Assignment #D: Mock Exam

Updated 1955 GMT+8 Dec 5, 2025

2025 fall, Complied by 林奕妃、环境科学与工程学院

> **说明：**
>
> 1. Dec月考： AC2（请改为同学的通过数）。考试题目都在"题库（包括计概、数算题目）"里面，按照数字题号能找到，可以重新提交。作业中提交自己最满意版本的代码和截图。
> 2. 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。
> 3. 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。
> 4. 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

E02734:十进制到八进制 （10分钟）

http://cs101.openjudge.cn/practice/02734

思路：使用Python自带功能进行进制转化。

代码

```
# 读取输入并转换为整数
a = int(input())

# 将整数转换为八进制并输出
# %o 是Python中用于格式化八进制输出的占位符
print("%o" % a)
```

代码运行截图（至少包含有"Accepted"）

#51162335提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
# 读取输入并转换为整数
a = int(input())

# 将整数转换为八进制并输出
# %o 是Python中用于格式化八进制输出的占位符
print("%o" % a)
```

基本信息

#: 51162335
题目: 02734
提交人: 25n2200013554
内存: 3600kB
时间: 26ms
语言: Python3
提交时间: 2025-12-06 16:19:23

English 帮助 关于

M21509:序列的中位数（15分钟）

heap, http://cs101.openjudge.cn/practice/21509

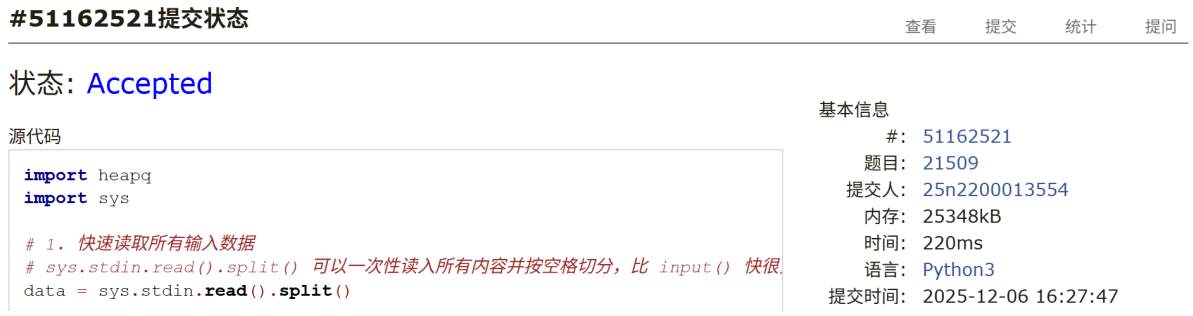
思路：如果使用简单的排序容易超时，因此采用堆的数据结构来处理。使用两个堆分别存放较小的和较大的数字，处理时根据要求将范围内数字划分成两部分，在两个堆之间交换使其储存的数字个数一致或较小的那个堆数字多一个。

代码

```
import heapq
import sys

# 1. 快速读取所有输入数据
# sys.stdin.read().split() 可以一次性读入所有内容并按空格切分，比 input() 快很多
data = sys.stdin.read().split()

# 转换第一个数据为 N
N = int(data[0])
# 剩下的数据就是序列 A，把它们都转成整数
nums = [int(x) for x in data[1:]]

# 2. 准备两个“堆”
# small: 存放较小的一半数字（大顶堆），Python默认是小顶堆，所以存负数来模拟
# large: 存放较大的一半数字（小顶堆）
small = []
large = []

results = []

# 3. 遍历每一个数字
for i in range(N):
    num = nums[i]

    # 核心逻辑：先把新数字放入 small 堆
    heapq.heappush(small, -num)

    # 为了保证 small 里的数都比 large 里的数小，
    # 我们把 small 里最大的数（-small[0]）拿出来，放到 large 里
    heapq.heappush(large, -heapq.heappop(small))

