Assignment #D: 十全十美

Updated 1254 GMT+8 Dec 17, 2024

2024 fall, Complied by <mark>同学的姓名、院系</mark>

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

02692: 假币问题

brute force, http://cs101.openjudge.cn/practice/02692

思路：主要思路就是先经过多轮筛选尽量找出真币（利用 even 说明这一次测试的全是真币，再利用不是 even 的测试说明没测试的全是真币）；
在大部分情况下，经过前面的这两轮筛选已经能够确定假币是谁了，但是如果此时还无法筛选出一个具体的假币，还有两三个人选的话（这也正是此题的难点），就得接着再看看能不能有某次测试某一边全是真币，如果有的话就能判断出假币到底是轻还是重；然后再利用这一点就能快速判断出假币到底是谁！

代码：

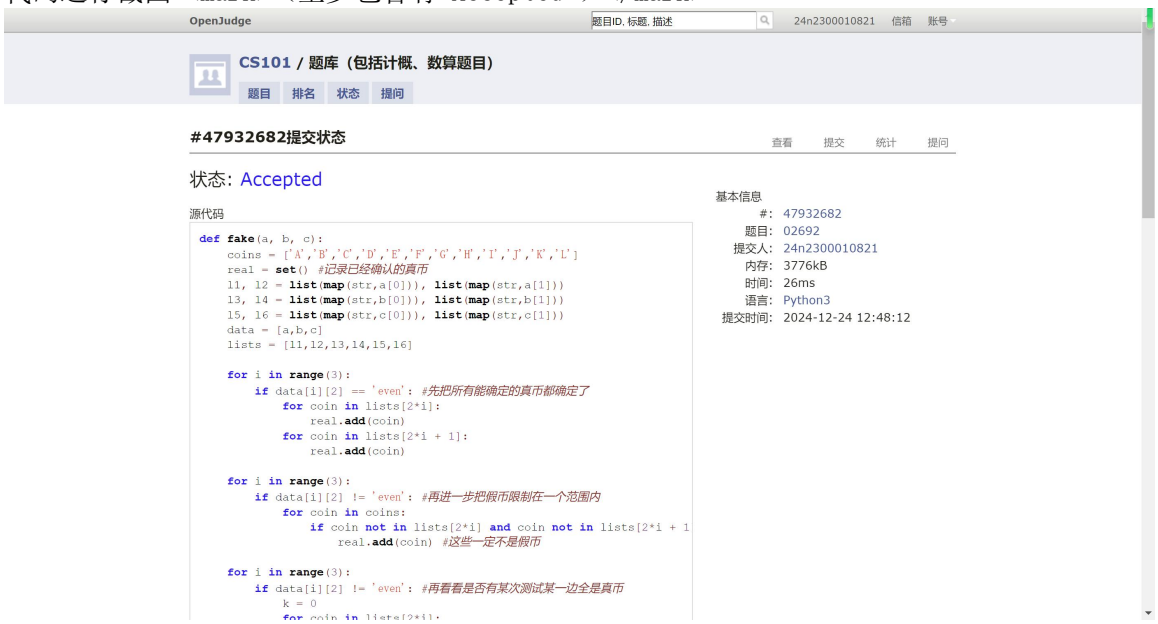
```python
```
```
def fake(a, b, c):
    coins = ['A','B','C','D','E','F','G','H','I','J','K','L']
    real = set() #记录已经确认的真币
    l1, l2 = list(map(str,a[0])), list(map(str,a[1]))
    l3, l4 = list(map(str,b[0])), list(map(str,b[1]))
    l5, l6 = list(map(str,c[0])), list(map(str,c[1]))
```

```
data = [a,b,c]
lists = [l1,l2,l3,l4,l5,l6]

for i in range(3):
    if data[i][2] == 'even': #先把所有能确定的真币都确定了
        for coin in lists[2*i]:
            real.add(coin)
        for coin in lists[2*i + 1]:
            real.add(coin)

for i in range(3):
    if data[i][2] != 'even': #再进一步把假币限制在一个范围内
        for coin in coins:
            if coin not in lists[2*i] and coin not in lists[2*i + 1]:
                real.add(coin) #这些一定不是假币

for i in range(3):
    if data[i][2] != 'even': #再看看是否有某次测试某一边全是真币
        k = 0
        for coin in lists[2*i]:
            if coin not in real:
                k = 1
        if k == 0: #说明左边的全部是真币
            if data[i][2] == 'up':
                status = 'light' #右边轻且假币在右边，说明假币轻
            else:
                status = 'heavy'
            for coin in coins: #再进一步添加一些真币
                if coin not in lists[2*i + 1]:
                    real.add(coin) #只要不在右边就不是假币

        k = 0
        for coin in lists[2*i + 1]:
            if coin not in real:
                k = 1
        if k == 0: #说明右边的全部是真币
            if data[i][2] == 'up':
                status = 'heavy' #右边轻且假币在左边，说明假币重
            else:
                status = 'light'
            for coin in coins: #再进一步添加一些真币
                if coin not in lists[2*i]:
                    real.add(coin) #只要不在左边就不是假币

if status: #如果此时已经确定了假币是轻还是重，那么可以利用这一点进一步排除很多真币
    for i in range(3):
```

```
            if data[i][2] == 'up':
                if status == 'light': #如果已知假币轻，又已知右边轻，则假币必在右
边
                    for coin in coins:
                        if coin not in lists[2*i + 1]:
                            real.add(coin) #只要不在右边就不是假币
                else: #反之亦然
                    for coin in coins:
                        if coin not in lists[2*i]:
                            real.add(coin) #只要不在左边就不是假币

            if data[i][2] == 'down':
                if status == 'heavy': #如果已知假币重，又已知右边重，则假币必在右
边
                    for coin in coins:
                        if coin not in lists[2*i + 1]:
                            real.add(coin) #只要不在右边就不是假币
                else: #反之亦然
                    for coin in coins:
                        if coin not in lists[2*i]:
                            real.add(coin) #只要不在左边就不是假币

    dup = coins[:]
    for coin in coins:
        if coin in real:
            dup.remove(coin)
    t = dup[0] #找出假币
    for i in range(3):
        if data[i][2] == 'up': #分析假币到底是轻还是重
            if t in lists[2*i]: #假币在左且右端轻，则说明假币重
                status = 'heavy'
            else:
                status = 'light'
            break
        elif data[i][2] == 'down':
            if t in lists[2*i]: #假币在左且右端重，则说明假币轻
                status = 'light'
            else:
                status = 'heavy'
            break

    result.append(f"{t} is the counterfeit coin and it is {status}.")

n = int(input())
result = []
for i in range(n):
    a = list(map(str,input().split()))
```

```
    b = list(map(str,input().split()))
    c = list(map(str,input().split()))
    fake(a, b, c)
for i in result:
    print(i)
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



01088: 滑雪

dp, dfs similar, http://cs101.openjudge.cn/practice/01088

思路：建立一个 dp 列表，用于记录此前已经计算过的 dfs 的结果（这样可以避免子问题的重复计算，从而可以大幅减少时间）；随后套用经典的 dfs 方法的模板即可

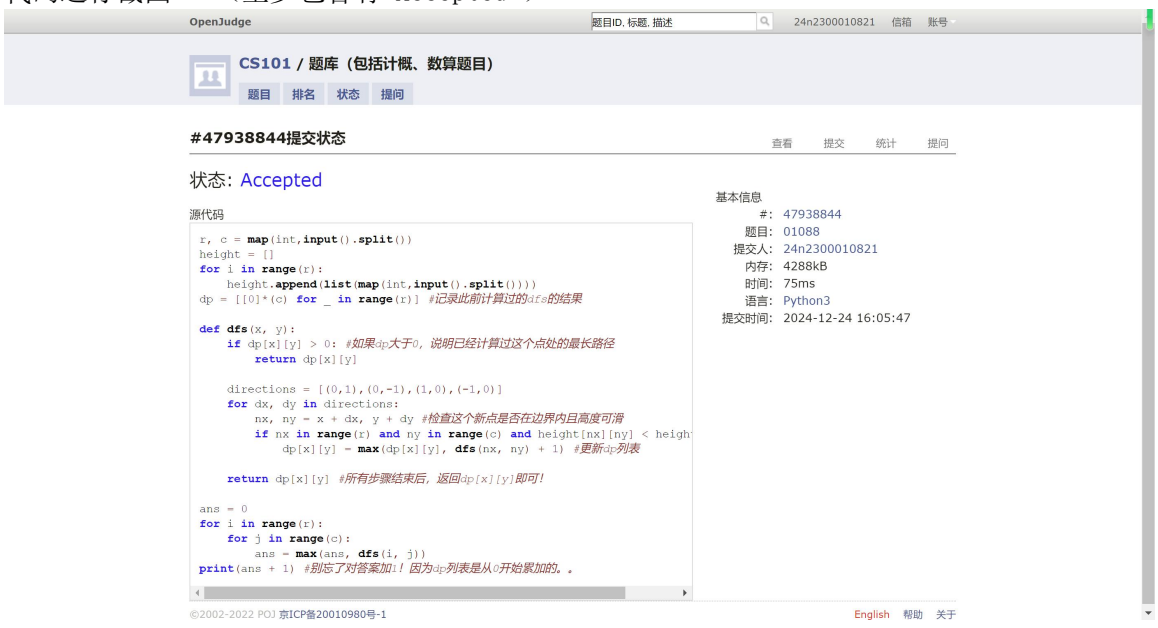
代码：

