Assignment #C: 五味杂陈

Updated 1148 GMT+8 Dec 10, 2024

2024 fall, Complied by 胡杨 元培学院

说明：

1）请把每个题目解题思路（可选），源码Python, 或者C++（已经在Codeforces/Openjudge上AC），截图（包含Accepted），填写到下面作业模版中（推荐使用 typora https://typoraio.cn，或者用word）。AC 或者没有AC，都请标上每个题目大致花费时间。

2）提交时候先提交pdf文件，再把md或者doc文件上传到右侧“作业评论”。Canvas需要有同学清晰头像、提交文件有pdf、"作业评论"区有上传的md或者doc附件。

3）如果不能在截止前提交作业，请写明原因。

1. 题目

1115. 取石子游戏

dfs, https://www.acwing.com/problem/content/description/1117/

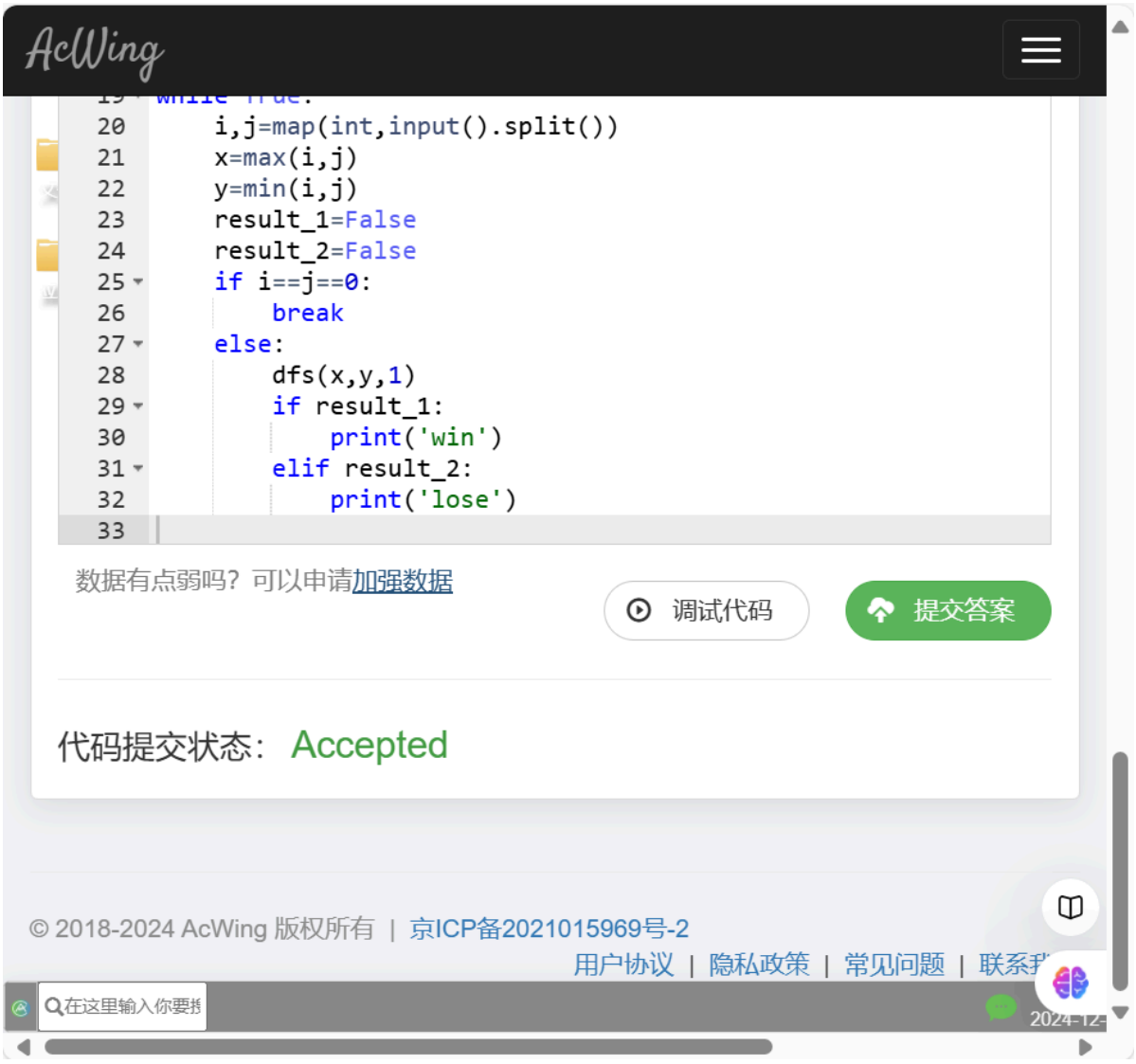
思路：根据提示写的，至于为什么我也想不明白（倒下）先判断a//b>=2是否满足，否则先手必胜，需要注意的是a==b时也是先手必胜！如果不满足，则先后手顺序交换，继续判断

