**目录：**

Page 1：输入；输出；

Page 2：生成器表达式；列表推导式；字典推导式；字符串；列表

Page 3：字典；集合；ASCII表

Page 4：ASCII表；包(math; itertools; deque; defaultdict)

Page 5：包(heapq; lru\_cache; copy)；dfs(迷宫的可行路径)

Page 6：dfs(最大连通域面积；指定步数迷宫)

Page 7：dfs(矩阵最大权值)

Page 8：dfs(迪杰斯特拉算法)；bfs(寻宝)

Page 9：bfs(迷宫最短路径；矩阵中的块)

Page 10：bfs(多终点迷宫问题)；螺旋矩阵

Page 11：；动态规划（双dp；拦截导弹；0-1背包；最长公共子序列）

Page 12：动态规划（多重背包）

Page 13：动态规划（完全背包）；贪心区间问题（合并区间；无重叠区间）

Page 14：充实的寒假生活；划分字母区间；heapq(毒药)

Page 15：；Bisect；递归（汉诺塔）

Page 16：递归（全排列）；排序

Page 17：双指针（回文串；移除重复元素；滑动窗口(最长无重复子序列串)）

Page 18：Stack（快速堆猪）

Page 19：找平方数；筛选质数；前缀和（flower）；求最大值；数字转列表

Page 20：Kadane's algorithm（⼀维整数数组中找到有最⼤和的连续⼦数组）；语言转换问题

**输入**

* 输入k行的矩阵：**matrix = [list(map(int, input().split())) for \_ in range(k)]**
* 输入一个n\*n的矩阵并且把每一个初始值都赋值为0：**D = [[0]\*n for \_ in range(n)]**
* 加保护圈的矩阵：**（n行m列）**

**M = [[-1] \* (m + 2)] + [[-1] + list(map(int, input().split())) + [-1] for i in range(n)] + [[-1] \* (m + 2)]**

或者这样：

**t=[[0 for \_ in range(m+2)] for \_ in range(n+2)]**

**for i in range(1,n+1):**

**t[i][1:-1]=map(int,input().split())**

* 一次性读取所有输入：

**import sys**

**input = sys.stdin.read**

**data = input().split()** # 读⼊所有数据并分割为列表

一次性输出所有数据：（代替print）**sys.stdout.write('\n'.join(map(str, results)) + '\n')**

* 没有明确指示停止的输入：

**while True:**

**try:**

**a,b=input().split()**

**except EOFError:**

**Break**

输出

* 基本模板：f"字符串 {表达式} 字符串": **print(f"My name is {name} and I am {age} years old.")**
* 三元运算符：**print(f"Adult status: {'Yes' if age >= 18 else 'No'}")**
* 保留小数输出：’%.2f’%\_\_（会四舍五入，什么不会四舍五入？）：例：print(‘%.2f’%3.1415926)=3.14； **print(f"{value:.2f}")**；百分数：**print(f"Percentage: {value:.2%}")**
* 字符串输出：**print("\n ".join(map(str,new\_list))) \n**是换行；先把列表里的元素转换为字符串
* 列表式输出：**list(map(print, my\_list))**
* 输出矩阵：

**for i in range(1,n+1):** 这是在有保护圈的情况下

**print(' '.join(map(str,matrix[i][1:-1])))**（注：str的作用是先化为字符串）

生成器表达式：

**squared\_gen = (x \* x for x in range(5))**

**for num in squared\_gen:**

**print(num)**

与for循环显式循环对比：

**squared\_list = []**

**for x in range(5):**

**squared\_list.append(x \* x)**

**for num in squared\_list:**

**print(num)**

列表推导式：用于基于现有的可迭代对象创建新的列表。通常用来替代传统的 for 循环

例：**even\_squares = [x \* x for x in range(10) if x % 2 == 0]**

也可在列表推导式中应用多个for循环：**pairs = [(x, y) for x in range(3) for y in range(3)]**

字典推导式：**numbers = [1, 2, 3, 4, 5]；squared\_dict = {n: n\*\*2 for n in numbers}；**

字符串：

* 大小写转换：**str.upper(); str.lower(); str.swapcase()**大小互换; **str.capitalize()**首字母大写;

**str.title()**单词首字母大写

* **str.replace(old, new, count) count**的作用是只替换前count次出现的old
* **str.count(substring, start, end)**
* **str.startswith('xxxx')** 用于判断字符串是否以指定的前缀开头，返回值为 True 或 False。
* **str.strip('xxxx')**；移除左边为lstrip(); 右边为rstrip()