```python

```

```
r, c = map(int,input().split())
height = []
```

```
for i in range(r):
    height.append(list(map(int,input().split())))
dp = [[0]*(c) for _ in range(r)] #记录此前计算过的 dfs 的结果

def dfs(x, y):
    if dp[x][y] > 0: #如果 dp 大于 0，说明已经计算过这个点处的最长路径
        return dp[x][y]

    directions = [(0,1),(0,-1),(1,0),(-1,0)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy #检查这个新点是否在边界内且高度可滑
        if nx in range(r) and ny in range(c) and height[nx][ny] < height[x][y]:
            dp[x][y] = max(dp[x][y], dfs(nx, ny) + 1) #更新 dp 列表

    return dp[x][y] #所有步骤结束后，返回 dp[x][y]即可！

ans = 0
for i in range(r):
    for j in range(c):
        ans = max(ans, dfs(i, j))
print(ans + 1) #别忘了对答案加 1！因为 dp 列表是从 0 开始累加的。。
```

代码运行截图 ==（至少包含有"Accepted"）==



25572: 螃蟹采蘑菇

bfs, dfs, http://cs101.openjudge.cn/practice/25572/

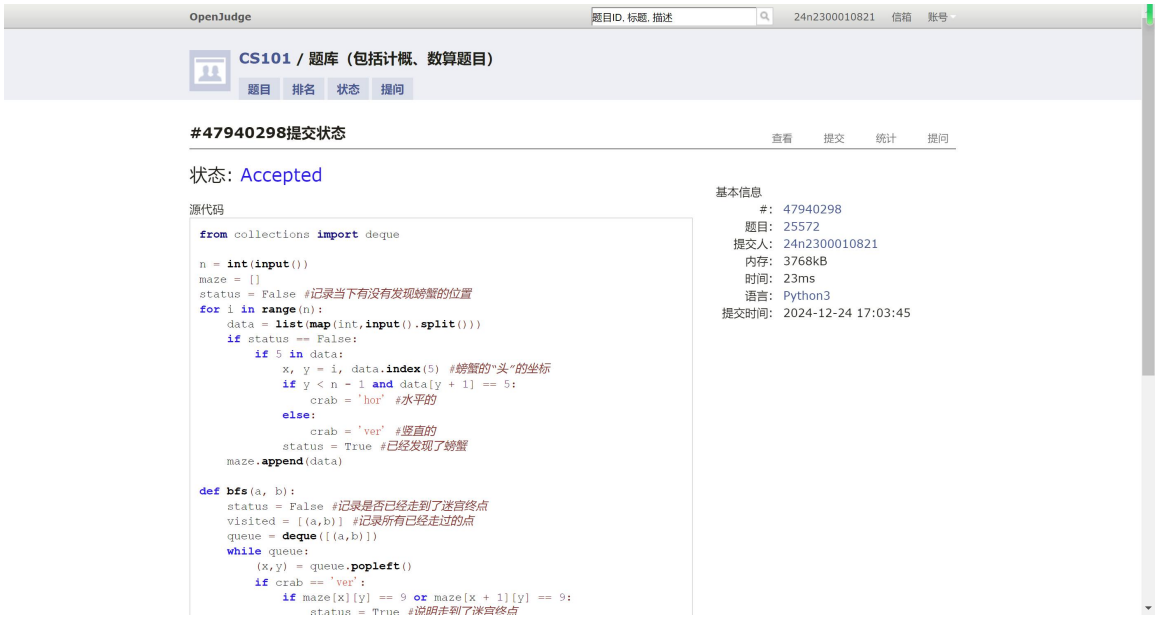
思路：这题算是整个作业里最简单的一道题了，直接按照螃蟹的身体状态（是水平的还是竖直的）进行分类讨论即可，方法就是最常规的bfs~

代码：

