Assignment #B: 图 (1/4)

Updated 2031 GMT+8 Nov 17, 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

E07218: 献给阿尔吉侬的花束（30分钟）

bfs, http://cs101.openjudge.cn/practice/07218/

思路：本题与第一次月考的逃出紫罗兰监狱类似，但是没有闪现技能会好处理很多，即在到达一个点位遇到#为墙壁可以直接判断不可行而不是判断是否还有闪现机会再讨论是否可行。

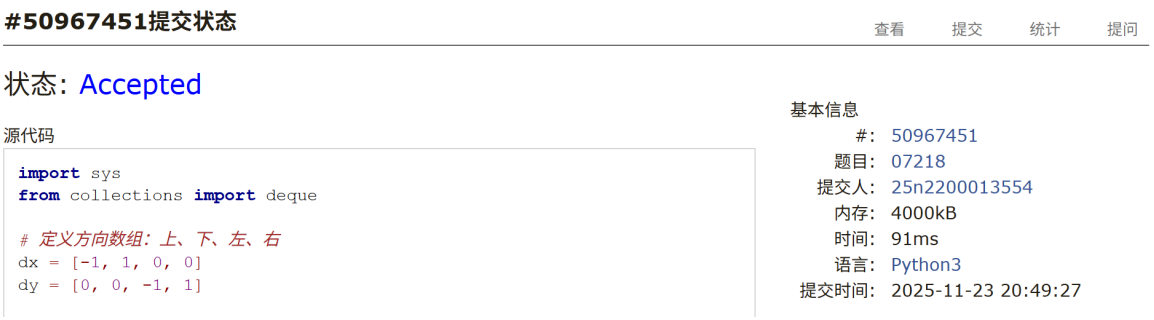
代码：

```
import sys
from collections import deque

# 定义方向数组：上、下、左、右
dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]

def bfs(start_node, end_node, grid, R, C):
    # 队列中存储元组：(行坐标, 列坐标, 当前耗时)
    queue = deque([(start_node[0], start_node[1], 0)])

    # 访问记录，防止重复走
    visited = [[False for _ in range(C)] for _ in range(R)]
    visited[start_node[0]][start_node[1]] = True
```

```
    while queue:
        x, y, time = queue.popleft()

        # 如果到达终点（注意：题目中S和E只是位置标记，E点本身是可以走的）
        if (x, y) == end_node:
            return time

        # 尝试四个方向
        for i in range(4):
            nx, ny = x + dx[i], y + dy[i]

            # 检查边界
            if 0 <= nx < R and 0 <= ny < C:
                # 检查是否是墙壁 且 没有访问过
                if grid[nx][ny] != '#' and not visited[nx][ny]:
                    visited[nx][ny] = True
                    queue.append((nx, ny, time + 1))

    return "oop!"

def solve():
    # 读取所有输入，按空白字符分割
    input_data = sys.stdin.read().split()

    if not input_data:
        return

    iterator = iter(input_data)

    try:
        T = int(next(iterator))  # 读取测试组数
    except StopIteration:
        return

    for _ in range(T):
        try:
            R = int(next(iterator))
            C = int(next(iterator))
        except StopIteration:
            break

        grid = []
        start_node = None
        end_node = None

        # 构建网格并寻找起点和终点
        for r in range(R):
            row = next(iterator)
            grid.append(row)
            for c in range(C):
                if row[c] == 'S':
                    start_node = (r, c)
                elif row[c] == 'E':
                    end_node = (r, c)
```

```
        # 执行BFS并输出结果
        result = bfs(start_node, end_node, grid, R, C)
        print(result)

if __name__ == "__main__":
    solve()
```