代码：

```
def dfs(a,b,t):
    global result_1,result_2
    if a<=0 or b<=0:
        return
    if t%2==1:
        if a//b>=2 or a==b:
            result_1=True
            return
        else:
            dfs(b,a-b,t+1)
    elif t%2==0:
        if a//b>=2 or a==b:
            result_2=True
            return
        else:
            dfs(b,a-b,t+1)


while True:
    i,j=map(int,input().split())
    x=max(i,j)
    y=min(i,j)
```

```
    result_1=False
    result_2=False
    if i==j==0:
        break
    else:
        dfs(x,y,1)
        if result_1:
            print('win')
        elif result_2:
            print('lose')
```

代码运行截图 （至少包含有"Accepted"）



25570: 洋葱

Matrices, http://cs101.openjudge.cn/practice/25570

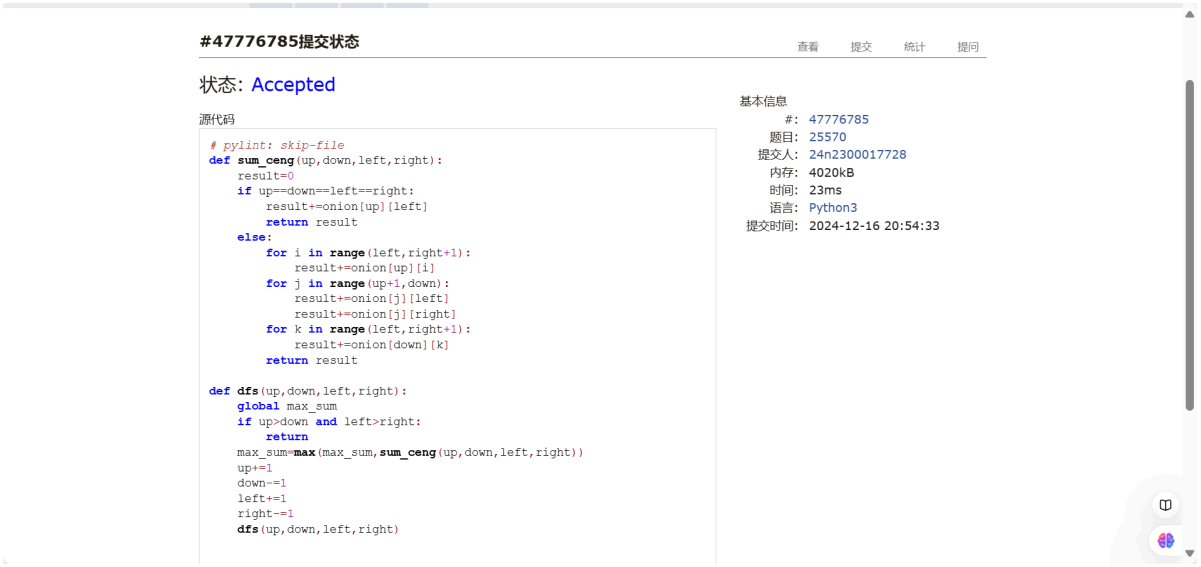
思路：使用上下左右四个指针，一圈一圈求和就行，注意结束节点是up>down and left>right，另外需要注意如果n是奇数，即最后中心只有一个数，如果按一圈的求和方式就会把那个数加两遍，单独讨论这种情况即可

OJ的pylint是静态检查，有时候报的不对。解决方法有两种，如下：
1）第一行加# pylint: skip-file
2）方法二：如果函数内使用全局变量（变量类型是immutable，如int），则需要在程序最开始声明一下。如果是全局变量是list类型，则不受影响。