```python

```
```
from collections import deque

n = int(input())
maze = []
status = False #记录当下有没有发现螃蟹的位置
for i in range(n):
    data = list(map(int,input().split()))
    if status == False:
        if 5 in data:
            x, y = i, data.index(5) #螃蟹的“头”的坐标
            if y < n - 1 and data[y + 1] == 5:
                crab = 'hor' #水平的
            else:
                crab = 'ver' #竖直的
            status = True #已经发现了螃蟹
    maze.append(data)

def bfs(a, b):
    status = False #记录是否已经走到了迷宫终点
    visited = [(a,b)] #记录所有已经走过的点
    queue = deque([(a,b)])
    while queue:
        (x,y) = queue.popleft()
        if crab == 'ver':
            if maze[x][y] == 9 or maze[x + 1][y] == 9:
                status = True #说明走到了迷宫终点
                break
        elif crab == 'hor':
            if maze[x][y] == 9 or maze[x][y + 1] == 9:
                status = True #说明走到了迷宫终点
                break
        directions = [(0,1),(0,-1),(1,0),(-1,0)]
        for dx, dy in directions:
```

```
                nx, ny = x + dx, y + dy #检查这个新点是否在边界内、且可走、且未走过
                if crab == 'hor' and (nx,ny) not in visited:
                    if nx in range(n) and ny in range(n - 1) and maze[nx][ny] != 1 and
maze[nx][ny + 1] != 1:
                        queue.append((nx,ny))
                        visited.append((nx,ny))
                elif crab == 'ver' and (nx,ny) not in visited:
                    if nx in range(n - 1) and ny in range(n) and maze[nx][ny] != 1 and
maze[nx + 1][ny] != 1:
                        queue.append((nx,ny))
                        visited.append((nx,ny))

    if status == True: #能走的路径已经走完了之后就判断是否已经到达终点
        print("yes")
    else:print("no")

bfs(x, y)
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



27373: 最大整数

dp, http://cs101.openjudge.cn/practice/27373/

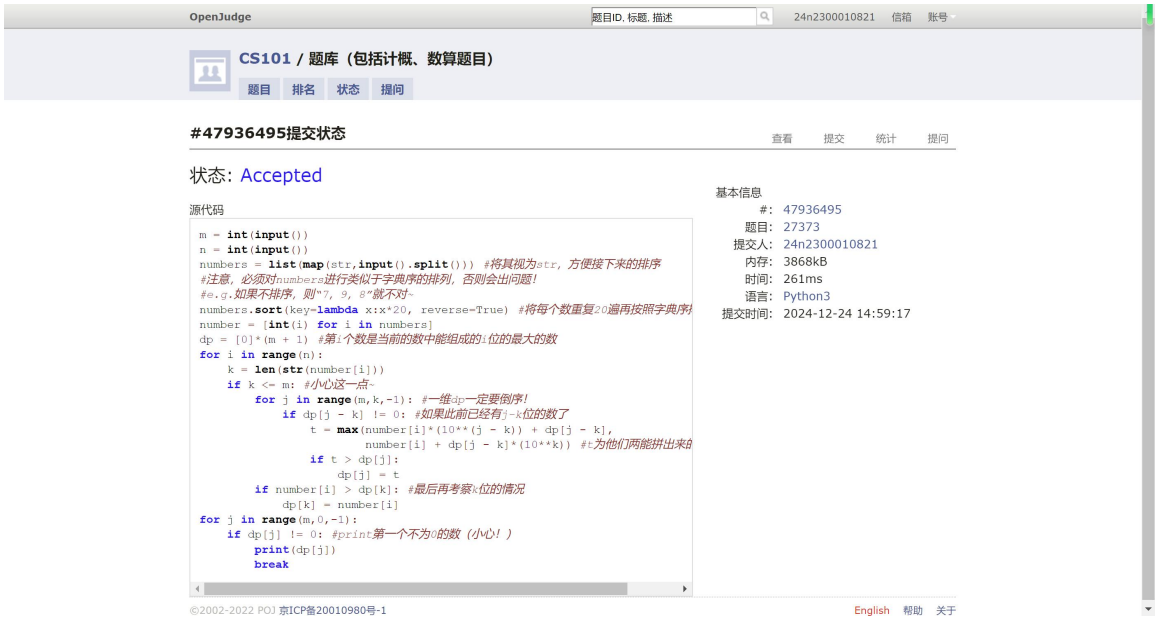
思路：先对列表 numbers 里的数按照类似于字典序的方法进行排序（将其视为 string，重复二