    # 平衡两个堆的数量
```

```
    # 规则：small 的数量允许比 large 多 1 个，但 large 不能比 small 多
    if len(small) < len(large):
        heapq.heappush(small, -heapq.heappop(large))

    # 4. 判断是否需要输出
    # 题目要求输出前1, 3, 5...个数的中位数
    # 在代码里，索引 i 是从 0 开始的 (0, 1, 2...)
    # 所以当 i 是偶数 (0, 2, 4...) 时，实际上处理了 1, 3, 5 个数
    if i % 2 == 0:
        # 因为 small 里的数更多（或相等），中位数就是 small 堆顶的数
        # 记得取反，因为我们存的是负数
        results.append(str(-small[0]))

# 打印结果，用换行符连接
print('\n'.join(results))
```

代码运行截图 （至少包含有"Accepted"）



M27306: 植物观察（40分钟）

disjoint set, bfs, http://cs101.openjudge.cn/practice/27306/

思路：题目要求对植物不能出现矛盾的判断。如果在推导过程中，发现某植物已经被判断，但根据新的线索产生不同的判断，这就产生了矛盾，说明结论不可能同时成立。

代码

```
import sys

def solve():
    # 1. 读取所有输入数据
    # 使用 sys.stdin.read() 一次性读入所有数据，处理速度快
    input_data = sys.stdin.read().split()

    if not input_data:
        return

    # 创建一个迭代器，方便一个个取数据
    iterator = iter(input_data)

    try:
        n = int(next(iterator)) # 植物数量
```

```
        m = int(next(iterator)) # 结论数量
    except StopIteration:
        return

    # 2. 构建“关系图”
    # adj 是一个列表，adj[x] 里面存着所有和植物 x 有关系的线索
    adj = [[] for _ in range(n)]

    for _ in range(m):
        u = int(next(iterator))
        v = int(next(iterator))
        w = int(next(iterator)) # w=0表示相同，w=1表示不同
        # 记录双向关系：u和v有关，v和u也有关
        adj[u].append((v, w))
        adj[v].append((u, w))

    # 3. 开始染色检查
    # colors 字典用来记录每棵植物的颜色（0 或 1）
    # 如果植物不在字典里，说明还没被检查过
    colors = {}

    # 遍历每一棵植物（防止图是不连通的，比如 0-1 有关，3-4 有关，但 1-3 没关）
    for i in range(n):
        if i not in colors:
            # 如果这棵植物还没被染色，我们先假设它是 0 (Type A)
            colors[i] = 0

            # 使用队列进行广度优先搜索 (BFS)
            queue = [i]

            while queue:
                curr = queue.pop(0) # 取出当前植物
                curr_color = colors[curr] # 当前植物的颜色

                # 查看所有和当前植物有关的线索
                for neighbor, relation in adj[curr]:
                    # 计算邻居理论上应该是什么颜色
                    # 如果关系是0(相同)，颜色应该一样
                    # 如果关系是1(不同)，颜色应该相反 (用 1 - curr_color 计算相反数)
                    expected_color = curr_color if relation == 0 else 1 -
curr_color

                    if neighbor in colors:
                        # 如果邻居已经染过色了，检查是否冲突
                        if colors[neighbor] != expected_color:
                            print("NO")
                            return # 发现矛盾，直接结束程序
                    else:
                        # 如果邻居没染过色，染上正确的颜色，并加入队列继续推导
                        colors[neighbor] = expected_color
                        queue.append(neighbor)