代码：

```
# pylint: skip-file
def sum_ceng(up,down,left,right):
    result=0
    if up==down==left==right:
        result+=onion[up][left]
        return result
    else:
        for i in range(left,right+1):
            result+=onion[up][i]
        for j in range(up+1,down):
            result+=onion[j][left]
            result+=onion[j][right]
        for k in range(left,right+1):
            result+=onion[down][k]
        return result

def dfs(up,down,left,right):
    global max_sum
    if up>down and left>right:
        return
    max_sum=max(max_sum,sum_ceng(up,down,left,right))
    up+=1
    down-=1
    left+=1
    right-=1
    dfs(up,down,left,right)


n=int(input())
onion=[]
for _ in range(n):
    onion.append(list(map(int,input().split())))
up=0
down=n-1
left=0
right=n-1
max_sum=float('-inf')

dfs(up,down,left,right)
print(max_sum)
```

代码运行截图（至少包含有"Accepted"）




```python
# pylint: skip-file
def sum_ceng(up,down,left,right):
    result=0
    if up==down==left==right:
        result+=onion[up][left]
        return result
    else:
        for i in range(left,right+1):
            result+=onion[up][i]
        for j in range(up+1,down):
            result+=onion[j][left]
            result+=onion[j][right]
        for k in range(left,right+1):
            result+=onion[down][k]
        return result

def dfs(up,down,left,right):
    global max_sum
    if up>down and left>right:
        return
    max_sum=max(max_sum,sum_ceng(up,down,left,right))
    up+=1
    down-=1
    left+=1
    right-=1
    dfs(up,down,left,right)
```

1526C1. Potions(Easy Version)

greedy, dp, data structures, brute force, *1500, https://codeforces.com/problemset/problem/1526/C1

思路：试图dp但是超时/爆内存的可能性过大，去学习了一下heapq和后悔贪心

后悔贪心这一算法的思路并没有理解，但是本题的思路是理解了的。先假设把目前这瓶药水喝了，如果健康值是正的，就先不管，该药水对健康值的影响加入heap；如果喝了过后健康值是负的，就舍弃负的最多的那一瓶药水，即heap[0]，**需要注意的是，此时既可以保证heap[0]是排在目前这瓶药水之前的（已经进队了），也可以保证舍弃的是最负的一个**

heapq

`heapq` 是 Python 的一个内置模块，提供了堆队列算法的实现。堆是一种特殊的树结构，满足堆属性：对于任意给定的节点 C，如果 P 是 C 的父节点，则 P 的键值小于或等于 C 的键值（在最小堆中），或者大于或等于 C 的键值（在最大堆中）。这种结构非常适合用来实现**优先级队列**。

`heapq` 模块实现了最小堆，意味着堆中的最小元素总是被弹出。Python 中的 `heapq` 实际上是**在列表（list）的基础上实现的**，但是它提供了一系列函数来维护这个列表作为一个堆。

以下是 `heapq` 模块的一些常用函数：

- `heappush(heap, item)`：将 `item` **添加到堆** `heap` **中，并保持堆的性质。**
- `heappop(heap)`：从堆 `heap` 中弹出并返回最小的元素。**如果堆为空，则引发 IndexError。**
- `heapreplace(heap, item)`：用 `item` 替换堆中的最小元素并返回旧的最小元素。等价于 `heappop()` 后跟 `heappush()`，但更高效。
- `heappushpop(heap, item)`：将 `item` 添加到堆 `heap` 中，然后弹出并返回最小的元素。与 `heapreplace()` 相反，当有新的 `item` 要插入时先推后弹。
- `heapify(x)`：**将列表** `x` **转换成堆，原地修改，线性时间内完成。**
- `nlargest(n, iterable[, key])`：返回 `iterable` 中**最大的** `n` **个元素组成的列表，可以指定** `key` **函数用于排序。**
- `nsmallest(n, iterable[, key])`：返回 `iterable` 中**最小的** `n` **个元素组成的列表，可以指定** `key` **函数用于排序。**

使用 `heapq` 创建和操作堆的一个简单例子如下：

```
import heapq

# 初始化一个空堆
heap = []

# 使用 heappush 添加元素
heapq.heappush(heap, 10)
heapq.heappush(heap, 1)
heapq.heappush(heap, 5)

# 查看堆顶元素（最小元素）
print("The smallest element is:", heap[0])

# 弹出最小元素
min_element = heapq.heappop(heap)
print("Popped the smallest element:", min_element)

# 将列表转换为堆
lst = [3, 1, 4, 1, 5, 9]
heapq.heapify(lst)
print("Heapified list:", lst)

# 获取三个最大的元素
largest_three = heapq.nlargest(3, lst)
print("Three largest elements are:", largest_three)
```