十遍后按照字典序进行排序），然后使用常规的 dp 方法即可（建立一个 dp 列表，其中的第 i 个数是当前的数中能组成的 i 位的最大的数，然后递归~）

代码：

```python
```
```
m = int(input())
n = int(input())
numbers = list(map(str,input().split())) #将其视为 str，方便接下来的排序
#注意，必须对 numbers 进行类似于字典序的排列，否则会出问题！
#e.g.如果不排序，则“7，9，8”就不对~
numbers.sort(key=lambda x:x*20, reverse=True) #将每个数重复 20 遍再按照字典序排列
number = [int(i) for i in numbers]
dp = [0]*(m + 1) #第 i 个数是当前的数中能组成的 i 位的最大的数
for i in range(n):
    k = len(str(number[i]))
    if k <= m: #小心这一点~
        for j in range(m,k,-1): #一维 dp 一定要倒序！
            if dp[j - k] != 0: #如果此前已经有 j-k 位的数了
                t = max(number[i]*(10**(j - k)) + dp[j - k],
                        number[i] + dp[j - k]*(10**k)) #t 为他们两能拼出来的最大的数
                if t > dp[j]:
                    dp[j] = t
        if number[i] > dp[k]: #最后再考察 k 位的情况
            dp[k] = number[i]
for j in range(m,0,-1):
    if dp[j] != 0: #print 第一个不为 0 的数（小心！）
        print(dp[j])
        break
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



02811: 熄灯问题

brute force, http://cs101.openjudge.cn/practice/02811

思路：因为只要确定了第一行的开关情况，就可以判断出第一行的灯的状态，进而决定第二行、第三行、...、一直到最后一行的开关情况和灯的状态，因此只需要对第一行的开关情况枚举即可（区区 64 种情况）~

代码：

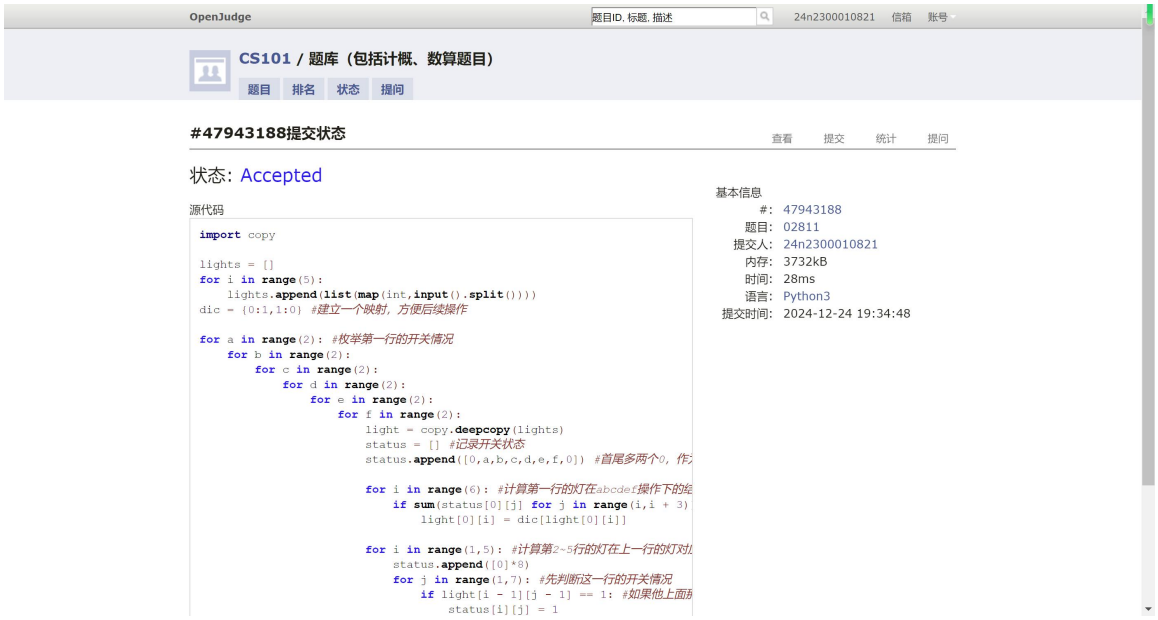
```python
```
import copy

lights = []
for i in range(5):
    lights.append(list(map(int,input().split())))
dic = {0:1,1:0} #建立一个映射，方便后续操作

for a in range(2): #枚举第一行的开关情况
    for b in range(2):