代码运行截图 （至少包含有"Accepted"）



M27925: 小组队列（50分钟）

dict, queue, http://cs101.openjudge.cn/practice/27925/

思路：使用双层队列，一个队列存放小组顺序，一个字典用来存放每个小组内部的排列顺序，对于散客，分配一个唯一的虚拟小组ID并加入成员归属表。

代码：

```
import sys
from collections import deque

def solve():
    # 使用 sys.stdin 读取所有数据，处理速度快
    # split('\n') 按行分割，保留行的结构以便区分小组定义
    input_lines = sys.stdin.read().split('\n')

    iterator = iter(input_lines)

    try:
        # 读取小组数量 t
        # 过滤掉可能存在的空行或无关行，直到读到数字
        line = next(iterator)
        while not line.strip() or not line.strip()[0].isdigit():
            line = next(iterator)
        t = int(line.strip())
    except StopIteration:
        return

    # 1. 建立成员到小组的映射
    group_mapping = {}
```

```
    for i in range(t):
        try:
            line = next(iterator)
            while not line.strip(): # 跳过空行
                line = next(iterator)
            # 读取该行的成员，映射到小组ID i
            members = list(map(int, line.strip().split()))
            for member in members:
                group_mapping[member] = i
        except StopIteration:
            break

    # 2. 初始化队列结构
    main_queue = deque()       # 存放 group_id
    sub_queues = {}            # 存放 group_id -> deque([member_id, ...])
    groups_in_queue = set()    # 快速判断某小组是否已在排队

    # 3. 处理指令
    for line in iterator:
        parts = line.strip().split()
        if not parts:
            continue

        cmd = parts[0]

        if cmd == 'STOP':
            break

        elif cmd == 'ENQUEUE':
            person = int(parts[1])

            # 确定该人的小组ID
            if person in group_mapping:
                gid = group_mapping[person]
            else:
                # 散客处理：分配一个唯一的ID，使用 -person 避免与 0~(t-1) 冲突
                gid = -person

            # 如果该小组不在大队列中，先排大队
            if gid not in groups_in_queue:
                groups_in_queue.add(gid)
                main_queue.append(gid)
                sub_queues[gid] = deque()

            # 然后加入小队
            sub_queues[gid].append(person)

        elif cmd == 'DEQUEUE':
            if not main_queue:
                continue

            # 找到排在最前面的小组
            first_gid = main_queue[0]

            # 只要大队列有这个组，说明小队列一定有值
            person = sub_queues[first_gid].popleft()
```

```
            print(person)

            # 如果这个小组的人走光了
            if not sub_queues[first_gid]:
                # 从大队列移除该小组
                main_queue.popleft()
                groups_in_queue.remove(first_gid)
                del sub_queues[first_gid] # 节省内存

if __name__ == "__main__":
    solve()
```

代码运行截图（至少包含有"Accepted"）

#50968542提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
import sys
from collections import deque

def solve():
    # 使用 sys.stdin 读取所有数据，处理速度快
    # split('\n') 按行分割，保留行的结构以便区分小组定义
    input_lines = sys.stdin.read().split('\n')
```

基本信息

#:	50968542
题目:	27925
提交人:	25n2200013554
内存:	7120kB
时间:	44ms
语言:	Python3
提交时间:	2025-11-23 21:30:49

M04089: 电话号码（40分钟）

trie, http://cs101.openjudge.cn/practice/04089/

思路：当我们把一个号码插入 Trie 树时，如果出现以下两种情况之一，说明存在前缀冲突（不一致）：

（1）路过已有的结束标记：
例如：Trie 中已有 911（在第二个 1 处标记为结束）。
插入 91125 时，我们会经过 9 -> 1 -> 1。发现第二个 1 是 is_end，说明 911 是 91125 的前缀。
结论：正在插入的长号码包含了一个已有的短号码。
（2）插入结束时没有开辟新路径：
例如：Trie 中已有 91125。
插入 911 时，路径 9 -> 1 -> 1 已经存在了，我们没有创建任何新节点就结束了插入。
结论：正在插入的短号码是已有长号码的前缀。

代码：

```
import sys

# 定义 Trie 节点
class TrieNode:
    # 使用 __slots__ 限制属性，减少内存占用，提高访问速度
    __slots__ = ('children', 'is_end')

    def __init__(self):
        self.children = {}  # 字典映射：字符 -> 子节点
        self.is_end = False # 标记是否是某个电话号码的结尾
```

```
class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, number):
        """
        插入号码，同时检查一致性。
        返回 False 表示发现冲突（不一致），返回 True 表示目前安全。
        """
        node = self.root
        # 标记1：在插入过程中，是否经过了别的号码的结尾？
        passed_end = False
        # 标记2：在插入过程中，是否开辟了新的分支？
        created_new_branch = False

        for char in number:
            if char not in node.children:
                node.children[char] = TrieNode()
                created_new_branch = True

            node = node.children[char]

