Assignment #A: 递归回溯、🌲 (3/4)

Updated 2203 GMT+8 Nov 3, 2025

2025 fall, Complied by 林奕妃、环境科学与工程学院

> **说明：**
>
> 1. **解题与记录：**
>
>   对于每一个题目，请提供其解题思路（可选），并附上使用Python或C++编写的源代码（确保已在OpenJudge， Codeforces，LeetCode等平台上获得Accepted）。请将这些信息连同显示“Accepted”的截图一起填写到下方的作业模板中。（推荐使用Typora https://typoraio.cn 进行编辑，当然你也可以选择Word。）无论题目是否已通过，请标明每个题目大致花费的时间。
>
> 2. **提交安排：**提交时，请首先上传PDF格式的文件，并将.md或.doc格式的文件作为附件上传至右侧的“作业评论”区。确保你的Canvas账户有一个清晰可见的本人头像，提交的文件为PDF格式，并且“作业评论”区包含上传的.md或.doc附件。
>
> 3. **延迟提交：**如果你预计无法在截止日期前提交作业，请提前告知具体原因。这有助于我们了解情况并可能为你提供适当的延期或其他帮助。
>
> 请按照上述指导认真准备和提交作业，以保证顺利完成课程要求。

1. 题目

T51.N皇后（40分钟）

backtracking, https://leetcode.cn/problems/n-queens/

思路：逐行从第一列开始放置皇后，并检查该位置的合法性，即当前列与当前位置两个对角线均没有皇后，当找到合法位置后开始找下一个解。主对角线的格子满足行号-列号的值相等，副对角线的格子满足行号+列号的值相等。遍历所有可能的情况

代码：

```
from typing import List

class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        """
        使用回溯法解决 N 皇后问题
        """
        # 最终结果列表
        result = []
        # 存储一个解决方案中，每行皇后的列位置。
        # 例如 queens[0] = 1 表示第 0 行的皇后放在第 1 列。
        queens_positions = [-1] * n

        # 使用集合来快速检查列和对角线是否已被占用
```

```
        columns = set()
        diagonals1 = set()  # 主对角线 (row - col)
        diagonals2 = set()  # 副对角线 (row + col)

        def backtrack(row: int):
            """
            在指定的 row 行尝试放置皇后。
            """
            # 递归终止条件：当所有行都成功放置皇后
            if row == n:
                board = generate_board()
                result.append(board)
                return

            # 遍历当前行的每一列，尝试放置皇后
            for col in range(n):
                # 检查当前位置 (row, col) 是否会与已放置的皇后冲突
                if col in columns or (row - col) in diagonals1 or (row + col) in
diagonals2:
                    continue  # 如果冲突，则跳过这一列

                # --- 做出选择 ---
                # 记录皇后位置
                queens_positions[row] = col
                # 更新状态，标记占用的列和对角线
                columns.add(col)
                diagonals1.add(row - col)
                diagonals2.add(row + col)

                # --- 进入下一行决策 ---
                backtrack(row + 1)

                # --- 撤销选择 (回溯) ---
                # 恢复状态，以便在当前行的其他列继续尝试
                columns.remove(col)
                diagonals1.remove(row - col)
                diagonals2.remove(row + col)
                # queens_positions[row] = -1  # 此行不是必需的，因为下次循环会被覆盖

        def generate_board() -> List[str]:
            """
            根据 queens_positions 数组生成最终的棋盘字符串格式。
            """
            board = []
            for i in range(n):
                # 创建一个全为 '.' 的行
                row_list = ['.'] * n
                # 在皇后的位置上放置 'Q'
                row_list[queens_positions[i]] = 'Q'
                board.append("".join(row_list))
            return board

        # 从第 0 行开始启动回溯过程
        backtrack(0)
        return result
```

代码运行截图（至少包含有"Accepted"）

M22275: 二叉搜索树的遍历（30分钟）

http://cs101.openjudge.cn/practice/22275/

思路：二差搜索树的特点为任意节点的左子树中的所有节点的值都小于该节点的值，而右子树中的所有节点的值都大于该节点的值。因此此题与上次月考第一题不同，此题有前序遍历即可确定二叉树，可以通过大小比较确定左右子树。