```
        for c in range(2):
            for d in range(2):
                for e in range(2):
                    for f in range(2):
                        light = copy.deepcopy(lights)
                        status = [] #记录开关状态
                        status.append([0,a,b,c,d,e,f,0]) #首尾多两个0，作为保护圈，不用分情况

                        for i in range(6): #计算第一行的灯在abcdef操作下的结果
                            if sum(status[0][j] for j in range(i,i + 3)) % 2 == 1:
                                light[0][i] = dic[light[0][i]]

                        for i in range(1,5): #计算第2~5行的灯在上一行的灯对应的操作下的结果
                            status.append([0]*8)
                            for j in range(1,7): #先判断这一行的开关情况
                                if light[i - 1][j - 1] == 1: #如果他上面那个灯还亮着，就需要按他
                                    status[i][j] = 1
                            for j in range(6): #再根据这一行的开关情况确定这一行的灯的情况
                                if (sum(status[i][j] for j in range(j,j + 3)) + status[i - 1][j + 1]) % 2 == 1:
                                    light[i][j] = dic[light[i][j]]

                        if light[4] == [0]*6: #最后，检查最后一行的灯是否全部熄灭即可（因为前面几行已经全部熄灭了）
                            for i in range(5):
                                print(" ".join(map(str,status[i][1:7])))
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



08210: 河中跳房子

binary search, greedy, http://cs101.openjudge.cn/practice/08210/

思路：主体思路是用两分查找的方法来限制最长可能的最短跳跃距离的范围；
具体而言，一开始的限制范围是[0，L]的全区间，将这个区间划分为一半，并利用 check 函数（原理后面讲）考虑其半程的地方(L//2)是否合题；若不合题则把范围缩小到[0，L//2]，若合题则把范围缩小到[L//2，L]；之后一直重复这个操作，即可找到最终的那一个夹在所有区间内的唯一合乎题意的数~
（p.s.check 函数的原理就是，我们先以 0 为基点，如果从基点到第 1 块岩石的距离小于这个最短跳跃距离，则移除第 1 块岩石；再看接下来那块岩石，如果他到 0 的距离还是小于最小跳跃距离，就继续移除；直至找到一块距离基点超过最小跳跃距离的岩石，保留这块岩石，并将它作为新的基点，再重复前述过程即可；如果最终需要移除的石头数大于 m 则说明这个距离不对，反之则说明是对的）