            # 如果当前路过的节点是之前某个号码的结尾，说明之前有个短号码是当前号码的前缀
            if node.is_end:
                passed_end = True

        # 标记当前节点为号码结尾
        node.is_end = True

        # 冲突判断逻辑：
        # 1. 如果插入过程中路过了别人的结尾 (passed_end is True) -> 冲突
        # 2. 如果插入结束了完全没有开辟新分支 (created_new_branch is False)
        if passed_end or not created_new_branch:
            return False # 不一致

        return True # 一致

def solve():
    input_data = sys.stdin.read().split()

    if not input_data:
        return

    iterator = iter(input_data)

    try:
        t = int(next(iterator))
    except StopIteration:
        return

    for _ in range(t):
        try:
            n = int(next(iterator))
            numbers = []
            for _ in range(n):
```

```
                numbers.append(next(iterator))

            # 构建 Trie 树并检查
            trie = Trie()
            is_consistent = True

            for num in numbers:
                # 如果插入返回 False，说明不一致
                if not trie.insert(num):
                    is_consistent = False
                    # 找到一个反例即可停止当前组的判断，但为了逻辑简单，
                    # 这里直接 break。由于数据已经全部读入 numbers 列表，
                    # break 不会影响下一组数据的读取。
                    break

            if is_consistent:
                print("YES")
            else:
                print("NO")

        except StopIteration:
            break

if __name__ == "__main__":
    solve()
```

代码运行截图 （至少包含有"Accepted"）

#50969468提交状态

查看 提交 统计 提问

状态：Accepted

源代码

```
import sys

# 定义 Trie 节点
class TrieNode:
    # 使用 __slots__ 限制属性，减少内存占用，提高访问速度
    __slots__ = ('children', 'is_end')
```

基本信息

#: 50969468
题目: 04089
提交人: 25n2200013554
内存: 19724kB
时间: 193ms
语言: Python3
提交时间: 2025-11-23 22:11:48

M3532.针对图的路径存在性查询I（50分钟）

disjoint set, https://leetcode.cn/problems/path-existence-queries-in-a-graph-i/

思路：使用并查集。在一个已排序的数组中，如果 nums[i]和 nums[j] (i<ji<j) 能够连通，那么它们之间的所有中间节点一定也能连通。可以基于此进行联通节点的合并。

代码

```
from typing import List

class UnionFind:
    def __init__(self, n):
```

```
        # 初始化 parent 数组，每个节点的父节点初始为自己
        self.parent = list(range(n))

    def find(self, i):
        # 路径压缩查找：在查找的同时将路径上的节点直接指向根节点
        if self.parent[i] != i:
            self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def union(self, i, j):
        # 合并两个集合
        root_i = self.find(i)
        root_j = self.find(j)
        if root_i != root_j:
            self.parent[root_i] = root_j
            return True
        return False

class Solution:
    def pathExistenceQueries(self, n: int, nums: List[int], maxDiff: int,
queries: List[List[int]]) -> List[bool]:
        # 1. 初始化并查集
        uf = UnionFind(n)

        # 2. 遍历数组，只检查相邻元素的差值
        # 因为 nums 是有序的，只要相邻的能连通，跨越的也能通过传递性连通
        for i in range(n - 1):
            if nums[i+1] - nums[i] <= maxDiff:
                uf.union(i, i+1)

        # 3. 处理查询
        answer = []
        for u, v in queries:
            # 检查 u 和 v 是否在同一个集合中（即根节点是否相同）
            if uf.find(u) == uf.find(v):
                answer.append(True)
            else:
                answer.append(False)

        return answer
```

代码运行截图（至少包含有"Accepted"）

M19943: 图的拉普拉斯矩阵（30分钟）

OOP, graph, implementation, http://cs101.openjudge.cn/pctbook/E19943/

要求创建Graph, Vertex两个类，建图实现。

思路：用面向对象方式将图抽象为顶点和图的类，通过邻接表存储连接关系，最后遍历每个顶点及其邻居来构建拉普拉斯矩阵。

代码