贪心的两种解法

排序解法

用排序法常见的情况是输入一个包含几个（一般一到两个）权值的数组，通过排序然后遍历模拟计算的方法求出最优值。

后悔解法

思路是无论当前的选项是否最优都接受，然后进行比较，如果选择之后不是最优了，则反悔，舍弃掉这个选项；否则，正式接受。如此往复。

代码：

```
import heapq
n=int(input())
a=list(map(int,input().split()))
heap=[]
health=0

for i in a:
    heapq.heappush(heap,i)
    health+=i
    if health<0:
```

```
        if heap: #确保heap里有元素，防止index error（虽然heap里肯定有元素
            give_up=heapq.heappop(heap)
            health-=give_up #放弃一瓶肯定也就够了，因为它是最负的，而health在喝i之前>=0
print(len(heap))
```

代码运行截图 （至少包含有"Accepted"）

teen	1526C1 - Potions (Easy Version)	Python 3	Accepted	77 ms	0 KB

感觉后悔解法真的好适合本题啊

22067: 快速堆猪

辅助栈，http://cs101.openjudge.cn/practice/22067/

思路：主要困难存在于：1.不定行输入的读取，2.不知道怎么在移除主栈元素的同时移除辅助栈中的该元素

原本尝试在每次移除主栈元素后再重新将其转化为heapq辅助栈，但是会超时

后来采用**延迟删除**方法，如果辅助栈堆顶即最小值已经不在主栈中，则删除它，直到堆顶值在主栈中停止，而对于堆中其他已经删除的值采取“暂时忍受”的策略，需要时（指已经影响到弹出最小值时）再删除

读取不定行输入

```
lines = []
while True:
    try:
        line = input()
        if line:  # 如果输入非空，则添加到列表中
            lines.append(line)
        else:  # 遇到空行则停止读取
            break
    except EOFError:  # 当用户输入 EOF (Ctrl+D on Unix, Ctrl+Z on Windows) 时退出循环
        break
```

辅助栈

辅助栈是一种数据结构设计模式，通常用于增强或扩展另一个栈的功能。通过使用一个或多个额外的栈（即辅助栈），可以更高效地实现某些操作或特性，而这些操作或特性在单个栈上可能难以实现或效率较低。

辅助栈的应用场景

1. **最小栈（Min Stack）**：
   - 问题描述：设计一个支持 `push`、`pop`、`top` 操作，并能在常数时间内检索到最小元素的栈。
   - 解决方案：除了主栈外，我们还可以维护一个辅助栈，用来存储当前栈中的最小值。每当有新元素入栈时，如果该元素小于或等于辅助栈顶元素，则也将其压入辅助栈；**当从主栈弹出元素时，如果该元素等于辅助栈顶元素，则同时从辅助栈中弹出。这样，辅助栈顶始终保存着当前栈中的最小值。**
2. **平衡括号检查**：
   - 问题描述：给定一个包含多种类型括号（如圆括号 `()`、方括号 `[]` 和花括号 `{}`）的字符串，判断这些括号是否正确配对。
   - 解决方案：可以使用一个主栈来处理括号匹配的问题，而辅助栈则可以帮助记录每种括号最后出现的位置或类型，以便进行更复杂的匹配逻辑。
3. **回滚机制**：
   - 在某些应用中，你可能需要能够撤销最近的操作。可以使用辅助栈来保存每个操作的状态，从而允许用户回退到之前的状态。
4. **多态性操作**：
   - 有时你需要根据栈的不同状态执行不同的操作。例如，在一个表达式求值器中，你可以使用辅助栈来跟踪运算符的优先级或者存储中间计算结果。

最小栈示例代码

下面是一个用 Python 实现的最小栈的例子：

```
class MinStack:

    def __init__(self):
        self.stack = []  # 主栈
        self.min_stack = []  # 辅助栈，用于保存每个状态下的最小值

    def push(self, x: int) -> None:
        self.stack.append(x)
        if not self.min_stack or x <= self.min_stack[-1]:
            self.min_stack.append(x)

    def pop(self) -> None:
        if self.stack:
            top = self.stack.pop()
            if top == self.min_stack[-1]:
                self.min_stack.pop()

    def top(self) -> int:
        if self.stack:
            return self.stack[-1]

    def getMin(self) -> int:
        if self.min_stack:
            return self.min_stack[-1]
```

```
# 示例用法
min_stack = MinStack()
min_stack.push(-2)
min_stack.push(0)
min_stack.push(-3)
print(min_stack.getMin())  # 返回 -3
min_stack.pop()
print(min_stack.top())     # 返回 0
print(min_stack.getMin())  # 返回 -2
```

