Assignment #7: bfs、 🌲

Updated 0851 GMT+8 Oct 21, 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

M23555: 节省存储的矩阵乘法（50分钟）

implementation, matrices, http://cs101.openjudge.cn/practice/23555

要求用节省内存的方式实现，不能还原矩阵的方式实现。

思路：需要直接在三元组（row, col, value）这种表示形式上进行运算。Result[i] [j] = Σ (A[i] [k] * B[k] [j])，对于输入的两个矩阵，找到相同的k对应的元素进行计算算出不为0的结果矩阵的元素，再按照三元组的形式输出。

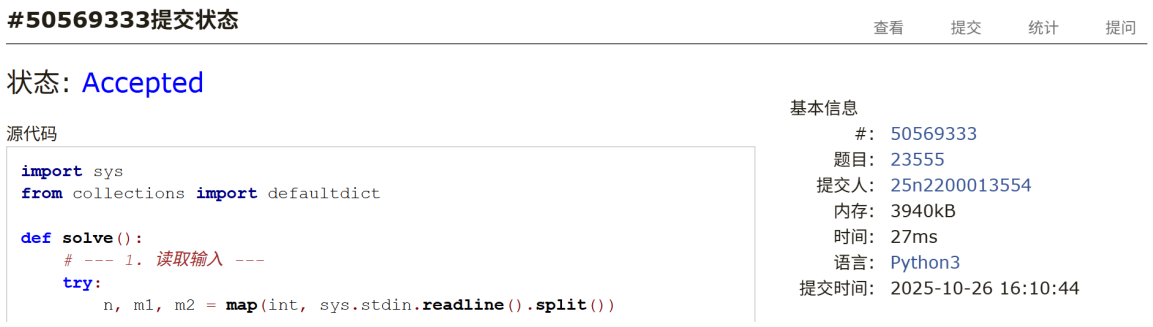
代码：

```
import sys
from collections import defaultdict

def solve():
    # --- 1. 读取输入 ---
    try:
        n, m1, m2 = map(int, sys.stdin.readline().split())

        # 读取矩阵 X 的三元组
        triplets_X = []
        for _ in range(m1):
```

```
            triplets_X.append(list(map(int, sys.stdin.readline().split())))

        # 读取矩阵 Y 的三元组
        triplets_Y = []
        for _ in range(m2):
            triplets_Y.append(list(map(int, sys.stdin.readline().split())))

    except (ValueError, IndexError):
        return

    # --- 2. 预处理数据结构 ---
    # X 按行分组: {row: [(col, val), ...]}
    X_rows = defaultdict(list)
    for r, c, v in triplets_X:
        X_rows[r].append((c, v))

    # Y 按列分组: {col: [(row, val), ...]}
    Y_cols = defaultdict(list)
    for r, c, v in triplets_Y:
        Y_cols[c].append((r, v))

    # 为了高效合并，对每个列表进行排序
    for r in X_rows:
        X_rows[r].sort() # 按 col 排序
    for c in Y_cols:
        Y_cols[c].sort() # 按 row 排序

    # --- 3. 计算乘积 ---
    result_triplets = []

    # 遍历所有可能产生非零结果的 (i, j)
    # i 是 X 的行号, j 是 Y 的列号
    sorted_X_rows = sorted(X_rows.keys())
    sorted_Y_cols = sorted(Y_cols.keys())

    for i in sorted_X_rows:
        for j in sorted_Y_cols:
            row_A = X_rows[i]
            col_B = Y_cols[j]

            current_sum = 0
            ptr_A, ptr_B = 0, 0

            # 双指针法合并
            while ptr_A < len(row_A) and ptr_B < len(col_B):
                k_A, val_A = row_A[ptr_A] # k_A 是列号
                k_B, val_B = col_B[ptr_B] # k_B 是行号

                if k_A == k_B:
                    current_sum += val_A * val_B
                    ptr_A += 1
                    ptr_B += 1
                elif k_A < k_B:
                    ptr_A += 1
                else: # k_A > k_B
                    ptr_B += 1
```

```
            # 如果结果非零，则添加到结果列表
            if current_sum != 0:
                result_triplets.append((i, j, current_sum))

    # --- 4. 排序和输出 ---
    # result_triplets 已经是按先行号后列号排序的
    # 如果外层循环顺序不确定，则需要下面这行排序
    # result_triplets.sort()

    for r, c, v in result_triplets:
        print(r, c, v)

# 调用主函数
solve()
```