```
import sys

# 1. 定义 Vertex 类
class Vertex:
    def __init__(self, key):
        self.id = key
        # 使用字典存储邻居，键是邻居Vertex对象，值是权重(虽然本题权重没用)
        self.connectedTo = {}

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in
self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getDegree(self):
        return len(self.connectedTo)

# 2. 定义 Graph 类
class Graph:
```

```
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self, key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self, n):
        return n in self.vertList

    def addEdge(self, f, t, cost=0):
        # 确保两个点都在图中
        if f not in self.vertList:
            self.addVertex(f)
        if t not in self.vertList:
            self.addVertex(t)

        # 无向图：双方互为邻居
        self.vertList[f].addNeighbor(self.vertList[t], cost)
        self.vertList[t].addNeighbor(self.vertList[f], cost)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())

    # 核心方法：生成拉普拉斯矩阵
    def generateLaplacianMatrix(self, n):
        # 初始化 n x n 矩阵，全为 0
        matrix = [[0] * n for _ in range(n)]

        # 遍历 0 到 n-1 的每个节点
        for i in range(n):
            v = self.getVertex(i)
            if v:
                # 1. 设置对角线元素 D_ii = Degree(i)
                matrix[i][i] = v.getDegree()

                # 2. 设置非对角线元素 L_ij = -1 (如果相连)
                # 获取 v 的所有邻居对象
                for neighbor in v.getConnections():
                    neighbor_id = neighbor.getId()
                    matrix[i][neighbor_id] = -1

        return matrix
```

```python
# 3. 主程序逻辑
def solve():
    # 读取所有输入
    input_data = sys.stdin.read().split()

    if not input_data:
        return

    iterator = iter(input_data)

    try:
        n_str = next(iterator)
        m_str = next(iterator)
        n = int(n_str)
        m = int(m_str)
    except StopIteration:
        return

    # 实例化图
    g = Graph()

    # 关键步骤：先添加所有 0 到 n-1 的顶点
    # 这是为了防止某些顶点是孤立点（没有边连接），导致在 addEdge 时才被创建，
    # 或者如果图中没有点，我们要保证矩阵大小正确。
    for i in range(n):
        g.addVertex(i)

    # 读取边并构建图
    for _ in range(m):
        u = int(next(iterator))
        v = int(next(iterator))
        g.addEdge(u, v)

    # 获取矩阵
    laplacian = g.generateLaplacianMatrix(n)

    # 输出矩阵
    for row in laplacian:
        print(*row)

if __name__ == "__main__":
    solve()
```

代码运行截图（至少包含有"Accepted"）

#50971074提交状态

查看　提交　统计　提问

状态: Accepted

源代码

```python
import sys

# 1. 定义 Vertex 类
class Vertex:
    def __init__(self, key):
        self.id = key
        # 使用字典存储邻居，键是邻居Vertex对象，值是权重(虽然本题权重没用)
        self.connectedTo = {}
```

基本信息

#: 50971074
题目: E19943
提交人: 25n2200013554
内存: 3772kB
时间: 24ms
语言: Python3
提交时间: 2025-11-24 00:51:18

思路：对于输出矩阵，对角线元素Lii为节点i的度数，非对角线元素Lij如果i和j之间有边则为-1，无边则为0。

代码

```
import sys

def solve():
    # 使用 sys.stdin.read 读取所有输入，以应对较大的数据量
    input_data = sys.stdin.read().split()

    if not input_data:
        return

    iterator = iter(input_data)

    try:
        # 读取节点数 n 和边数 m
        n = int(next(iterator))
        m = int(next(iterator))
    except StopIteration:
        return

    # 1. 初始化 n x n 的矩阵，所有元素默认为 0
    #matrix[row][col]
    matrix = [[0] * n for _ in range(n)]

    # 2. 遍历每一条边，更新矩阵
    for _ in range(m):
        try:
            u = int(next(iterator))
            v = int(next(iterator))