在这个例子中，`MinStack` 类实现了基本的栈操作以及 `getMin` 方法，可以在 O(1) 时间复杂度内获取当前栈中的最小值。通过维护两个栈——一个用于所有元素，另一个仅用于最小值——我们可以确保每次 `push` 和 `pop` 操作都能正确更新最小值信息。

这里用的辅助栈是普通的列表而非heapq，但是heapq更无脑XD

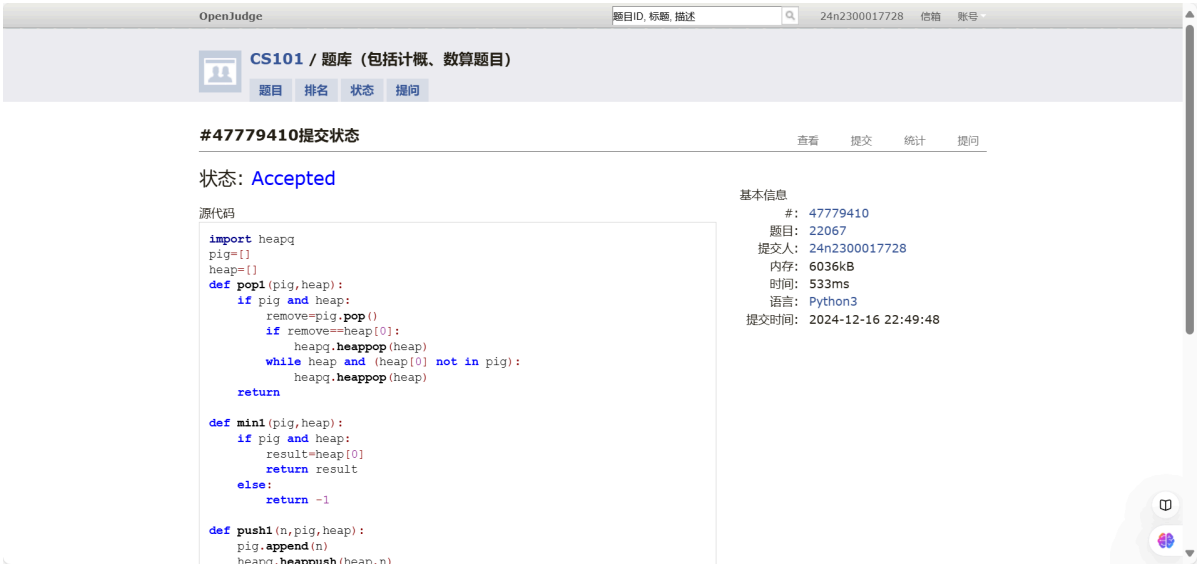
使用heapq的最小栈示例代码即本题代码

代码：

```
import heapq
pig=[]
heap=[]
def pop1(pig,heap):
    if pig and heap:
        remove=pig.pop()
        if remove==heap[0]:
            heapq.heappop(heap)
        while heap and (heap[0] not in pig):
            heapq.heappop(heap)
    return

def min1(pig,heap):
    if pig and heap:
        result=heap[0]
        return result
    else:
        return -1

def push1(n,pig,heap):
    pig.append(n)
    heapq.heappush(heap,n)
    return

while True:
    try:
        line=list(input().split())
        if line:  #如果输入非空，执行操作
            if line[0]=='pop':
                pop1(pig,heap)
            elif line[0]=='min':
                if min1(pig,heap)==-1:
                    continue
                else:
```

```
                print(min1(pig,heap))
            elif line[0]=='push':
                n=int(line[1])
                push1(n,pig,heap)
        else:  # 遇到空行则停止读取
            break
    except EOFError:  # 当用户输入 EOF (Ctrl+D on Unix, Ctrl+Z on Windows) 时退出循环
        break
```