代码：

```
import sys

# 题目中 n 最大为 2000，递归深度可能较大，增加递归深度限制
sys.setrecursionlimit(2000 + 50)

def solve():
    try:
        while True:
            line = sys.stdin.readline().strip()
            if not line:
                break

            n = int(line)
            if n == 0:
                print("")
                continue

            preorder = list(map(int, sys.stdin.readline().strip().split()))

            postorder_result = []

            def get_postorder(start, end):
                """
                根据前序遍历的子数组 preorder[start...end] 来构建后序遍历结果
                """
                # 基本情况：如果子数组为空或无效，则返回
                if start > end:
                    return

                # 根节点是子数组的第一个元素
                root_val = preorder[start]
```

```
                # 寻找右子树的起点
                # 遍历 start+1 到 end，找到第一个比 root_val 大的元素
                split_index = end + 1  # 默认情况下，假设没有右子树
                for i in range(start + 1, end + 1):
                    if preorder[i] > root_val:
                        split_index = i
                        break

                # 递归处理左子树
                # 左子树的范围是 [start + 1, split_index - 1]
                get_postorder(start + 1, split_index - 1)

                # 递归处理右子树
                # 右子树的范围是 [split_index, end]
                get_postorder(split_index, end)

                # 左、右子树都处理完毕后，将根节点加入结果
                postorder_result.append(root_val)

            # 对整个前序遍历序列启动递归过程
            get_postorder(0, n - 1)

            # 打印最终结果
            print(*postorder_result)

    except (IOError, ValueError):
        # 捕获可能的异常，例如文件结束或空行
        pass

# 调用主函数
solve()
```

代码运行截图（至少包含有"Accepted"）

#50866484提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
import sys

# 题目中 n 最大为 2000，递归深度可能较大，增加递归深度限制
sys.setrecursionlimit(2000 + 50)

def solve():
    try:
```

基本信息

#: 50866484
题目: 22275
提交人: 25n2200013554
内存: 3808kB
时间: 23ms
语言: Python3
提交时间: 2025-11-16 15:50:57

M25145: 猜二叉树（按层次遍历）（30分钟）

http://cs101.openjudge.cn/practice/25145/

思路：与上次月考题类似，由后续遍历确定根，由中序遍历确定左右子树，再按层次遍历逐层输出。

代码：

```python
import sys
from collections import deque

# 定义二叉树节点类
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def build_tree(inorder, postorder):
    """
    根据中序和后序遍历序列重建二叉树
    """
    # 使用哈希表优化查找根节点在中序遍历中的索引
    inorder_map = {val: i for i, val in enumerate(inorder)}

    def build(in_start, in_end, post_start, post_end):
        """
        递归辅助函数，使用索引来划分子树，避免列表切片的开销
        """
        # 基本情况：如果序列为空，返回空节点
        if in_start > in_end or post_start > post_end:
            return None

        # 后序遍历的最后一个节点是根
        root_val = postorder[post_end]
        root = TreeNode(root_val)

        # 在中序遍历中找到根节点的索引
        root_inorder_idx = inorder_map[root_val]

        # 计算左子树的节点数量
        left_subtree_size = root_inorder_idx - in_start

        # 递归构建左子树
        # 左子树的中序范围: [in_start, root_inorder_idx - 1]
        # 左子树的后序范围: [post_start, post_start + left_subtree_size - 1]
        root.left = build(in_start, root_inorder_idx - 1,
                          post_start, post_start + left_subtree_size - 1)

        # 递归构建右子树
        # 右子树的中序范围: [root_inorder_idx + 1, in_end]
        # 右子树的后序范围: [post_start + left_subtree_size, post_end - 1]
        root.right = build(root_inorder_idx + 1, in_end,
                           post_start + left_subtree_size, post_end - 1)

        return root

    # 启动递归过程
    n = len(inorder)
    return build(0, n - 1, 0, n - 1)

def level_order_traversal(root):
    """
```

```
    对二叉树进行层次遍历（BFS）
    """
    if not root:
        return ""

    result = []
    queue = deque([root])

    while queue:
        node = queue.popleft()
        result.append(node.val)

        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    return "".join(result)

def solve():
    """
    主解决函数，处理输入输出
    """
    try:
        # 读取测试用例的数量
        num_cases = int(sys.stdin.readline().strip())

        for _ in range(num_cases):
            # 读取中序和后序遍历序列
            inorder_str = sys.stdin.readline().strip()
            postorder_str = sys.stdin.readline().strip()