            # 更新对角线：度数 +1
            matrix[u][u] += 1
            matrix[v][v] += 1

            # 更新非对角线：有边相连的位置设为 -1
            # L = D - A，由于 Aij 为 1，所以 Lij 为 -1
            matrix[u][v] = -1
            matrix[v][u] = -1

        except StopIteration:
            break

    # 3. 输出结果
    for row in matrix:
        # 使用 * 解包列表，默认以空格分隔打印
        print(*row)

if __name__ == "__main__":
    solve()
```

代码运行截图（至少包含有"Accepted"）

T25353: 排队（60分钟）

http://cs101.openjudge.cn/pctbook/T25353/

思路：节点数N高达 10^5。如果是稠密图（例如D很小，大部分人互斥），边的数量可能达到 O(N^2)，直接建图和计算入度会超时（TLE）并超内存（MLE），所以不能建图。此题与此前最小数类似，但是多了一个交换的限制条件，因此在逐个交换使最左边的值最小时需要进行判断，若无法继续交换则放弃此处跳至下一个数字处。如此确定了最左边之后再确实左边第二个，以此类推。但如果只是快排依然无法达到时间要求，所以参考群中的思路采取基准值比较的方法。

代码：

```
import sys

# 关键：增加递归深度限制
# 题目中 N 最大 10^5，Python 默认递归深度只有 1000，不设置会报错 RecursionError
sys.setrecursionlimit(200000)

def quick_sort_constrained(d, num_list):
    # 递归终止条件：列表为空
    if not num_list:
        return []

    # 1. 选取基准：队伍的第一名同学
    pivot = num_list[0]

    left = []   # 存放能排到 pivot 左边的同学
    right = []  # 存放只能排在 pivot 右边的同学

    # 初始化阻挡区间的最大值和最小值
    # 最开始阻挡区间里只有 pivot 一个人
    max_blocker = pivot
    min_blocker = pivot

    # 2. 遍历剩下的人，进行“划分”
    for num in num_list[1:]:
        # 判断能否去左边：
        # 条件1: 必须比 pivot 小 (字典序要求)
        # 条件2: 必须能跟“阻挡组”里最高的人交换 (max_blocker - num <= d)
```

```
        # 条件3: 必须能跟“阻挡组”里最矮的人交换 (num - min_blocker <= d)
        if num < pivot and (max_blocker - num <= d) and (num - min_blocker <= d):
            left.append(num)
        else:
            # 去右边，并加入“阻挡组”
            right.append(num)
            # 更新阻挡组的最值
            if num > max_blocker:
                max_blocker = num
            if num < min_blocker:
                min_blocker = num

    # 3. 递归合并结果：左边排好序 + [基准] + 右边排好序
    return quick_sort_constrained(d, left) + [pivot] + quick_sort_constrained(d,
right)

def solve():
    # 使用 sys.stdin.read 快速读取所有输入
    input_data = sys.stdin.read().split()

    if not input_data:
        return

    iterator = iter(input_data)
    try:
        N = int(next(iterator))
        D = int(next(iterator))
        # 读取剩余的 N 个身高数据
        heights = [int(next(iterator)) for _ in range(N)]
    except StopIteration:
        return

    # 调用排序函数
    result = quick_sort_constrained(D, heights)

    # 输出结果
    print('\n'.join(map(str, result)))

if __name__ == "__main__":
    solve()
```

代码运行截图 （至少包含有"Accepted"）

#50971078提交状态

查看　提交　统计　提问

状态: Accepted

源代码

```
import sys

# 关键：增加递归深度限制
# 题目中 N 最大 10^5，Python 默认递归深度只有 1000，不设置会报错 RecursionErro
sys.setrecursionlimit(200000)

def quick_sort_constrained(d, num_list):
    # 递归终止条件：列表为空
    if not num_list:
        return []
```

基本信息

#:	50971078
题目:	T25353
提交人:	25n2200013554
内存:	60308kB
时间:	1512ms
语言:	Python3
提交时间:	2025-11-24 00:52:59

2. 学习总结和个人收获

本次作业中有几处是此前作业的变式，有此前的经验之后再理解起来会更熟练，说明有空还是要多练习。本次作业涉及图的表示、图的遍历、连通性判断以及依赖关系处理，结合题目与讲义再次熟悉了相关知识，但还是不太熟练，还需要多多加强。