代码运行截图 （至少包含有"Accepted"）



20106: 走山路

Dijkstra, http://cs101.openjudge.cn/practice/20106/

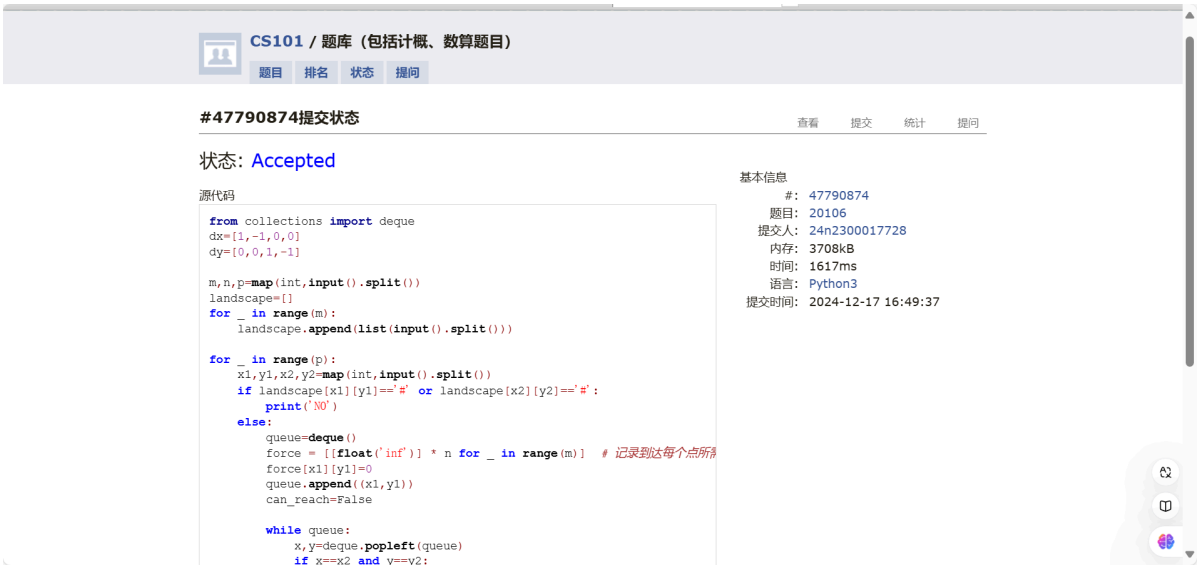
思路：和bfs最大的区别在于每一步的权重不都是1，而是不同的非负数！进队的条件需要改成到达该点的体力值更小，而不是没有进过对就行，同时允许重复进队以反复更新最小体力值

代码：

```
from collections import deque
dx=[1,-1,0,0]
dy=[0,0,1,-1]

m,n,p=map(int,input().split())
landscape=[]
for _ in range(m):
    landscape.append(list(input().split()))

for _ in range(p):
    x1,y1,x2,y2=map(int,input().split())
    if landscape[x1][y1]=='#' or landscape[x2][y2]=='#':
```

```
        print('NO')
    else:
        queue=deque()
        force = [[float('inf')] * n for _ in range(m)]  # 记录到达每个点所需要的最小体力值
        force[x1][y1]=0
        queue.append((x1,y1))
        can_reach=False

        while queue:
            x,y=deque.popleft(queue)
            if x==x2 and y==y2:
                can_reach=True

            for i in range(4):
                nf=force[x][y]
                nx=x+dx[i]
                ny=y+dy[i]
                if 0<=nx<m and 0<=ny<n and landscape[nx][ny]!='#':
                    nf+=abs(int(landscape[nx][ny])-int(landscape[x][y]))
                    if nf<force[nx][ny]:
                        force[nx][ny]=nf
                        queue.append((nx,ny))

        if not can_reach:
            print('NO')
        else:
            print(force[x2][y2])
```

代码运行截图 （至少包含有"Accepted"）



04129: 变换的迷宫

bfs, http://cs101.openjudge.cn/practice/04129/

思路：原本写了一个和走山路差不多的代码，然后发现了关键问题——我们应该首先考虑能够到达的情况，然后再研究最短的时间。如果一个地方是石头，可以在其旁边来回踱步直到能够通过，如果按照更短时间进队的话，踱步的情况显然时间更长，无法被考虑到。