            # 1. 重建二叉树
            root = build_tree(inorder_str, postorder_str)

            # 2. 进行层次遍历并输出结果
            level_order_result = level_order_traversal(root)
            print(level_order_result)

    except (IOError, ValueError):
        # 处理可能的输入结束或格式错误
        pass

# 调用主函数
solve()
```

代码运行截图（至少包含有"Accepted"）

#50868467提交状态

查看　提交　统计　提问

状态: Accepted

源代码

```
import sys
from collections import deque

# 定义二叉树节点类
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```

基本信息

#: 50868467
题目: 25145
提交人: 25n2200013554
内存: 3640kB
时间: 24ms
语言: Python3
提交时间: 2025-11-16 17:16:11

T20576: printExp（逆波兰表达式建树）（1小时）

http://cs101.openjudge.cn/practice/20576/

思路：一个子表达式周围的括号是否"必要"，取决于这个子表达式的根运算符的优先级是否低于其外部（父）运算符的优先级。首先将中缀转后缀，将输入的中缀表达式字符串转换为后缀表达式（RPN）的token 序列。这一步可以清晰地处理掉所有的括号和运算符优先级。再根据后缀表达式序列构建一棵表达式树。在表达式树中，叶子节点是操作数（True, False），内部节点是运算符（and, or, not）。最后对构建好的表达式树进行一次特殊的中序遍历来生成最终的字符串。在遍历时，根据父子节点的运算符优先级来决定是否需要打印括号。

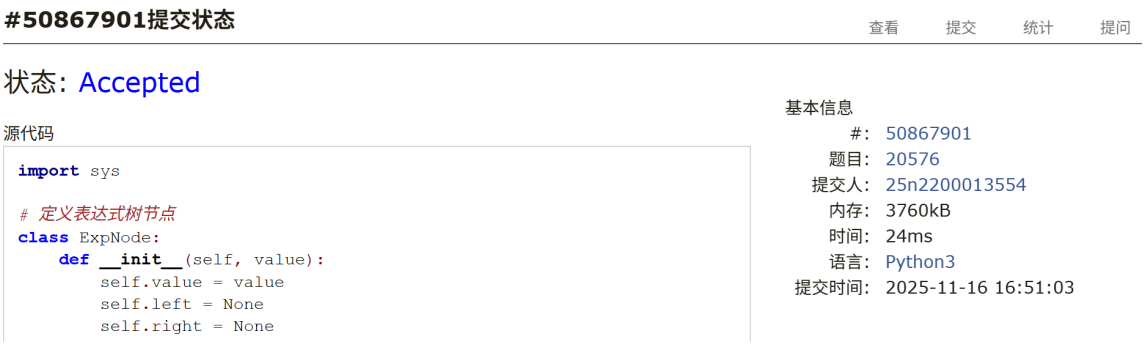
代码

```
import sys

# 定义表达式树节点
class ExpNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

# 定义运算符优先级
precedence = {
    'or': 1,
    'and': 2,
    'not': 3
}

def infix_to_postfix(tokens):
    """使用 Shunting-yard 算法将中缀表达式转为后缀表达式"""
    output = []
    ops = []
    for token in tokens:
        if token in ['True', 'False']:
            output.append(token)
        elif token in precedence:
            while (ops and ops[-1] != '(' and
                   precedence.get(ops[-1], 0) >= precedence.get(token, 0)):
                output.append(ops.pop())
            ops.append(token)
```

```
        elif token == '(':
            ops.append(token)
        elif token == ')':
            while ops and ops[-1] != '(':
                output.append(ops.pop())
            if ops and ops[-1] == '(':
                ops.pop()  # 弹出 '('
    while ops:
        output.append(ops.pop())
    return output

def build_tree_from_postfix(postfix_tokens):
    """从后缀表达式构建表达式树"""
    stack = []
    for token in postfix_tokens:
        if token in ['True', 'False']:
            stack.append(ExpNode(token))
        else: # Operator
            node = ExpNode(token)
            if token == 'not':
                if stack:
                    node.right = stack.pop()
            else: # and, or
                if len(stack) >= 2:
                    node.right = stack.pop()
                    node.left = stack.pop()
            stack.append(node)
    return stack[0] if stack else None

def print_tree_infix(node, parent_prec):
    """通过中序遍历表达式树，生成带最少括号的中缀表达式字符串"""
    if not node:
        return ""

    # 如果是叶子节点（操作数）
    if not node.left and node.value not in ['not']:
        return node.value

    current_prec = precedence.get(node.value, 0)

    # 根据运算符类型递归生成子表达式字符串
    if node.value == 'not':
        # not 的右孩子优先级按 not 自己的优先级传递
        s = "not " + print_tree_infix(node.right, current_prec)
    else: # and, or
        left_s = print_tree_infix(node.left, current_prec)
        right_s = print_tree_infix(node.right, current_prec)
        s = f"{left_s} {node.value} {right_s}"

    # 如果当前运算符优先级低于父运算符，需要加括号
    if current_prec < parent_prec:
        return f"( {s} )"
    else:
        return s

def solve():
```

```
    """主解决函数"""
    try:
        line = sys.stdin.readline().strip()
        if not line:
            return

        # 预处理输入，方便分割
        line = line.replace('(', ' ( ').replace(')', ' ) ')
        tokens = line.split()

        # 1. 中缀转后缀
        postfix = infix_to_postfix(tokens)

        # 2. 后缀建树
        root = build_tree_from_postfix(postfix)

        # 3. 遍历树并打印
        # 初始父优先级设为0，确保最外层表达式不会被不必要地加括号
        result = print_tree_infix(root, 0)

        print(result)

    except (IOError, ValueError):
        pass

solve()
```