    # 如果所有植物都检查完了，没有发现矛盾
    print("YES")

if __name__ == "__main__":
```

```
    solve()
```

代码运行截图 （至少包含有"Accepted"）

M29740:神经网络（60分钟）

Topological order, http://cs101.openjudge.cn/practice/29740/

思路：题目本质上是一个基于有向图的数值计算模拟题。解题的核心在于确定计算的先后顺序。因为每个神经元必须在接收完所有上游神经元传来的信号后才能计算自己的最终状态，所以我们需要使用拓扑排序算法来理清依赖关系。具体步骤是先建立图结构并统计每个节点的入度，接着将所有入度为0的输入层节点放入队列。程序开始循环处理队列中的节点，对于每个取出的节点，将其计算后的信号值根据边权累加传递给下游邻居，并将邻居的入度减1。一旦某个邻居的入度减为0，说明其所有前置信号均已到位，此时减去该节点的阈值得到最终激活值。如果激活值大于0则将其加入队列继续传播，否则该节点不再向下传递非零信号。最后，如果处理的节点总数少于实际节点数则说明存在环路输出NULL，否则检查所有输出层节点，按编号顺序输出最终状态大于0的结果，若无符合条件的输出层节点则输出NULL。

代码

```
import sys

def solve():
    # 1. 快速读取所有输入
    # sys.stdin.read() 能一次性读取所有内容，处理大量数据时比 input() 快很多
    input_data = sys.stdin.read().split()
    if not input_data:
        return

    iterator = iter(input_data)

    try:
        n = int(next(iterator)) # 神经元总数
        p = int(next(iterator)) # 边的数量
    except StopIteration:
        return

    # 初始化数据结构
    # 题目中神经元编号是 1 到 n，为了方便，我们开 n+1 的数组，下标 0 不用
    c = [0] * (n + 1) # 记录神经元的状态值 (C)
    u = [0] * (n + 1) # 记录阈值/偏置 (U)
```

```python
    # 读取每个神经元的初始状态和阈值
    for i in range(1, n + 1):
        c[i] = int(next(iterator))
        u[i] = int(next(iterator))

    # 构建图 (邻接表) 和 记录入度/出度
    # adj[i] 存放从 i 出发的边，格式为 {目标节点: 权重}
    adj = [{} for _ in range(n + 1)]
    in_degree = [0] * (n + 1)  # 入度：有多少条边指向我
    out_degree = [0] * (n + 1) # 出度：我指向多少条边 (用于判断是否是输出层)

    for _ in range(p):
        src = int(next(iterator))
        dst = int(next(iterator))
        w = int(next(iterator))

        # 处理重边的情况：如果边已经存在，权重累加
        if dst not in adj[src]:
            adj[src][dst] = 0
            in_degree[dst] += 1 # 只有新连接才增加入度
            out_degree[src] += 1
        adj[src][dst] += w

    # 2. 拓扑排序准备
    # 队列里存放所有“入度为0”的节点（即不需要等待上游信号的节点）
    queue = []
    for i in range(1, n + 1):
        if in_degree[i] == 0:
            queue.append(i)

    processed_count = 0 # 记录处理了多少个节点，用于判断是否有环

    # 3. 开始信号传播 (BFS / 拓扑排序)
    while queue:
        curr = queue.pop(0)
        processed_count += 1

        # 只有当前神经元状态 > 0 时，才会向下游发送信号
        # 如果 <= 0，它虽然参与排序，但不贡献数值
        if c[curr] > 0:
            for neighbor, weight in adj[curr].items():
                c[neighbor] += weight * c[curr]

        # 无论当前节点是否发射信号，它的下游节点的等待数都要减1
        # 因为我们在逻辑上已经“处理完”了当前节点
        for neighbor in adj[curr]:
            in_degree[neighbor] -= 1
            # 如果邻居的所有上游都处理完了（入度变0）
            if in_degree[neighbor] == 0:
                # 核心公式：C = Sum(W * C_prev) - U
                # 这里减去阈值 U。注意：输入层一开始入度就是0，已经在循环外了，
                # 所以这里减阈值的操作只会发生在中间层和输出层，符合题目逻辑。
                c[neighbor] -= u[neighbor]
                queue.append(neighbor)

    # 4. 输出结果
```

```
    # 情况1：如果有环，图无法完成拓扑排序，处理的节点数 < n
    if processed_count < n:
        print("NULL")
        return

    # 情况2：输出合法的输出层神经元
    has_output = False
    for i in range(1, n + 1):
        # 必须是输出层（出度为0） 且 最终状态 > 0
        if out_degree[i] == 0 and c[i] > 0:
            print(f"{i} {c[i]}")
            has_output = True