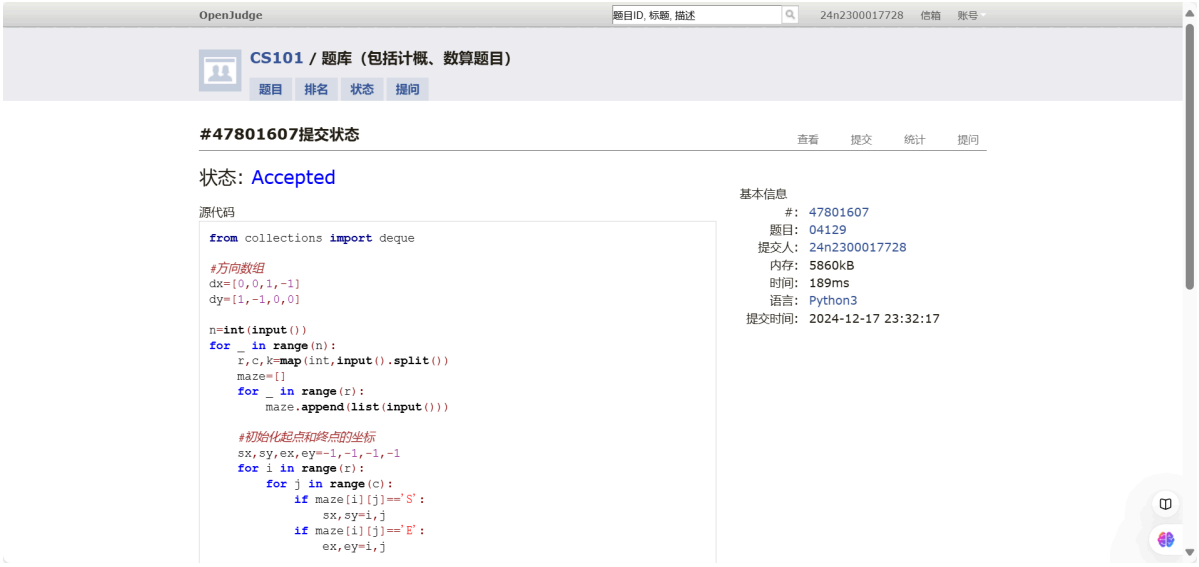
询问AI后得到的解决方案是在times数组里添加一个数据维度，将时间按照除以k的余数分类，以保证能通过的石头都通过，同时用时最短

代码：

```
from collections import deque

#方向数组
dx=[0,0,1,-1]
dy=[1,-1,0,0]

n=int(input())
for _ in range(n):
    r,c,k=map(int,input().split())
    maze=[]
    for _ in range(r):
        maze.append(list(input()))

    #初始化起点和终点的坐标
    sx,sy,ex,ey=-1,-1,-1,-1
    for i in range(r):
        for j in range(c):
            if maze[i][j]=='S':
                sx,sy=i,j
            if maze[i][j]=='E':
                ex,ey=i,j

    #bfs
    queue=deque()
    queue.append((sx,sy,0))  #初始位置和时间
    times=[[[float('inf')]*k for _ in range(c)] for _ in range(r)]  #记录到达每个点的最短用时,按t%k分类
    times[sx][sy][0]=0

    found=False  #是否找到终点
    while queue:
        x,y,t=deque.popleft(queue)

        #检查是否到达出口
        if (x,y)==(ex,ey):
            print(t)
            found=True
            break

        for i in range(4):
            nx=x+dx[i]
```

```
            ny=y+dy[i]
            nt=t+1
            nk=nt%k

            if 0<=nx<r and 0<=ny<c:
                # 如果下一步的时间是K的倍数，那么可以走任何位置；否则检查目标位置是否为石头
                if nk==0 or maze[nx][ny]!='#':
                    if nt<times[nx][ny][nk]:
                        times[nx][ny][nk]=nt
                        queue.append((nx,ny,nt))

    if not found:
        print('Oop!')
```

代码运行截图 （至少包含有"Accepted"）



2. 学习总结和收获

如果作业题目简单，有否额外练习题目，比如：OJ“计概2024fall每日选做”、CF、LeetCode、洛谷等网站题目。

学到了很多东西，后悔贪心，heapq，辅助栈，三维数组等等

一个星期没写代码，感觉手生得厉害，本来会的东西都忘完了的感觉，马上要机考了多练练leetcode吧，但同时又有各种其他的作业，忙得想死，尽力局

为了防止忘记模板题和典型题准备把讲义上dp、bfs、dfs的典型模板和典型例题全部抄在cheating paper上（x）