棋盘问题

def count\_ways(board, row, k, used\_cols):

if k == 0:

return 1

if row == len(board):

return 0

ways = 0

for col in range(len(board)):

if board[row][col] == '#' and col not in used\_cols:

used\_cols.add(col)

ways += count\_ways(board, row + 1, k - 1, used\_cols)

used\_cols.remove(col) # Backtrack

# Also consider skipping the current row

ways += count\_ways(board, row + 1, k, used\_cols)#此时used\_cols中不含有新加的

return ways

while True:

n\_k = input().strip()

if n\_k == "-1 -1":

break

n, k = map(int, n\_k.split())

board = [input().strip() for \_ in range(n)]

print(count\_ways(board, 0, k, set()))

为啥左面对右边不对：

1. 增加**跳跃机制**，避免漏数如

10000

11000

01100

数两个时候可以是（0，0）（1，1）也可以是（0，1）（1，0）

1. **一次性数完**避免重复代入（这种和起始的位置其实无关系，不如把ij遍历的过程拿回到函数里面去
2. 只用**行数（可以取不到）**进行递归的策略要**比必须取到该行的严格限制**更有优势

while True:

a,b=map(int,input().split())

def counts(i,j,row,n):

s=0

if n==0:

return 1

for k in range(i,a):

for l in range(a):

if l not in row and mapi[k][l]=="#":

s+=counts(k,l,row+[l],n-1)

return s

if a==b==-1:

break

mapi=[input() for i in range(a) ]

COU=0

for i in range(a):

for j in range(a):

if mapi[i][j]=="#":

COU+=counts(i,j,[j],b-1)

print(COU)

复习：欧几里得算法最大公约数；def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

**用heap确定索引并在有限长度内找出所有的最小情况**

06648:Sequence

总时间限制: 3000ms 内存限制: 65536kB

[OpenJudge - 06648:Sequence](http://cs101.openjudge.cn/practice/06648/)

描述

给定m个数字序列，每个序列包含n个非负整数。我们从每一个序列中选取一个数字组成一个新的序列，显然一共可以构造出n^m个新序列。接下来我们对每一个新的序列中的数字进行求和，一共会得到n^m个和，请找出最小的n个和

输入

输入的第一行是一个整数T，表示测试用例的数量，接下来是T个测试用例的输入

每个测试用例输入的第一行是两个正整数m（0 < m <= 100）和n(0 < n <= 2000)，然后有m行，每行有n个数，数字之间用空格分开，表示这m个序列

序列中的数字不会大于10000

输出

对每组测试用例，输出一行用空格隔开的数，表示最小的n个和

样例输入

1

2 3

1 2 3

2 2 3

样例输出

3 3 4

import heapq

for \_ in range(int(input())):

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

\*ous,=(map(int,input().split()))

ous.sort()

for I in range(m-1):

\*se,=(map(int,input().split()))

se.sort()

mi=[(ous[0]+se[i],i,0)for i in range(n)]

heapq.heapify(mi)

result=[]

while len(result)<n:

cou,i,j=heapq.heappop(mi)