列表：

* **enumerate**用法：

**for idx, value in enumerate(my\_list, start=1):**

**print(idx, value)**

用enumerate创建字典：**word\_dict={idx: word for idx, word in enumerate(my\_list)}**

与if搭配实现查找：**fruits = ['apple', 'banana', 'cherry', 'orange']**

**for index, fruit in enumerate(fruits):**

**if 'a' in fruit:**

**print(f"Found fruit with 'a' at index {index}: {fruit}")**

* **sort:**

**sorted\_T = sorted(T, key=lambda x: (x[0], x[1]))**（先根据第一个元素排，再根据第二个）

按长度排序：**sorted\_words = sorted(words, key=len)**

* **my\_list.remove(2)** # 删除第一个元素2
* **removed\_element = my\_list.pop(2)** # 删除索引为 2 的元素
* **lambda**用法：**squared\_numbers = list(map(lambda x: x \*\* 2, numbers))**



字典：

* 创建字典：**my\_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}**
* 通过键来搜索值：**print(my\_dict['name'])** # 输出：Alice

**print(my\_dict.get('address', 'Not Found'))** # 输出：Not Found

* 通过值来搜索键

找到所有的键：**keys = [key for key, value in my\_dict.items() if value == search\_value]**

找到第一个符合条件的键：**key = next((key for key, value in my\_dict.items() if value == search\_value), None)**

* 添加或更新元素（键值对）：**my\_dict['age'] = 26 ；my\_dict['country'] = 'USA'**
* 向字典中某一个键下添加元素：

my\_dict = {'key1': [1, 2, 3], 'key2': [4, 5]}；添加：**my\_dict['key1'].append(4)**

* 删除键值对：**del my\_dict['city']** # 删除 'city' 键值对

**age = my\_dict.pop('age')**；print(age) # 输出：26

* 遍历字典：

# 遍历键：**for key in my\_dict:**

# 遍历值：**for value in my\_dict.values(): print(value)**

# 遍历键值对**for key, value in my\_dict.items(): print(f"{key}: {value}")**

* 字典排序：**sorted\_dict = dict(sorted(my\_dict.items(), key=lambda x: x[1], reverse=True))**

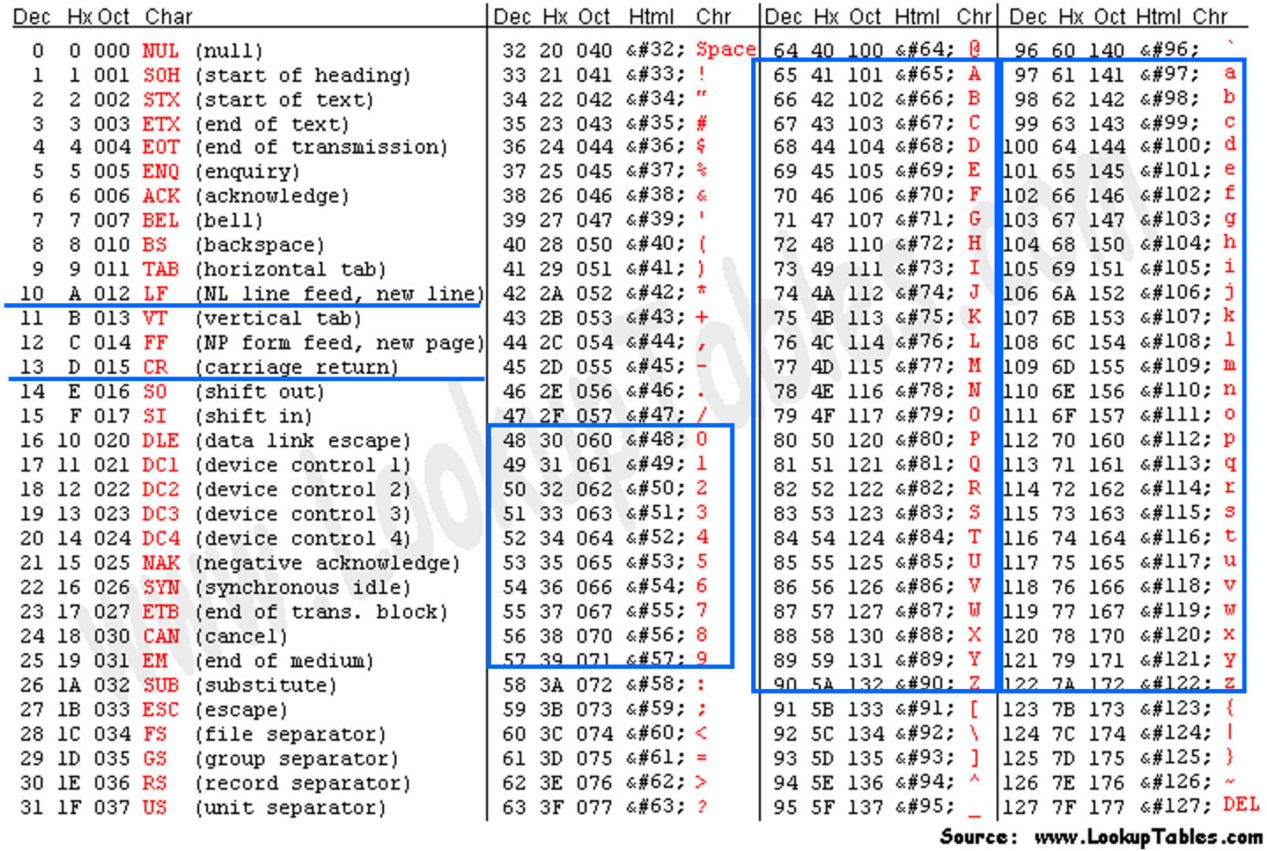
(sorted得到的只是一个列表，想要转化为字典则需要在外面加上dict)