代码运行截图 （至少包含有"Accepted"）



M102.二叉树的层序遍历（25分钟）

bfs, https://leetcode.cn/problems/binary-tree-level-order-traversal/

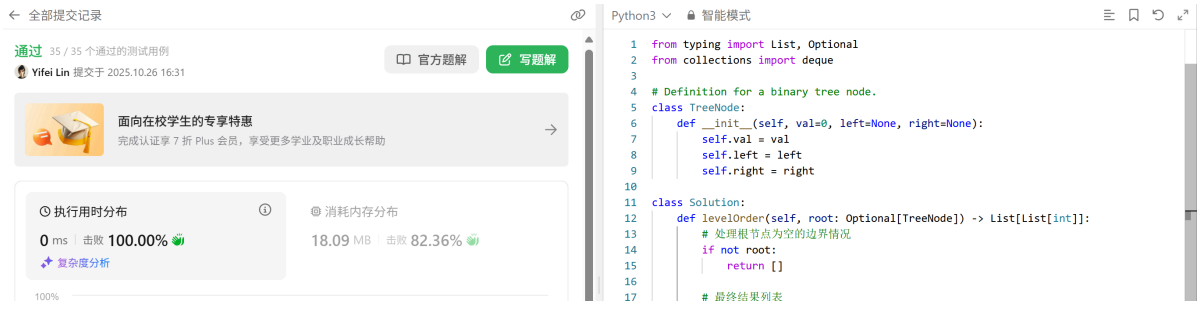
思路：逐层从左到右访问节点。注意边界条件，如果根节点 root 为空，直接返回一个空列表 []。

代码：

```
from typing import List, Optional
from collections import deque

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        # 处理根节点为空的边界情况
        if not root:
            return []
```

```
        # 最终结果列表
        result = []
        # 使用双端队列 collections.deque 实现高效的队列操作
        queue = deque([root])

        # 当队列不为空时，循环处理
        while queue:
            # 当前层的节点数量
            level_size = len(queue)
            # 用于存放当前层所有节点值的列表
            current_level = []

            # 遍历当前层的所有节点
            for _ in range(level_size):
                # 从队列左侧（头部）取出一个节点
                node = queue.popleft()

                # 将节点值加入当前层列表
                current_level.append(node.val)

                # 如果有左子节点，将其加入队列
                if node.left:
                    queue.append(node.left)

                # 如果有右子节点，将其加入队列
                if node.right:
                    queue.append(node.right)

            # 将处理完的当前层加入最终结果
            result.append(current_level)

        return result
```

代码运行截图 （至少包含有"Accepted"）



M131.分割回文串（30分钟）

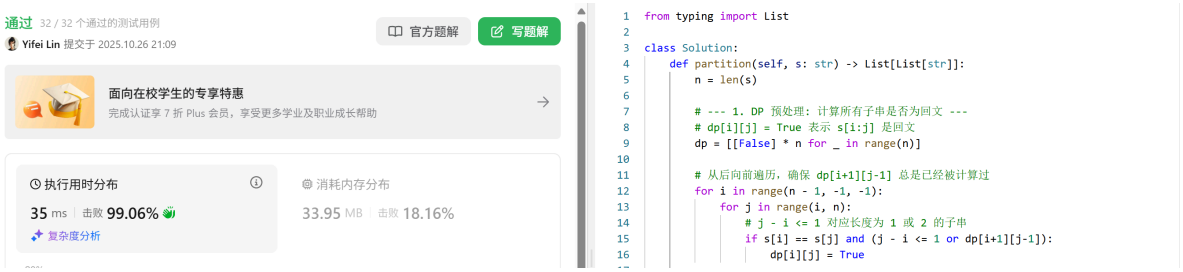
dp, backtracking, https://leetcode.cn/problems/palindrome-partitioning/

