右子树 -> 根节点
    return left_postorder + right_postorder + root

while True:
    try:
        preorder=input()
        inorder=input()
        print(postorder(preorder, inorder))
    except EOFError:
        break
```

基本信息

#: 48859799
题目: 22158
提交人: 2400093012 苏倩仪
内存: 3592kB
时间: 19ms
语言: Python3
提交时间: 2025-04-09 15:22:40

English 帮助 关于

大约用时：45 分钟

T24729:括号嵌套树

dfs, stack, http://cs101.openjudge.cn/practice/24729/

思路：需要定义树的节点类（每个节点的一个值 val 和一个子节点列表 child），然后先定义前序遍历和后序遍历的函数（递归构建），方便之后直接调用，然后用栈来依次判断输入的每个元素是什么并进行对应的操作（如果是第一个大写字母=根节点，如果“（”将上一个压入栈表示接下来的元素是子节点，“，”=跳过，“）”=弹出栈中最后一个元素），之后将得到的 root 调用前序和后序遍历得到所需的结果。

代码：

```
class TreeNode:
    def __init__(self,val):
        self.val=val # 存储节点的值（大写字母）
        self.child=[] # 一个列表，存储该节点的所有子节点
```

```
def preorder(root,res): # 前序遍历
    if not root:
        return
    res.append(root.val)
    for i in root.child:
        preorder(i,res)

def postorder(root,res): # 后序遍历
    if not root:
        return
    for i in root.child:
        postorder(i,res)
    res.append(root.val)

s=input()
stack=[]
root=None
i=0
while i<len(s):
    char=s[i]
    if char.isalpha(): # 判断大写字母
        node=TreeNode(char)
        if not root:
            root=node # 第一个节点
        if stack: # 如果不为 None 说明当前节点有父节点
            stack[-1].child.append(node) # 当前节点添加为栈顶节点的子节点
        last=node # 更新为当前节点
        i += 1
    elif char == '(':
        stack.append(last) # 将最近的节点压入栈（接下来的节点是子节点）
        i += 1
    elif char == ',':
        i += 1 # 跳过
    elif char == ')':
        stack.pop() # 弹出栈顶元素（当前子树结束）
        i += 1

# print(root)
pre_res=[]
post_res=[]
```

preorder(root,pre_res)
postorder(root,post_res)
print("".join(pre_res))
print("".join(post_res))

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

#48870774提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
class TreeNode:
    def __init__(self,val):
        self.val=val # 存储节点的值（大写字母）
        self.child=[] # 一个列表，存储该节点的所有子节点

def preorder(root,res): # 前序遍历
    if not root:
        return
    res.append(root.val)
    for i in root.child:
        preorder(i,res)

def postorder(root,res): # 后序遍历
    if not root:
        return
    for i in root.child:
        postorder(i,res)
    res.append(root.val)

s=input()
stack=[]
root=None
i=0
while i<len(s):
    char=s[i]
    if char.isalpha(): # 判断大写字母
        node=TreeNode(char)
        if not root:
            root=node # 第一个节点
        if stack: # 如果不为None说明当前节点有父节点
            stack[-1].child.append(node) # 当前节点添加为栈顶节点的子节点
        last=node # 更新为当前节点
        i += 1
    elif char == '(':
        stack.append(last) # 将最近的节点压入栈（接下来的节点是子节点）
        i += 1
    elif char == ',':
        i += 1 # 跳过
    elif char == ')':
        stack.pop() # 弹出栈顶元素（当前子树结束）
        i += 1

# print(root)
pre_res=[]
post_res=[]
preorder(root,pre_res)
postorder(root,post_res)
print("".join(pre_res))
print("".join(post_res))
```

基本信息

#: 48870774
题目: 24729
提交人: 2400093012 苏倩仪
内存: 3652kB
时间: 20ms
语言: Python3
提交时间: 2025-04-10 15:39:56

English 帮助 关于

大约用时：1 小时

LC3510.移除最小数对使数组有序 II

doubly-linked list + heap, https://leetcode.cn/problems/minimum-pair-removal-to-sort-array-ii/