集合：set() 用来创建集合时，它接受一个可迭代对象（如列表、元组、字符串等）

**my\_set = set([3, 4, 5, 6, 7])：**或**set={3,4,5,6,7}**;

**my\_set.add(6); my\_set.remove(2); my\_set.discard(10)**（若元素不在不会报错）

ASCII表：



# 获取字符的 ASCII 码——**ord()：ascii\_value = ord('A')；**print(ascii\_value) # 输出：65

# 获取 ASCII 码对应的字符——**chr()：char = chr(65)；**print(char) # 输出：A

包：

* **math.ceil()（向上取整）；math.floor()（向下取整）；math.sqrt()（开平方根）；math.trunc()（保留整数部分）**
* **Itertools: (import itertools)**对数据进行排列/组合/重复/筛选；

**A=itertools.count(start=0, step=1):**生成一个无限序列，从 start 开始， step 为步长递增

**A=itertools.chain(\*iterables):**将多个可迭代对象连接在一起，形成一个单一的迭代器。

ch = itertools.chain([1, 2, 3], ['a', 'b', 'c'])

print(list(ch)) (结果： [1, 2, 3, 'a', 'b', 'c'])

**A=itertools.permutations(iterable, r=None):**返回输入可迭代对象的所有 r 长度的排列

**perm = itertools.permutations([1, 2, 3], 2)**

**print(list(perm)) (结果： [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)])**

**A=itertools.combinations(iterable, r)**:返回输入可迭代对象的所有 r 长度的组合

comb = itertools.combinations([1, 2, 3], 2)

print(list(comb)) (结果： [(1, 2), (1, 3), (2, 3)])

**A=itertools.combinations\_with\_replacement(iterable, r)**：返回带有重复元素的所有 r 长度的组合（允许重复元素）（例如，有(1,1), (2,2), (3,3)）

其它：**from** **itertools import repeat; a=repeat(obj, time)**

**itertools.filterfalse(lambda x: x%2==0, range(10))**（过滤满足条件元素）；

* deque — 双端队列：**a=deque()**

**a.append(x)** ：将元素 x 添加到右端；**a.appendleft(x)** ：将元素 x 添加到左端。

**a.pop()** ：从右端弹出元素；**a.popleft()** ：从左端弹出元素。

**a.extend(iterable)** ：在右端添加多个元素；**a.extendleft(iterable)** ：在左端添加多个元素

* **defaultdict: (from collections import defaultdict)** 当访问一个不存在的键时，defaultdict 会自动为这个键创建一个默认值

d=defaultdict(int); d[‘a’]返回默认值0 ; d=defaultdict(list);d[‘a’]返回默认空列表[]; d=defaultdict(set); d[‘a’]返回空集合set()