思路：刚开始看这个题目的时候在纠结子串的定义，后面发现网站子串给了定义是字符串中连续的非空字符序列，说明不用考虑字符随机排列的问题，只需要考虑隔断的情况。判断字符串是不是回文时，使用动态规划，回文字符串要求去除第一个和最后一个相同的字符后内部依然是一个回文字符串或不剩下字符。

代码：

```
from typing import List

class Solution:
    def partition(self, s: str) -> List[List[str]]:
        n = len(s)

        # --- 1. DP 预处理：计算所有子串是否为回文 ---
        # dp[i][j] = True 表示 s[i:j] 是回文
        dp = [[False] * n for _ in range(n)]

        # 从后向前遍历，确保 dp[i+1][j-1] 总是已经被计算过
        for i in range(n - 1, -1, -1):
            for j in range(i, n):
                # j - i <= 1 对应长度为 1 或 2 的子串
                if s[i] == s[j] and (j - i <= 1 or dp[i+1][j-1]):
                    dp[i][j] = True

        # --- 2. 回溯 ---
        result = []
        path = []

        def backtrack(start_index: int):
            # 终止条件：已经分割到字符串末尾
            if start_index == n:
                result.append(path[:])
                return

            # 遍历所有可能的切点
            for i in range(start_index, n):
                # 如果 s[start_index:i+1] 是回文
                if dp[start_index][i]:
                    # 做出选择
                    path.append(s[start_index : i+1])

                    # 进入下一层决策
                    backtrack(i + 1)

                    # 撤销选择
                    path.pop()

        # 初始调用
        backtrack(0)
        return result
```

代码运行截图 （至少包含有"Accepted"）



M200.岛屿数量（50分钟）

dfs, bfs, https://leetcode.cn/problems/number-of-islands/

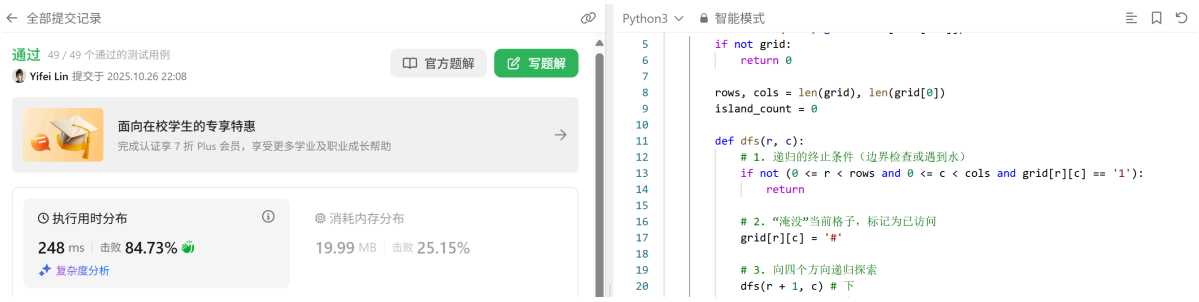
思路：遍历网格，每找到一块未被标记的陆地，就将计数器加一，并用搜索算法将与这块陆地相连的所有部分都标记掉。都是找到一个岛屿的所有部分，深度优先搜索 (DFS)使用递归，代码通常更简洁。但如果岛屿非常大且形状狭长，可能会导致递归深度过大，有栈溢出的风险。广度优先搜索 (BFS)使用队列，是迭代式的。没有递归深度的问题，更稳健。在寻找最短路径问题时，BFS是首选。