思路：首先遍历数组，找出递减对数量并初始化堆，堆用来记录每对相邻数字的和以及它们的位置，其中递减对会按和从小到大排序，接着使用堆来删除递减对，通过 right 和 left 数组管理删除操作和并模拟双向链表的删除，保证每次删除操作都有效，避免重复计算（懒删除），如果有递减对，则进入循环并处理过时的数据（堆顶元素的下标>=数组长度/堆顶的和不等于实际数组中这两个元素的和），每次删除一对递减对（取出堆中的有效元素并检查当前元素的左右，更新 dec 并在堆中重新插入更新后的元素（分别处理 next：检查是否产生新的递减对；pre：更新 dec 并更新堆；next2：更新递减对的计数和堆），最后模拟双向链表的删除操作（next 的左边的值指向右边的值，右边的值指向左边的值，并删除 next，这样避免了重复删除操作的出现），直到递减对数量 dec 为零，返回 ans（删除的递减对次数）

代码：

```
class Solution:
    def minimumPairRemoval(self, nums: List[int]) -> int:
        n = len(nums) # 数组的长度
        h = []  # 堆，用来保存每对相邻数字的和以及它们的下标
        dec = 0  # 递减的相邻对的个数
        for i, (x, y) in enumerate(pairwise(nums)): # 获取相邻的数字对 (x, y)
            if x > y: # 如果一个数字比下一个数字大，说明这是一个递减的对
                dec += 1 # 记录递减对的个数
            h.append((x + y, i)) # 将每对相邻数字的和 x + y 和它们的下标 i 加入堆 h
        heapify(h) # 将列表 h 转换成堆结构，保证堆顶是和最小的相邻对

        # left 和 right 用来记录每个节点左右相邻的未删除的数字位置
        left = list(range(-1, n))  # 防止下标越界
```

right = list(range(1, n + 1)) # 注意最下面的代码，删除 next 的时候额外把 right[next] 置为 n

ans = 0 # 记录删除的配对次数

while dec: # 大于 0 时继续循环，表示还有递减对需要处理

ans += 1

“懒删除”操作：当堆顶的数据不再符合当前状态时（即堆顶的数据已经不合法），就跳过

while right[h[0][1]] >= n or h[0][0] != nums[h[0][1]] + nums[right[h[0][1]]]: # 该节点的右侧元素已经被删除，或者堆顶的数据不再匹配当前的数字和（即，数据已经过期），就弹出堆顶

heappop(h)