常用来进行元素计数：

**word\_count = defaultdict(int)**

**words = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']**

**for word in words:**

**word\_count[word] += 1**

**print(word\_count)**# defaultdict(<class 'int'>, {'apple': 3, 'banana': 2, 'orange': 1})

（或**print(dict(word\_count)**)，则只生成字典）

* **heapq: import heapq**

**heapq.heappush(heap, item):** 压进堆的元素会自动从小到大排列

**heapq.heappop(heap)：**弹出并返回堆中最小的元素，同时保持堆的性质。

**heapq.heapify(my\_list)**把列表转为堆

**heapq.heappushpop(heap, item)：**将 item 插入堆后再弹出最小值，结合了heappush和 heappop

如果需要实现**最大堆**，可以通过取负值实现：

**data = [1, 3, 5, 7, 9]**

**max\_heap = []**

**for item in data:**

**heapq.heappush(max\_heap, -item)**  # 插入负值

**print(-heapq.heappop(max\_heap))** # 9，弹出最大值

* 用dfs做dp类的题，只要用这个就不会超时：

**from functools import lru\_cache**

**@lru\_cache(maxsize=None)**

* 拷贝：import copy

浅拷贝：**shallow\_copy = copy.copy(original)**

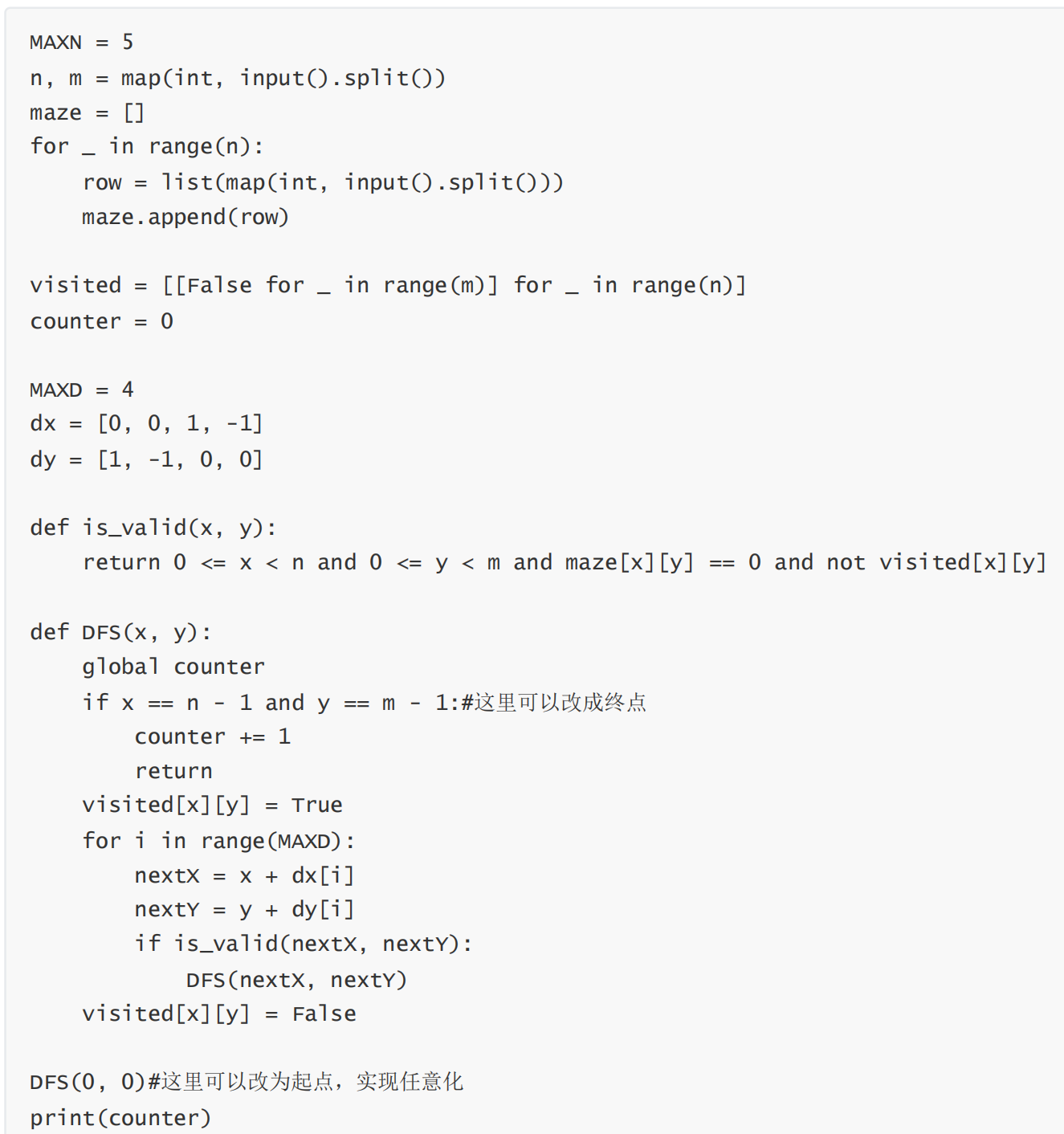
深拷贝：**deep\_copy = copy.deepcopy(original)**

算法类：

1. 搜索题

**dfs模板题：**

迷宫的可行路径数：



最大连通域面积：

directions=[(1,0),(-1,0),(1,1),(-1,1),(0,1),(0,-1),(-1,-1),(1,-1)]

def dfs(x,y):

cnt = 1

maze[x][y]='.'

for dx,dy in directions:

nx=x+dx

ny=y+dy

if 0<=nx<N and 0<=ny<M and maze[nx][ny]=='W':

cnt += dfs(nx,ny)

return cnt

T=int(input())

for \_ in range(T):

N,M=map(int,input().split())

maze=[list(input()) for \_ in range(N)]

max\_area=0

for i in range(N):