    # 情况3：如果所有输出层神经元都 <= 0
    if not has_output:
        print("NULL")

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

代码运行截图 （至少包含有"Accepted"）

#51164096提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
import sys

def solve():
    # 1. 快速读取所有输入
    # sys.stdin.read() 能一次性读取所有内容，处理大量数据时比 input() 快很多
    input_data = sys.stdin.read().split()
    if not input_data:
        return
```

基本信息

#: 51164096
题目: 29740
提交人: 25n2200013554
内存: 3920kB
时间: 26ms
语言: Python3
提交时间: 2025-12-06 18:07:28

T27351:01 最小生成树（50分钟）

mst, http://cs101.openjudge.cn/practice/27351/

思路：解题的核心是将问题转化为求只包含0边时的连通块数量。我们可以维护一个未访问节点的集合，每次从未访问集合中取出一个节点作为新连通块的起点进行广度优先搜索。在搜索过程中，我们遍历当前未访问集合中的所有节点，如果发现某节点与当前节点之间没有题目给定的1边，说明它们之间存在0边，于是将该节点加入队列并从未访问集合中移除。重复这个过程直到所有节点都被访问，最终连通块的总数减1即为最小生成树的边权和。

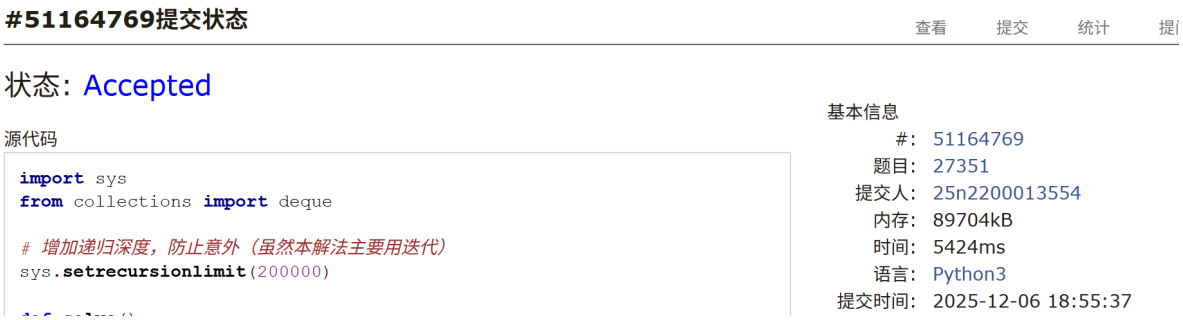
代码

```
import sys
from collections import deque

# 增加递归深度，防止意外（虽然本解法主要用迭代）
sys.setrecursionlimit(200000)
```

```python
def solve():
    # 1. 读取输入
    # sys.stdin.read 能够快速读取大量数据
    input_data = sys.stdin.read().split()
    if not input_data:
        return

    iterator = iter(input_data)
    try:
        n = int(next(iterator))
        m = int(next(iterator))
    except StopIteration:
        return

    # 2. 构建 "1-边" 的邻接表
    # 注意：我们只存题目给出的 1-边。没存的就是 0-边。
    # 使用 set 是为了 O(1) 快速判断边是否存在
    adj_one = [set() for _ in range(n + 1)]
    for _ in range(m):
        u = int(next(iterator))
        v = int(next(iterator))
        adj_one[u].add(v)
        adj_one[v].add(u)

    # 3. 初始化未访问集合
    # 相当于 Prim 算法中待处理的节点池
    unvisited = set(range(1, n + 1))

    # 记录连通块（岛屿）的数量
    components = 0

    # 4. 类似 Prim 的过程
    # 我们不使用优先队列，因为权值只有 0 和 1。
    # 我们总是优先处理 0 边的邻居（直接放入 queue），这相当于权值为0的优先队列。

    while unvisited:
        components += 1

        # 从未访问集合中随便拿一个点作为起点
        # 这一步相当于被迫花费 1 的代价跳到了一个新的连通块（除了第一个块）
        start_node = unvisited.pop()
        queue = deque([start_node])

        while queue:
            u = queue.popleft()