深度优先搜索 (DFS)代码

```
from typing import List

class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid:
            return 0

        rows, cols = len(grid), len(grid[0])
        island_count = 0

        def dfs(r, c):
            # 1. 递归的终止条件（边界检查或遇到水）
            if not (0 <= r < rows and 0 <= c < cols and grid[r][c] == '1'):
                return

            # 2. “淹没”当前格子，标记为已访问
            grid[r][c] = '#'

            # 3. 向四个方向递归探索
            dfs(r + 1, c) # 下
            dfs(r - 1, c) # 上
            dfs(r, c + 1) # 右
            dfs(r, c - 1) # 左

        # --- 主循环 ---
        for r in range(rows):
            for c in range(cols):
                # 如果找到一个新岛屿的“一角”
                if grid[r][c] == '1':
                    island_count += 1
                    # 从这个点开始，淹没整个岛屿
                    dfs(r, c)
```

```
        return island_count
```

（至少包含有"Accepted"）

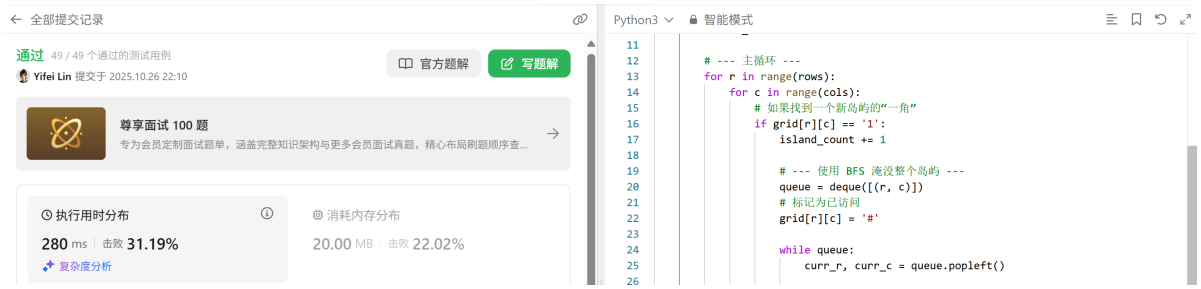


广度优先搜索 (BFS)代码

```
from typing import List
from collections import deque

class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid:
            return 0

        rows, cols = len(grid), len(grid[0])
        island_count = 0

        # --- 主循环 ---
        for r in range(rows):
            for c in range(cols):
                # 如果找到一个新岛屿的“一角”
                if grid[r][c] == '1':
                    island_count += 1

                    # --- 使用 BFS 淹没整个岛屿 ---
                    queue = deque([(r, c)])
                    # 标记为已访问
                    grid[r][c] = '#'

                    while queue:
                        curr_r, curr_c = queue.popleft()

                        # 探索四个方向
                        for dr, dc in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                            next_r, next_c = curr_r + dr, curr_c + dc

                            # 检查边界和是否为陆地
                            if (0 <= next_r < rows and
                                0 <= next_c < cols and
                                grid[next_r][next_c] == '1'):

                                queue.append((next_r, next_c))
                                # 标记为已访问
                                grid[next_r][next_c] = '#'
```

```
        return island_count
```

（至少包含有"Accepted"）



1123.最深叶节点的最近公共祖先（30分钟）

dfs, https://leetcode.cn/problems/lowest-common-ancestor-of-deepest-leaves/

思路：其中需要注意如果一个分支下面只有一个元素，另一个元素为null，则这个元素的最近祖先为它自己而不是上一级。当左右子树的深度一样时，最近祖先就是这个节点。

代码

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def lcaDeepestLeaves(self, root: TreeNode) -> TreeNode:

        def dfs(node: TreeNode):
            """
            返回一个元组 (depth, lca_node)。
            depth: 以 node 为根的子树中，最深叶子的深度。
            lca_node: 以 node 为根的子树中，包含其所有最深叶子的最小子树的根。
            """
            # 基本情况：空节点
            if not node:
                return 0, None