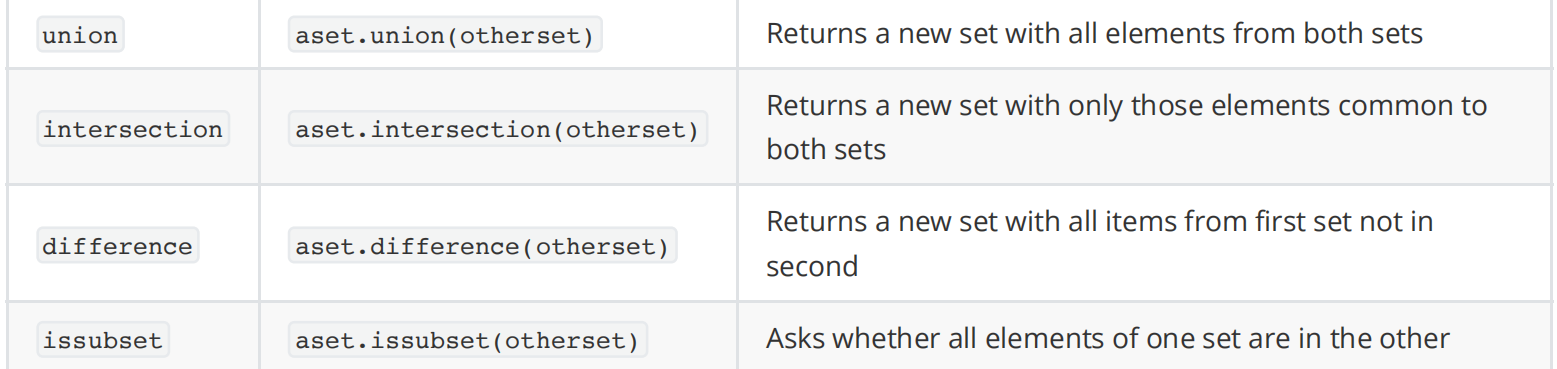
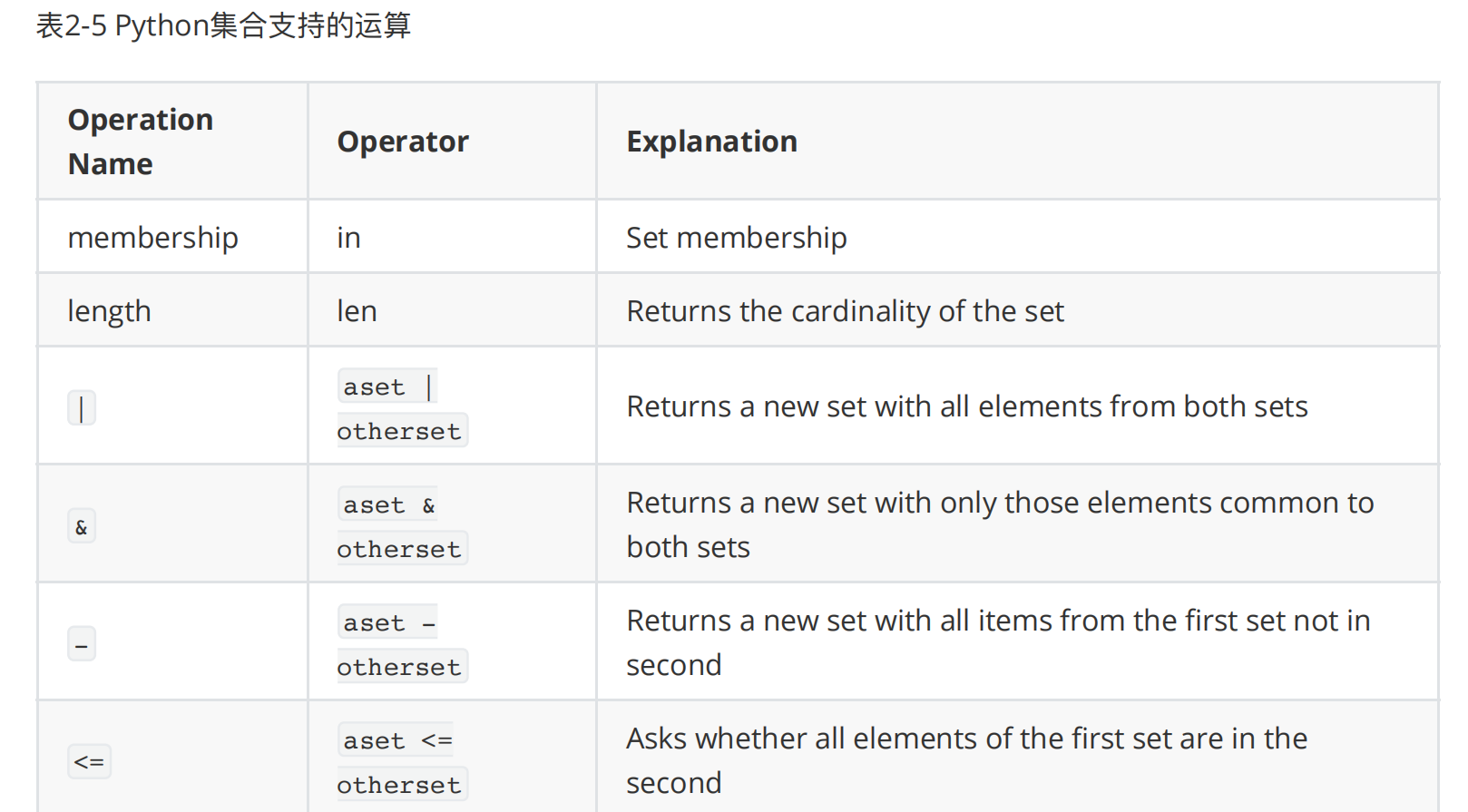
if j+1<n:

heapq.heappush(mi,(ous[j+1]+se[i],i,j+1))#让i证明自己足够小，有能力让和足够小，此时可以往后托付更多的j

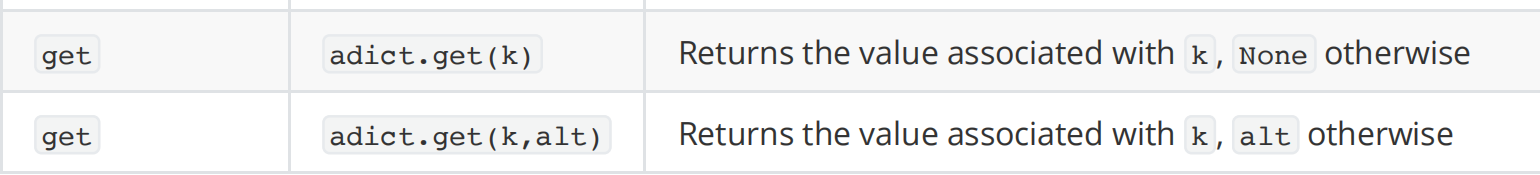
result.append(cou)

ous=result

print(" ".join(map(str,ous)))



字典



Namedtuple

namedtuple 是 Python 的 collections 模块中提供的⼀种数据结构，它扩展了标准元组（tuple）的功能。与普通元组不同的是， namedtuple 允许你通过名称访问元素，⽽不仅仅是通过索引，这使得代码更具可读性和⾃解释性。

namedtuple 特别适合⽤来创建轻量级的、不可变的数据对象，当你有⼀组相关的数据项并且希望以⼀种更加⾯向对象的⽅式访问这些数据时， namedtuple 是⼀个⾮常⽅便的选择。

示例继承

from collections import namedtuple

# 定义⼀个名为 'Car' 的 namedtuple 类型，它有两个字段：'make' 和 'model'

Car = namedtuple('Car', ['make', 'model'])

# 创建⼀个 Car 实例

my\_car = Car(make='Toyota', model='Corolla')

# 访问元素

print(my\_car.make) # 输出: Toyota

print(my\_car.model) # 输出: Corolla

# 也可以像普通 tuple 那样⽤索引访问

print(my\_car[0]) # 输出: Toyota print(my\_car[1]) # 输出: Corolla

# 尝试修改元素会引发错误，因为 namedtuple 是不可变的

# my\_car.make = 'Honda' # 这⾏代码将导致 AttributeError

链表  
# Definition for singly-linked list.

# class ListNode:

# def \_\_init\_\_(self, val=0, next=None):

# self.val = val

# self.next = next

class Solution:

def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:

**left=ListNode(next=head)#在已知的链表前创建了一个哨兵节点**

return left

定义类的时候少用点新常数变量

class MinStack(object):

def \_\_init\_\_(self):

"""

initialize your data structure here.

"""

self.stack = []

def push(self, x):

"""

:type x: int

:rtype: void

"""

if not self.stack:

self.stack.append((x, x))

else:

self.stack.append((x, min(x, self.stack[-1][1])))

def pop(self):

"""

:rtype: void

"""

self.stack.pop()

def top(self):

"""

:rtype: int

"""

return self.stack[-1][0]

def getMin(self):

"""

:rtype: int

"""

return self.stack[-1][1]

**一个超复杂的group排序L:用group做单位不用人做单位**

[OpenJudge - 27925:小组队列](http://cs101.openjudge.cn/practice/27925/)

from collections import deque # 时间: 105ms

# Initialize groups and mapping of members to their groups

t = int(input())

groups = {}

member\_to\_group = {}

for \_ in range(t):

members = list(map(int, input().split()))

group\_id = members[0] # Assuming the first member's ID represents the group ID

groups[group\_id] = deque()

for member in members:

member\_to\_group[member] = group\_id

# Initialize the main queue to keep track of the group order

queue = deque()

# A set to quickly check if a group is already in the queue

queue\_set = set()

while True:

command = input().split()

if command[0] == 'STOP':

break

elif command[0] == 'ENQUEUE':

x = int(command[1])

group = member\_to\_group.get(x, None)

# Create a new group if it's a new member not in the initial list

if group is None:

group = x

groups[group] = deque([x])

member\_to\_group[x] = group

else:

groups[group].append(x)

if group not in queue\_set:

queue.append(group)

queue\_set.add(group)

elif command[0] == 'DEQUEUE':

if queue:

group = queue[0]

x = groups[group].popleft()

print(x)

if not groups[group]: # If the group's queue is empty, remove it from the main queue

queue.popleft()

queue\_set.remove(group)