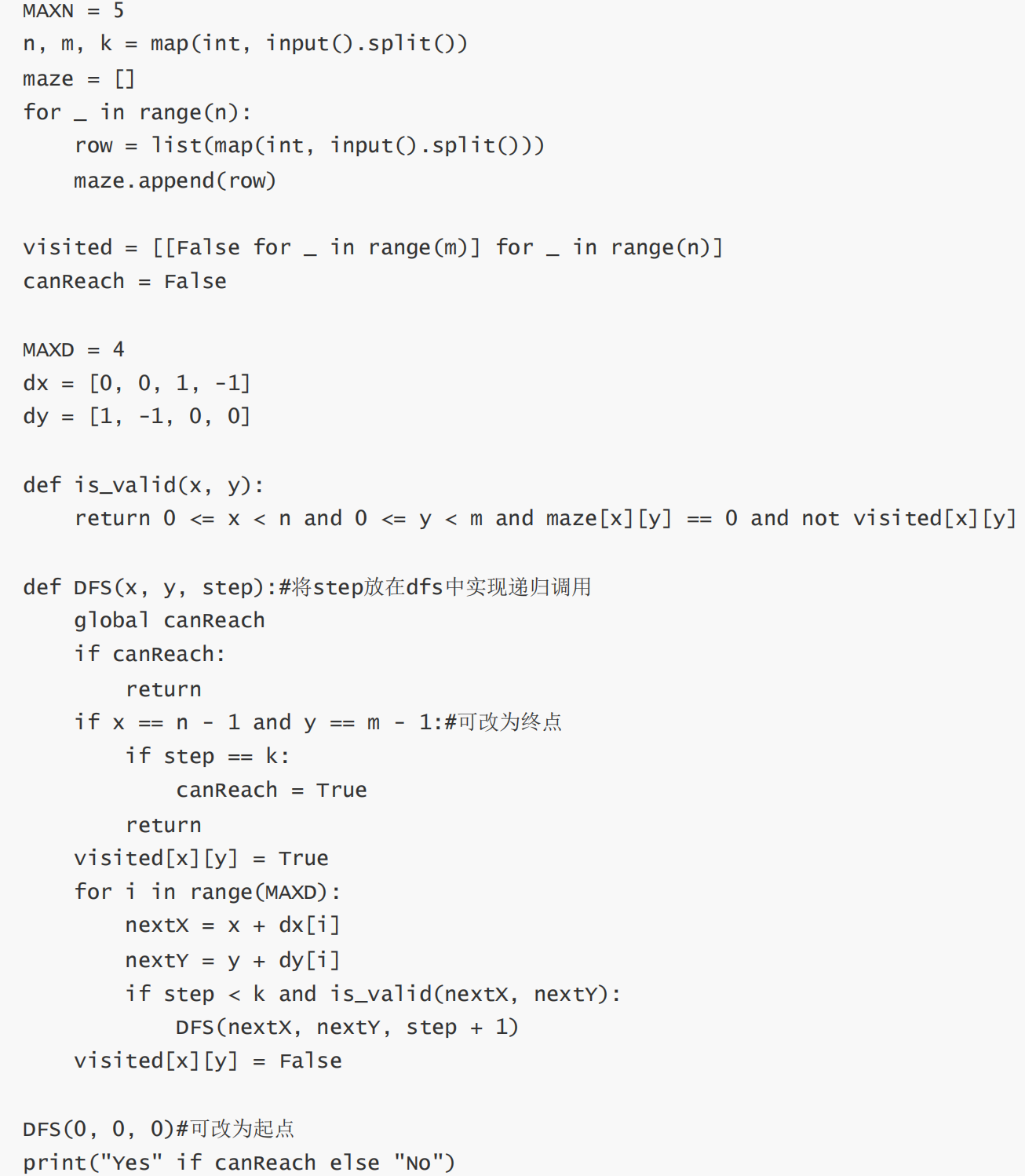
for j in range(M):

if maze[i][j]=='W':

max\_area=max(max\_area,dfs(i,j))

print(max\_area)

指定步数迷宫（能否从左上角到右下角）



矩阵最大权值（求路径与最大值，有回溯和visited的版本）

directions=[(1,0),(-1,0),(0,1),(0,-1)]

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

maze=[list(map(int,input().split())) for \_ in range(n)]

visited=[[False]\*m for \_ in range(n)]

max\_sum=-float('inf')

max\_path=[]

def dfs(x,y,cur\_sum,cur\_path):

global max\_sum

global max\_path

if x==n-1 and y==m-1:（终止条件）

if cur\_sum>max\_sum:

max\_sum=cur\_sum

max\_path=cur\_path[:]

return

for dx,dy in directions:

nx=x+dx

ny=y+dy

if 0<=nx<n and 0<=ny<m and visited[nx][ny]==False:

visited[nx][ny]=True

cur\_path.append((nx,ny))

cur\_sum+=maze[nx][ny]

dfs(nx,ny,cur\_sum,cur\_path)