代码运行截图（至少包含有"Accepted"）



#50867901提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
import sys

# 定义表达式树节点
class ExpNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

基本信息

#: 50867901
题目: 20576
提交人: 25n2200013554
内存: 3760kB
时间: 24ms
语言: Python3
提交时间: 2025-11-16 16:51:03

T04080:Huffman编码树（1小时）

greedy, http://cs101.openjudge.cn/practice/04080/

思路：为了使带权路径长度总和最小，我们应该让权值大的叶子节点离根节点越近（路径长度越短），让权值小的叶子节点离根节点越远（路径长度越长）。因此每一步都选择最小的两个来合并直至最终合并结束。

代码

```
import sys
import heapq
```

```python
def solve():
    """
    主解决函数，处理输入输出
    """
    try:
        # 读取外部节点个数 n
        line_n = sys.stdin.readline().strip()
        if not line_n:
            return
        n = int(line_n)

        # 读取 n 个权值
        weights = list(map(int, sys.stdin.readline().strip().split()))

        # 如果只有一个节点，路径长度为0（题目保证 N>=2，但代码可以更健壮）
        if n <= 1:
            print(0)
            return

        # 将所有权值放入一个最小堆（在 Python 中用列表模拟）
        # heapq.heapify 可以将列表原地转换为最小堆，效率比逐个 push 更高
        pq = weights
        heapq.heapify(pq)

        total_cost = 0

        # 当堆中元素多于一个时，持续合并
        while len(pq) > 1:
            # 1. 弹出权值最小的两个节点
            w1 = heapq.heappop(pq)
            w2 = heapq.heappop(pq)

            # 2. 计算合并后的权值（即合并的代价）
            combined_weight = w1 + w2

            # 3. 将代价累加到总和中
            total_cost += combined_weight

            # 4. 将合并后的新节点推回堆中
            heapq.heappush(pq, combined_weight)

        # 循环结束后，total_cost 就是最小带权路径长度
        print(total_cost)

    except (IOError, ValueError):
        # 处理可能的输入结束或格式错误
        pass

# 调用主函数
solve()
```

代码运行截图（至少包含有"Accepted"）

#50870184提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
import sys
import heapq

def solve():
    """
    主解决函数，处理输入输出
    """
    try:
```

基本信息

#: 50870184
题目: 04080
提交人: 25n2200013554
内存: 3632kB
时间: 24ms
语言: Python3
提交时间: 2025-11-16 19:06:15

M04078: 实现堆结构

http://cs101.openjudge.cn/practice/04078/

要求手搓堆实现。

思路：使用数组来表示堆结构。假设一个节点的索引是 i。它的父节点的索引是 (i - 1) // 2。它的左子节点的索引是 2 * i + 1。它的右子节点的索引是 2 * i + 2。数组的第一个元素（索引为0）就是堆的根节点。增添元素操作即将新元素添加到数组的末尾，再根据大小将其向上调整。输出并删除最小元素，最小的元素就是根节点，将数组最后一个元素移动到根节点的位置，再根据大小向下调整。

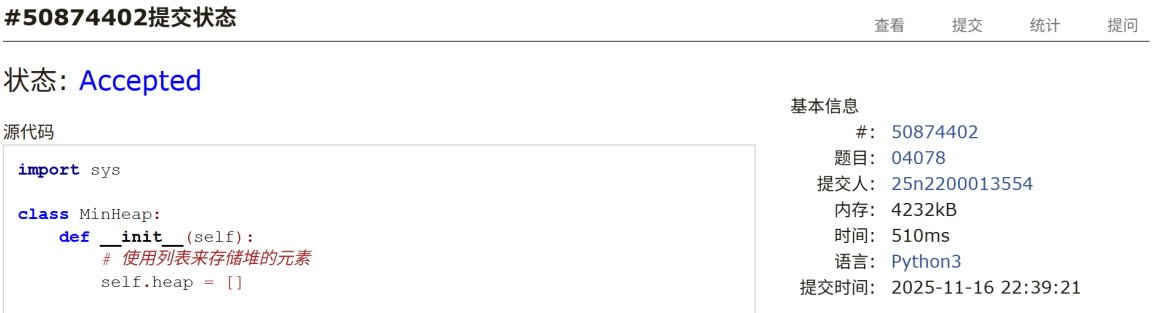
代码：