s, i = heappop(h) # 每次从堆中取出相邻数字和最小的一个，i 是下标

next = right[i] # 当前元素的右边最近未删除元素的下标

if nums[i] > nums[next]: # 旧数据（递减对，dec 减一）

dec -= 1

pre = left[i] # 当前元素的左边最近未删除元素的下标

if pre >= 0: # i 的左边有元素

if nums[pre] > nums[i]: # 旧数据（递减对， dec 减 1）

dec -= 1

if nums[pre] > s: # 新数据（添加新的递减对， dec 增加 1）

dec += 1

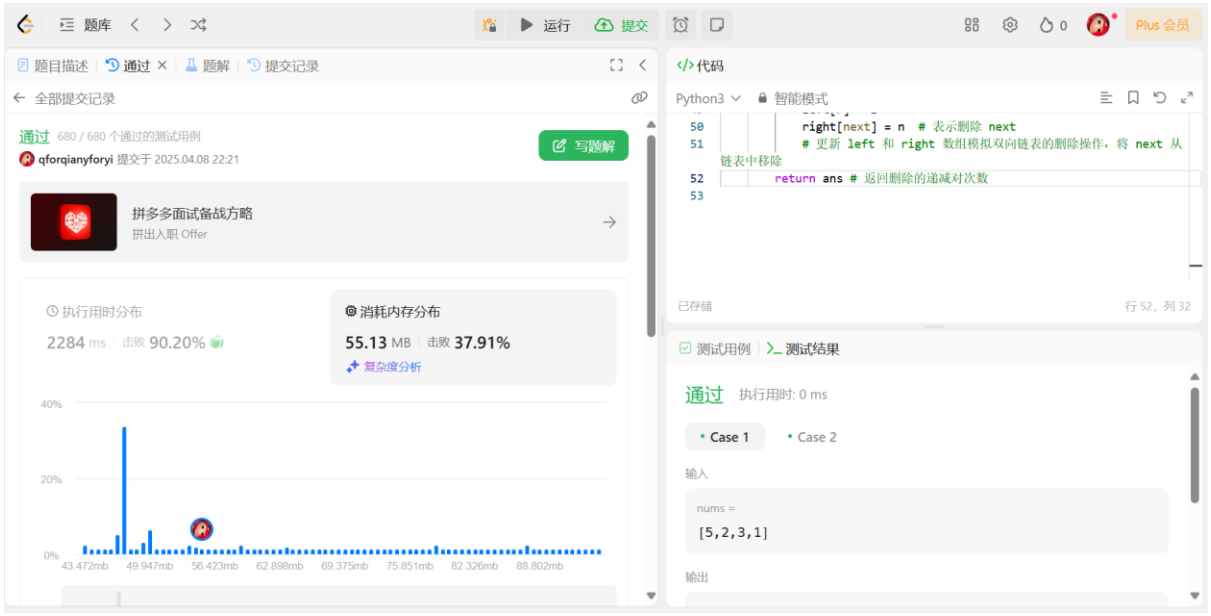
heappush(h, (nums[pre] + s, pre)) #更新后的 (nums[pre] + s) 和 pre 重新加入堆中

next2 = right[next] # next 的右边最近未删除元素的下标

```
        if next2 < n: # next 的右边还有元素
            if nums[next] > nums[next2]:  # 旧数据（减对，dec 减 1）
                dec -= 1
            if s > nums[next2]:  # 新数据（新的递减对，dec 加 1）
                dec += 1
            heappush(h, (s + nums[next2], i)) # 更新后的 (s + nums[next2]) 和 i 重新加入堆中

        nums[i] = s # 更新 nums[i]
        # 删除 next
        l, r = left[next], right[next]
        right[l] = r  # 模拟双向链表的删除操作
        left[r] = l
        right[next] = n  # 表示删除 next
        # 更新 left 和 right 数组模拟双向链表的删除操作，将 next 从链表中移除
    return ans  # 返回删除的递减对次数
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



大约用时：2 小时

2. 学习总结和收获

<mark>如果发现作业题目相对简单，有否寻找额外的练习题目，如“数算 2025spring 每日选做”、LeetCode、Codeforces、洛谷等网站上的题目。</mark>

一开始不知道为什么第一题递归左右子树时要加上 self.后来知道了是因为调用的是“自己类里的方法”（同一个类里的另一个函数）所以要加上，否则程序不知道说的是哪个函数。第二题题目看了好久一直不明白样例 2 的树是长什么样的 hhh，ai 也一直给我错的答案，最后问了朋友才明白，用总节点结合-子节点集合找到根节点的方式和 dfs 中将每个根节点放在最后查找的方式也很特别~学到了学到了。从第三题学到 if root.left is root.right 即二者都是 None 时说明到了节点结尾的神奇技巧。第四题写起来不难，但是要先知道可以递归用前中遍历的左/右子树找到后序遍历这个方法。第五题也不难，但是要自己定义树的类有点复杂，不知道要怎么定义，写栈的时候对子节点的处理一开始也有点混乱，但稍加思考能大概写出来（但是也不对，最后交给 gpt 捉虫了）。第六题好复杂（自己完全无从下手，之后搭配题解和 AI 艰难的理解了，对堆的概念有了比之前更清晰的认识，可是双向链表对我来说还是很模糊，只能勉强看懂。这次好多树，逃不了了 hhh，不含链表操作的树还是挺好做的，这次原本打算不用 ai 但是怕死磕太久来不及读期中就还是用了 TT。