（下面三行都是回溯）

cur\_sum-=maze[nx][ny]

cur\_path.pop()

visited[nx][ny]=False

visited[0][0]=True

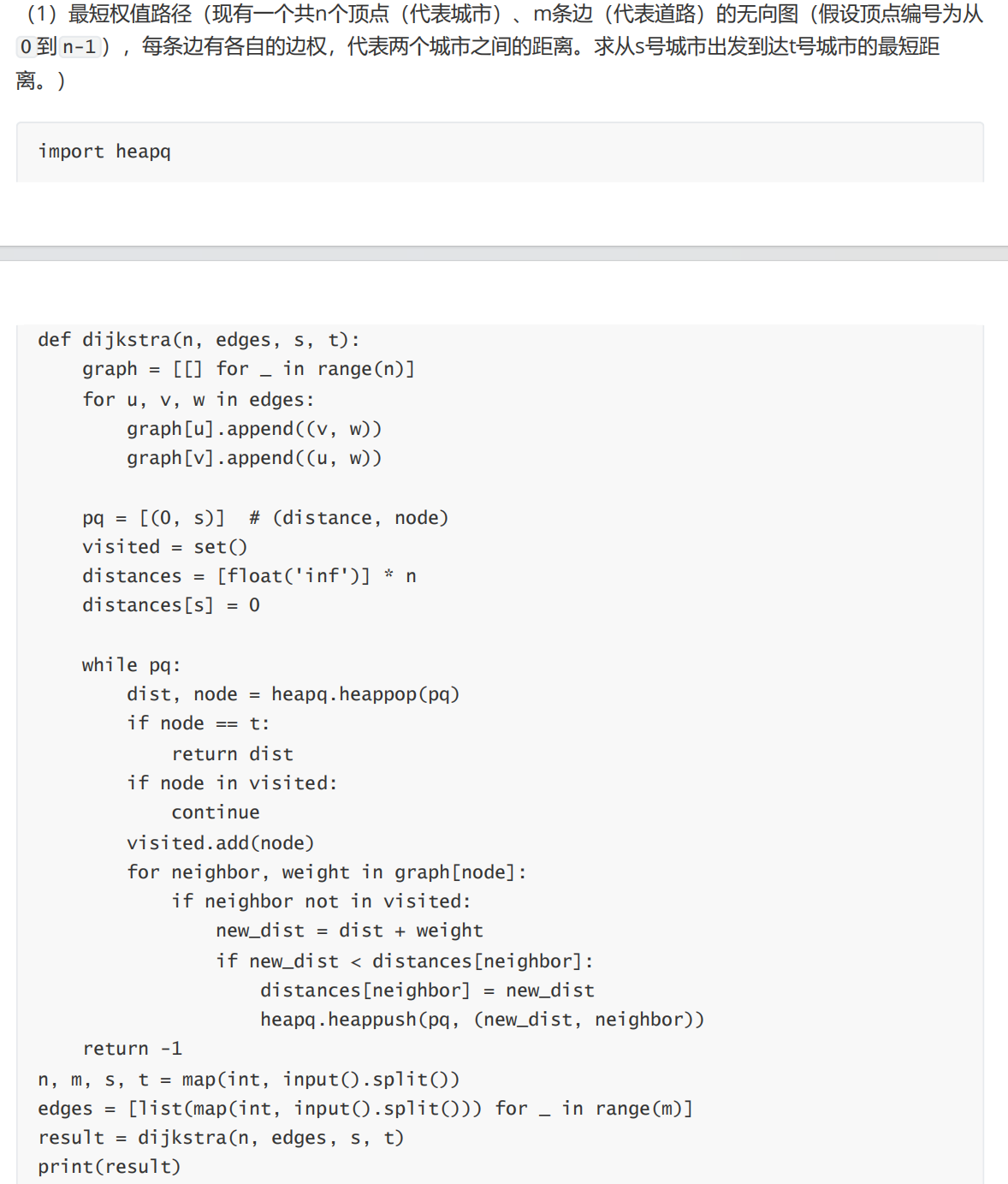
dfs(0,0,maze[0][0],[(0,0)])

for x,y in max\_path:

print(x+1,y+1)

迪杰斯特拉算法（最短路径；bfs+greedy）：

最短权值路径：现有一个共n个顶点（代表城市）、m条边（代表道路）的无向图（假设顶点编号为从0到n-1），每条边有各自的边权，代表两个城市之间的距离。求从s号城市出发到达t号城市的最短距离。



**bfs模板题：**

OJ19930：寻宝代码

from collections import deque

dx = [0, 0, 1, -1]

dy = [1, -1, 0, 0]

def bfs(x,y):

q=deque()

q.append((x,y))

inq = set()

inq.add((x,y))

step=0

while q:

for \_ in range(len(q)):

cur\_x, cur\_y = q.popleft()

if maze[cur\_x][cur\_y]==1:

return step

for direction in range(4):

nx = cur\_x + dx[direction]

ny = cur\_y + dy[direction]

if 1<=nx<=m and 1<=ny<=n and maze[nx][ny]!=2 and (nx,ny) not in inq:

inq.add((nx,ny))

q.append((nx,ny))