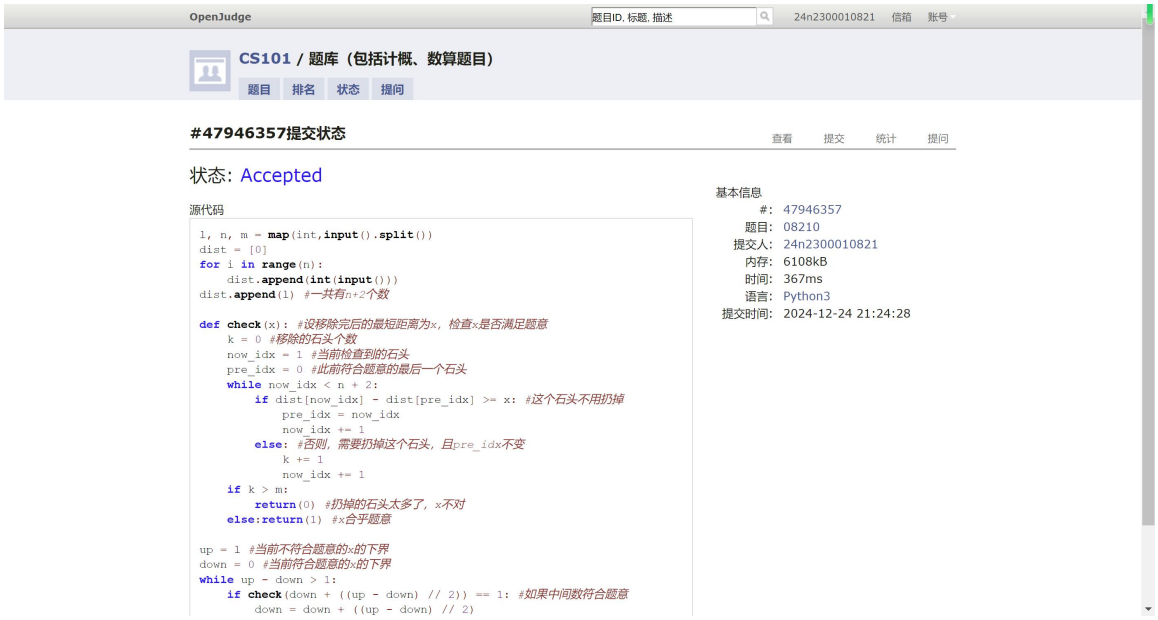
代码：

```python

```

```
l, n, m = map(int,input().split())
```

```
dist = [0]
for i in range(n):
    dist.append(int(input()))
dist.append(l) #一共有 n+2 个数

def check(x): #设移除完后的最短距离为 x，检查 x 是否满足题意
    k = 0 #移除的石头个数
    now_idx = 1 #当前检查到的石头
    pre_idx = 0 #此前符合题意的最后一个石头
    while now_idx < n + 2:
        if dist[now_idx] - dist[pre_idx] >= x: #这个石头不用扔掉
            pre_idx = now_idx
            now_idx += 1
        else: #否则，需要扔掉这个石头，且 pre_idx 不变
            k += 1
            now_idx += 1
    if k > m:
        return(0) #扔掉的石头太多了，x 不对
    else:return(1) #x 合乎题意

up = l #当前不符合题意的 x 的下界
down = 0 #当前符合题意的 x 的下界
while up - down > 1:
    if check(down + ((up - down) // 2)) == 1: #如果中间数符合题意
        down = down + ((up - down) // 2)
    else: #如果中间数不符合题意
        up = down + ((up - down) // 2)
    continue
print(down)
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



2. 学习总结和收获

<mark>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。</mark>

这次作业感觉还是相当有难度的。上来的第一道题感觉就是一道本质难度不算太高，但是相当耗费时间的一道题。思路不难想到，主要就是先尽量找出真币（经过多轮筛选），接着再看看假币到底是谁，以及孰轻孰重；但是有个别边缘样例相当的阴间，我提交了几次都一直在 wa，但是难以想到它到底是在哪些情况下会无法筛选出假币是谁。。直到看了测试数据才意识到问题；如果是在考场上恐怕就做不出来了呜呜呜...

第二题倒是一个比较偏模板的题，直接用常规的 dfs 模版套即可；需要注意的是，因为如果直接对每个方格 dfs 的话，会多次计算相同的子问题，因此会浪费大量的时间，所以可以建立一个 dp 列表来记录之前所计算过的 dfs 结果，总而节约时间~

第三题的螃蟹采蘑菇真的是这次作业里难得的一道水题了，只需要按照螃蟹是 horizontal 的还是 vertical 的情况进行分类讨论就行了，剩下的完全套 bfs 方法的 deque 模版即可，完全改都不带改的~要是机考也有这么简单就好了 qwq（刚开始学的时候感觉 bfs 和 dfs 是最难的，思维模式让人捉摸不透，结果现在题目做多了之后，反而感觉这两类能套模板的题目才是最简单的，因为几乎不需要动脑子哈哈哈）

第四题也蛮有意思的，我一开始以为这就是一道简单的 dp 题，直接用一个一维的 dp 列表（第 i 个数是当前的数中能组成的 i 位的最大的数）来递归即可（如下图）：