            # 递归探索左右子树 (后序遍历)
            left_depth, left_lca = dfs(node.left)
            right_depth, right_lca = dfs(node.right)

            # 比较左右子树的深度
            if left_depth > right_depth:
                # 最深叶子只在左子树
                return left_depth + 1, left_lca
            elif right_depth > left_depth:
```

```
            # 最深叶子只在右子树
            return right_depth + 1, right_lca
        else: # left_depth == right_depth
            # 最深叶子在左右子树中都有，或者 node 本身就是叶子
            # 此时，当前节点 node 就是 lca
            return left_depth + 1, node

    # 初始调用，我们只需要返回 lca_node
    _, lca_node = dfs(root)
    return lca_node
```

（至少包含有"Accepted"）


```
    返回一个元组 (depth, lca_node)。
    depth: 以 node 为根的子树中，最深叶子的深度。
    lca_node: 以 node 为根的子树中，包含其所有最深叶子的最小子树的根。
    """
    # 基本情况：空节点
    if not node:
        return 0, None

    # 递归探索左右子树（后序遍历）
    left_depth, left_lca = dfs(node.left)
    right_depth, right_lca = dfs(node.right)

    # 比较左右子树的深度
    if left_depth > right_depth:
        # 最深叶子只在左子树
        return left_depth + 1, left_lca
    elif right_depth > left_depth:
        # 最深叶子只在右子树
        return right_depth + 1, right_lca
    else: # left_depth == right_depth
        # 最深叶子在左右子树中都有，或者 node 本身就是叶子
        # 此时，当前节点 node 就是 lca
        return left_depth + 1, node

# 初始调用，我们只需要返回 lca_node
```

M79.单词搜索（30分钟）

回溯，https://leetcode.cn/problems/word-search/

思路：需要双重循环，既需要遍历每一个格子找到第一个字母，也需要在找到第一个字母后遍历上下左右直至找到整个单词。

代码：

```
from typing import List

class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        if not board:
            return False

        rows, cols = len(board), len(board[0])

        def backtrack(r: int, c: int, k: int) -> bool:
            # 1. 失败终止条件：越界或字符不匹配
            if not (0 <= r < rows and 0 <= c < cols and board[r][c] == word[k]):
                return False

            # 2. 成功终止条件：word 的所有字符都已匹配
            if k == len(word) - 1:
                return True
```

```
            # 3. 做出选择：标记当前格子为已访问
            # 保存原始字符，以便回溯
            temp, board[r][c] = board[r][c], '#'

            # 4. 探索相邻格子 (上, 下, 左, 右)
            found = (backtrack(r + 1, c, k + 1) or
                     backtrack(r - 1, c, k + 1) or
                     backtrack(r, c + 1, k + 1) or
                     backtrack(r, c - 1, k + 1))

            # 5. 撤销选择：恢复当前格子
            board[r][c] = temp

            return found

        # --- 主循环：遍历所有可能的起点 ---
        for r in range(rows):
            for c in range(cols):
                # 如果以 (r, c) 为起点可以找到 word，则立即返回 True
                if backtrack(r, c, 0):
                    return True

        # 如果所有起点都尝试失败，则返回 False
        return False
```

代码运行截图 （至少包含有"Accepted"）

2. 学习总结和个人收获

感觉第一个问题很有难度，可能我对矩阵的相关运算还是很不熟练（感觉可能是受到数学的限制），在题目的理解上就花了很多时间，可能需要再复习一下相关的数学课程。M200.岛屿数量我尝试了dfs和bfs两种方法进行操作，感觉在做题中对这两种解法的思路有了更好的体会和辨析。题解里还提供了使用并查集代替搜索的方法，但是感觉比较复杂且时空复杂度都不如前两种方法。