step += 1

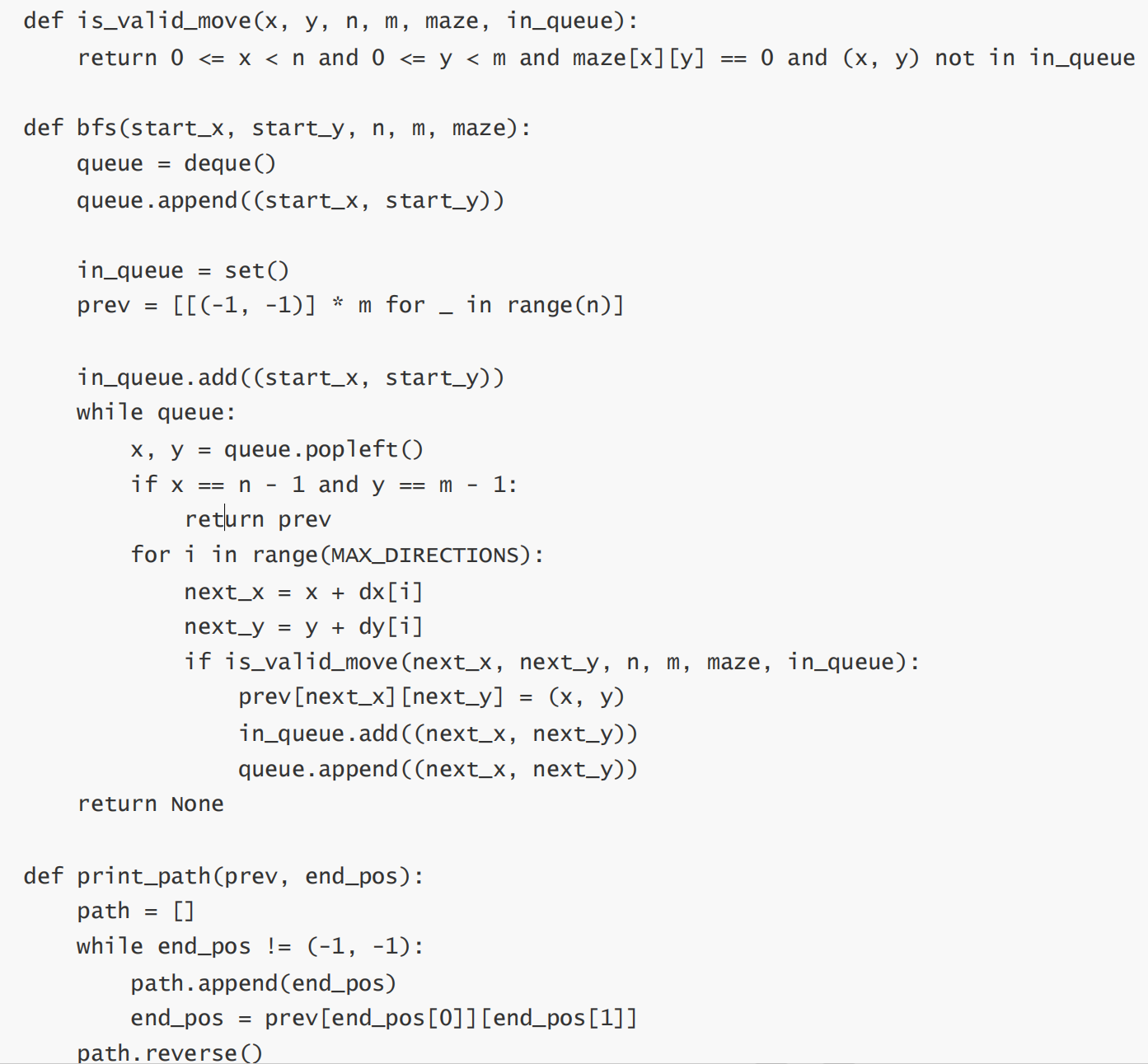
return 'NO'

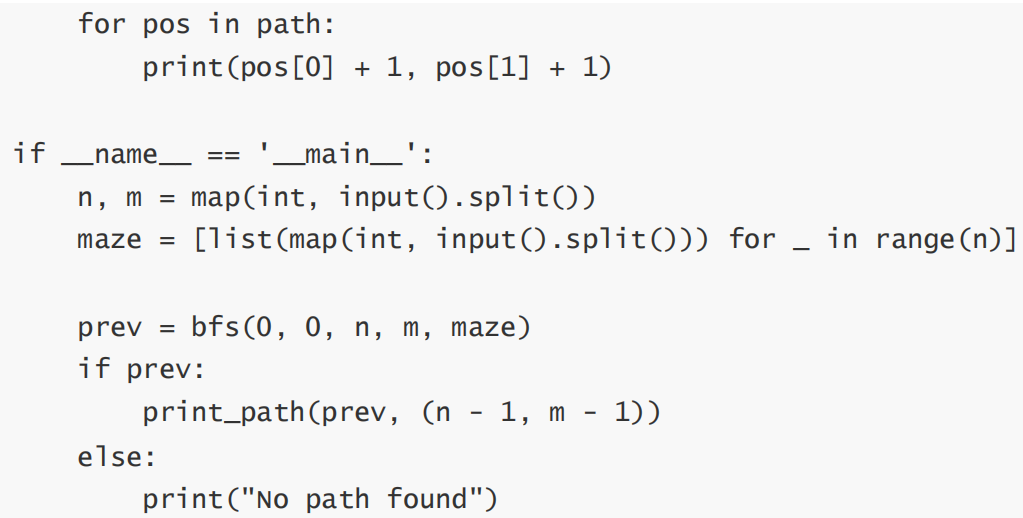
m, n = map(int, input().split())