```
import sys

class MinHeap:
    def __init__(self):
        # 使用列表来存储堆的元素
        self.heap = []

    def _parent(self, i):
        return (i - 1) // 2

    def _left_child(self, i):
        return 2 * i + 1

    def _right_child(self, i):
        return 2 * i + 2

    def _swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def _sift_up(self, i):
        """
        向上调整，将索引 i 的元素移动到正确的位置
        """
        # 当 i 不是根节点且当前节点比父节点小时，继续向上
        while i > 0 and self.heap[i] < self.heap[self._parent(i)]:
            parent_idx = self._parent(i)
            self._swap(i, parent_idx)
            i = parent_idx
```

```
    def _sift_down(self, i):
        """
        向下调整，将索引 i 的元素移动到正确的位置
        """
        max_index = len(self.heap) - 1

        while True:
            left_idx = self._left_child(i)
            right_idx = self._right_child(i)
            smallest = i

            # 找出当前节点和其左右子节点中的最小值索引
            if left_idx <= max_index and self.heap[left_idx] <
self.heap[smallest]:
                smallest = left_idx

            if right_idx <= max_index and self.heap[right_idx] <
self.heap[smallest]:
                smallest = right_idx

            # 如果当前节点就是最小的，则调整结束
            if smallest == i:
                break

            # 否则，与最小的子节点交换，并继续向下调整
            self._swap(i, smallest)
            i = smallest

    def push(self, value):
        """
        增添一个元素
        """
        self.heap.append(value)
        self._sift_up(len(self.heap) - 1)

    def pop(self):
        """
        输出并删除最小的元素
        """
        if not self.heap:
            return None # 或者抛出异常

        size = len(self.heap)

        # 如果只有一个元素，直接弹出
        if size == 1:
            return self.heap.pop()

        # 1. 保存根节点（最小值）
        min_value = self.heap[0]

        # 2. 将最后一个元素移动到根部
        self.heap[0] = self.heap.pop()

        # 3. 向下调整新的根节点
        self._sift_down(0)
```

```
        return min_value

    def is_empty(self):
        return len(self.heap) == 0


def solve():
    """
    主解决函数，处理输入输出
    """
    try:
        n = int(sys.stdin.readline())
        my_heap = MinHeap()

        for _ in range(n):
            line = sys.stdin.readline().split()
            op_type = int(line[0])

            if op_type == 1:
                u = int(line[1])
                my_heap.push(u)
            elif op_type == 2:
                if not my_heap.is_empty():
                    min_val = my_heap.pop()
                    print(min_val)

    except (IOError, ValueError):
        pass

# 调用主函数
solve()
```

代码运行截图（至少包含有"Accepted"）



#50874402提交状态

查看 提交 统计 提问

状态：Accepted

源代码

```
import sys

class MinHeap:
    def __init__(self):
        # 使用列表来存储堆的元素
        self.heap = []
```

基本信息

#:	50874402
题目:	04078
提交人:	25n2200013554
内存:	4232kB
时间:	510ms
语言:	Python3
提交时间:	2025-11-16 22:39:21

2. 学习总结和个人收获

N皇后问题非常经典，题解部分有许多讨论，在阅读题解中学到很多，我使用的是基于集合的回溯方法，题解中针对位运算的回溯方法虽然降低了记录皇后信息的空间复杂度，但是用二进制表示中的一个数位这部分的设计比较复杂。本次作业出了两个和上次月考类似的二叉树问题，感觉目前解决起来更熟练了。printExp题目比较抽象，一开始没有想到怎么建树，后续发现通过树的方式解决这个问题非常巧妙，感觉可以和之前写过的括号问题放在一起比较。手搓堆还是很有挑战，需要再熟悉熟悉。