            # 【核心优化逻辑】
            # 我们要在 unvisited 里找 u 的 0-边邻居。
            # 0-边邻居 = 所有点 - (u 的 1-边邻居)
            # 我们遍历 unvisited，如果 v 不是 1-边邻居，那它就是 0-边邻居！

            to_remove = [] # 暂存这一轮找到的 0 边邻居

            # 这个循环看起来是 O(N)，但因为节点会被从 unvisited 中移除，
            # 所以每个节点只会被"成功处理"一次。
            # 只有当 v 是 1-边邻居时，才会留在 unvisited 里重复被查。
```

```
            # 总复杂度控制在 O(N + M)
            for v in unvisited:
                if v not in adj_one[u]:
                    to_remove.append(v)

            # 将找到的 0 边邻居加入队列继续扩展，并从待访问列表中移除
            for v in to_remove:
                unvisited.remove(v)
                queue.append(v)

    # 5. 输出结果
    # 最小生成树权值 = 需要连接 K 个连通块的边数 = K - 1
    print(components - 1)

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

代码运行截图 （至少包含有"Accepted"）



T30193:哈密顿激活层（90分钟）

DFS+剪枝, http://cs101.openjudge.cn/practice/30193/

思路：解题的核心是使用深度优先搜索配合强力剪枝。我们在递归搜索路径时，利用已知的关键点时间约束来进行两类判断：一是直接判断当前尝试的下一步是否符合该位置的时间戳要求或者该时刻必须到达的位置要求；二是利用曼哈顿距离和奇偶性原理预测未来，如果当前位置到下一个关键点的曼哈顿距离超过了剩余可用时间，或者两者差值的奇偶性不一致，说明无论如何都不可能按时到达，从而立即停止该分支的搜索并回溯。