```python
m = int(input())
n = int(input())
numbers = list(map(int,input().split()))
dp = [0]*(m + 1) #第i个数是当前的数中能组成的i位的最大的数
for i in range(n):
    k = len(str(numbers[i]))
    if k <= m:
        for j in range(m,k,-1): #一维dp一定要倒序!
            if dp[j - k] != 0: #如果此前已经有j-k位的数了
                t = max(numbers[i]*(10**(j - k)) + dp[j - k],
                        numbers[i] + dp[j - k]*(10**k)) #t为他们两能拼出来的最大的数
                if t > dp[j]:
                    dp[j] = t
        if numbers[i] > dp[k]: #最后再考察k位的情况
            dp[k] = numbers[i]
for j in range(m,0,-1):
    if dp[j] != 0: #print第一个不为0的数（小心！）
        print(dp[j])
        break
```

结果在多次 wa 之后，我才意识到问题没那么简单：这种方法有个 critical flaw，就是他所构造出来的数里面的组成部分在最初的 list 里都是相邻的。。（只能构造出一些特定顺序的数）因此，如果想要沿用这个方法，就必须在一开始就对整个 list 里的数按照特定的顺序进行排列才行！想到这一点后，问题就简单一些了，也就可以解决了。不过题解里的“list.sort(key=lambda x:x*20,reverse=True”方法我还是第一次见，蛮有意思的，很有收获~

第五题的话，我个人则认为是一道极其困难的题目，如果不看题目提示直接想的话应该是非常难想到的。看了题目提示之后，我才稍微有点思路；一开始我想用 brute force 暴力枚举，但是那样很快就 TLE；于是我试图优化枚举的情况数量。思来想去之后，我意识到后几行的开关情况其实并不本质，因为只要确定了第一行的开关情况，就能够确定第一行的灯的状态，进而能够确定接下来每一行的开关情况，进而确定每一行的灯的状态。因此，我们只需要对第一行的开关情况的 64 种情况进行讨论即可！

想到这里之后，这道题基本就迎刃而解了。但即便如此，因为各种原因，比如代码太长，导致里面有些字母或者变量打错的时候很难 debug 出来；以及因为我对深拷贝不够了解，所以导致我这道题从想到思路到 AC 仍然花了不少时间。总而言之，感觉这是一道非常难的题目，希望机考没有这么难呜呜呜

（下面我稍微总结了一下深拷贝的用途：）

在 Python 中，**深拷贝**（`deepcopy`）是对对象的完整复制，包括所有嵌套对象的复制。与 **浅拷贝**（`copy`）不同，深拷贝会递归地复制对象及其内部所有的子对象，因此拷贝后的对象与原对象完全独立，修改深拷贝后的对象不会影响原对象，也不会影响原对象的子对象。

浅拷贝与深拷贝的区别:

- **浅拷贝**：复制的是对象的引用，而不是对象本身。如果原对象包含其他对象（如列表中的列表），那么这些子对象仍然是共享的，修改子对象会影响到原对象。
- **深拷贝**：递归地复制所有的对象，包括对象内部的子对象。因此，深拷贝创建了一个与原对象完全独立的新对象。

`copy.deepcopy` 的使用:

`copy.deepcopy(X)` 用于创建对象 `X` 的一个深拷贝。`deepcopy` 会创建一个新对象，并递归地复制 `X` 内部所有的元素和嵌套结构，确保新对象与原对象在任何层级上都是完全独立的。

示例代码:

python 复制代码

```python
import copy
```

示例代码:

python 复制代码

```python
import copy

# 原始对象
original = [1, 2, [3, 4]]

# 使用浅拷贝
shallow_copy = copy.copy(original)

# 使用深拷贝
deep_copy = copy.deepcopy(original)

# 修改嵌套列表的元素
original[2][0] = 99

# 打印查看结果
print("原始对象:", original)  # [1, 2, [99, 4]]
print("浅拷贝对象:", shallow_copy)  # [1, 2, [99, 4]]  浅拷贝中嵌套列表也改变
print("深拷贝对象:", deep_copy)  # [1, 2, [3, 4]]  深拷贝中的嵌套列表没有改变
```

解释：

- `original` 是一个包含嵌套列表的列表，`[1, 2, [3, 4]]`。
- `shallow_copy` 是对 `original` 的浅拷贝，它与 `original` 共享内部的嵌套列表 `[3, 4]`。
- `deep_copy` 是对 `original` 的深拷贝，它创建了一个独立的嵌套列表 `[3, 4]`，与原始列表完全独立。

当我们修改 `original` 中嵌套列表的元素时，`shallow_copy` 中的嵌套列表也会受到影响，因为它们共享相同的子对象。而 `deep_copy` 中的嵌套列表则没有受到影响，因为它是完全独立的。

为什么需要深拷贝：

- **避免共享引用**：有时你希望一个对象的拷贝不会影响到原对象（尤其是当对象内部包含可变元素时）。例如，在处理嵌套结构时，如果只进行浅拷贝，修改嵌套结构的元素会同时影响原对象及其拷贝。
- **数据隔离**：深拷贝确保你对一个对象的修改不会影响其他地方的引用，适用于需要完全独立操作的情况。

（总结一下：copy.deepcopy(X) 创建了 X 的深拷贝，确保完全独立于原对象；而 deepcopy 与 shallowcopy 的区别在于，浅拷贝（copy.copy(X)）只复制了引用，而深拷贝复制了整个对象及其嵌套的所有部分）

第六题的话感觉也很难，方法光靠自己很难想出来；我也是看了题解之后才恍然大悟，居然有这么神奇（厚颜无耻）的方法，有一种空手套白狼的感觉，学会了哈哈哈~

最后，希望自己能在机考里稳定发挥，把自己会做的题都做出来吧，问心无愧于自己这一个学期的努力就好啦~