代码

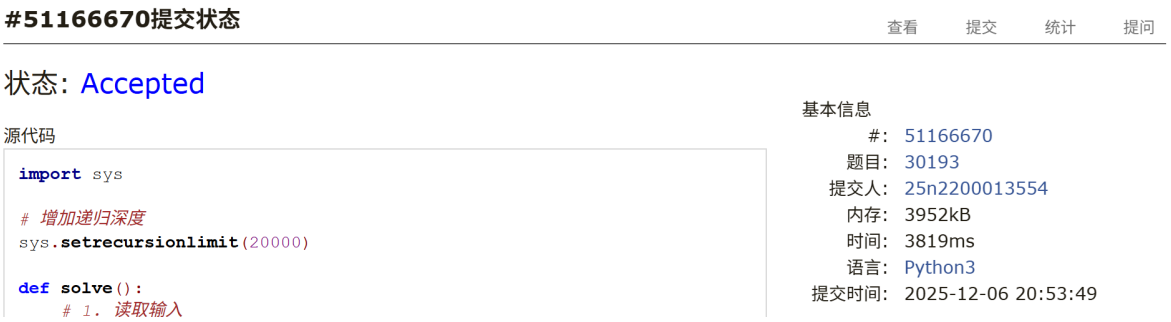
```
import sys

# 增加递归深度
sys.setrecursionlimit(20000)

def solve():
    # 1. 读取输入
    input_data = sys.stdin.read().split()
    if not input_data:
        return
```

```python
iterator = iter(input_data)
try:
    N = int(next(iterator))
    M = int(next(iterator))
    K = int(next(iterator))
    B = int(next(iterator))
except StopIteration:
    return

# 坐标映射：(r, c) -> time (0表示无限制，-1表示障碍)
grid_constraints = [[0] * M for _ in range(N)]
# 时间映射：time -> (r, c)
time_constraints = {}

# 存储所有检查点，用于剪枝 [(time, r, c), ...]
checkpoints = []

# 读取关键神经元 (K行)
start_pos = None
for _ in range(K):
    r = int(next(iterator)) - 1
    c = int(next(iterator)) - 1
    t = int(next(iterator))

    grid_constraints[r][c] = t
    time_constraints[t] = (r, c)
    checkpoints.append((t, r, c))

    if t == 1:
        start_pos = (r, c)

# 对检查点按时间排序
checkpoints.sort()

# 读取障碍物 (B行)
for _ in range(B):
    r = int(next(iterator)) - 1
    c = int(next(iterator)) - 1
    grid_constraints[r][c] = -1 # -1 代表障碍

# 计算路径总长度
total_steps = N * M - B

# 记录路径和访问状态
path = []
visited = [[False] * M for _ in range(N)]

# 方向数组
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

def dfs(x, y, step, cp_idx):
    # 1. 成功终止条件
    if step == total_steps:
        return True
```

```
        next_step = step + 1

        # 2. 核心剪枝：曼哈顿距离 + 奇偶性校验
        # 寻找下一个需要满足的 checkpoint
        # cp_idx 参数优化了查找过程，不用每次遍历列表
        if cp_idx < K:
            target_t, target_r, target_c = checkpoints[cp_idx]
            dist = abs(x - target_r) + abs(y - target_c)
            rem_time = target_t - step

            # 如果剩余步数 < 距离，或者奇偶性不匹配，直接剪枝
            if dist > rem_time or (rem_time - dist) % 2 != 0:
                return False

        # 3. 尝试四个方向
        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            # 越界检查
            if not (0 <= nx < N and 0 <= ny < M):
                continue

            # 访问检查 & 障碍检查
            # grid_constraints[nx][ny] == -1 是障碍
            if visited[nx][ny] or grid_constraints[nx][ny] == -1:
                continue

            # --- 约束逻辑检查 ---

            # 情况A: 下一个格子 (nx, ny) 身上有特定时间要求 T_req
            req_time = grid_constraints[nx][ny]
            if req_time > 0 and req_time != next_step:
                # 如果这个格子有要求，但要求的不是下一步，那现在不能进去
                continue

            # 情况B: 下一步时间 next_step 要求必须去某个特定格子 (rx, ry)
            if next_step in time_constraints:
                required_pos = time_constraints[next_step]
                if (nx, ny) != required_pos:
                    # 如果时间表要求去别的地方，那也不能去 (nx, ny)
                    continue

            # 4. 递归下探
            visited[nx][ny] = True
            path.append((nx, ny))

            # 如果当前访问的恰好是 cp_idx 指向的检查点，递归时索引+1，看下一个检查点
            next_cp_idx = cp_idx
            if req_time == next_step:
                next_cp_idx += 1

            if dfs(nx, ny, next_step, next_cp_idx):
                return True

            # 回溯
            path.pop()
```

```
            visited[nx][ny] = False

        return False

    # 启动 DFS
    # 从起点开始，起点已经在输入中被标记为 visited
    if start_pos:
        sx, sy = start_pos
        visited[sx][sy] = True
        path.append((sx, sy))

        # 初始 cp_idx: 如果起点就是第一个检查点(t=1)，那么我们要关注的是列表里的第2个(index
1)
        # 输入保证有一行 t=1，且 checkpoints 已排序，所以 checkpoints[0] 肯定是 t=1
        if dfs(sx, sy, 1, 1):
            for r, c in path:
                print(f"{r + 1} {c + 1}")
        else:
            print("-1")
    else:
        # 理论上根据题目描述不会发生（保证包含起点信息）
        print("-1")

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

代码运行截图 （至少包含有"Accepted"）



2. 学习总结和收获

本次月考第一题比较基础，其他题目对我来说都有些难度，尤其是植物观察和神经网络的题目，理解题目意图就很费劲了。随着课程的进行月考会涉及已经讲过的知识，对应的10页cheatingpaper需要塞下更多东西了，对我来说上机考试还是具有非常大的压力。