maze = [[-1] \* (n + 2)] + [[-1] + list(map(int, input().split())) + [-1] for i in range(m)] + [[-1] \* (n + 2)]（m是行, n是列）

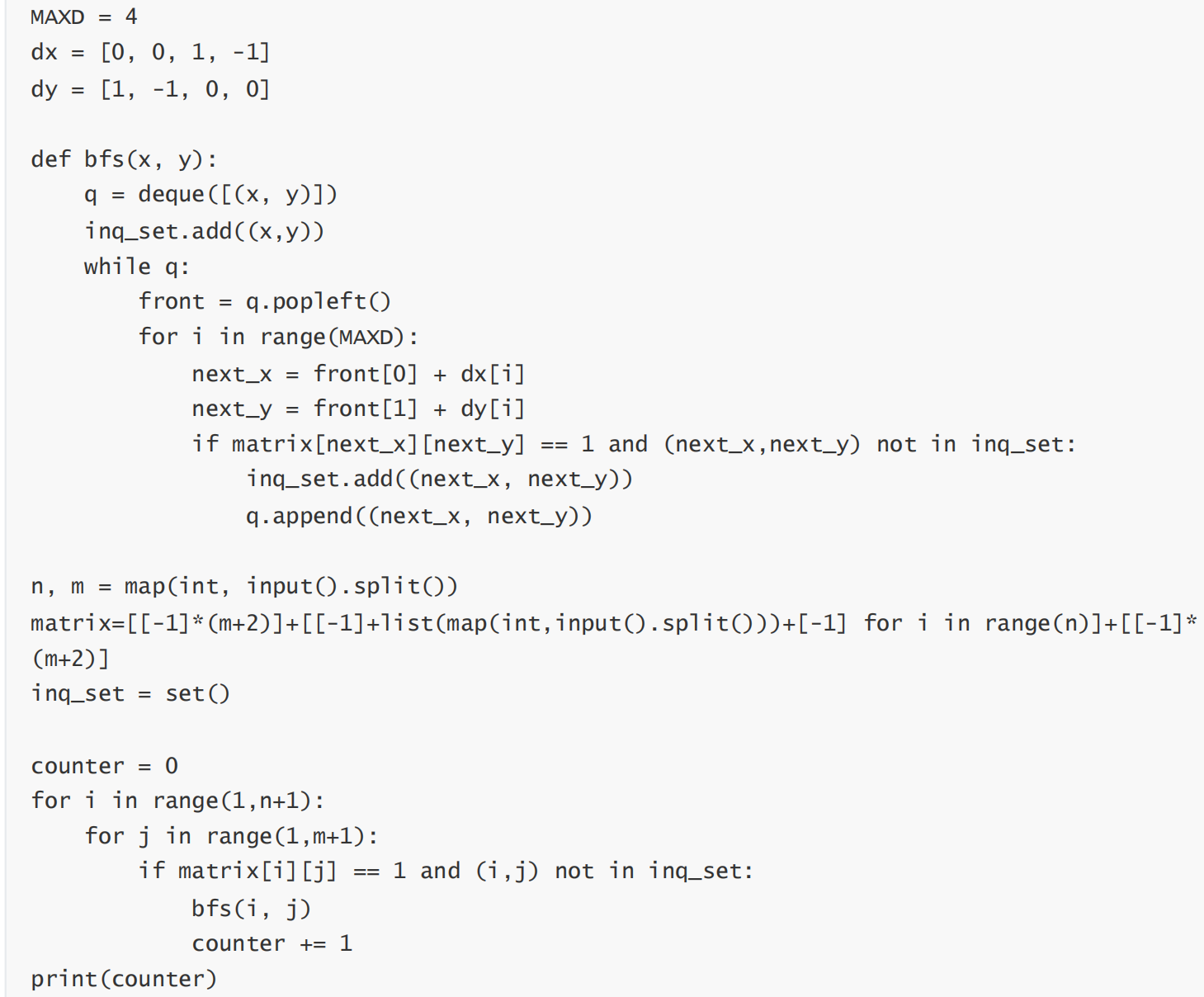
print(bfs(1,1))

迷宫最短路径：

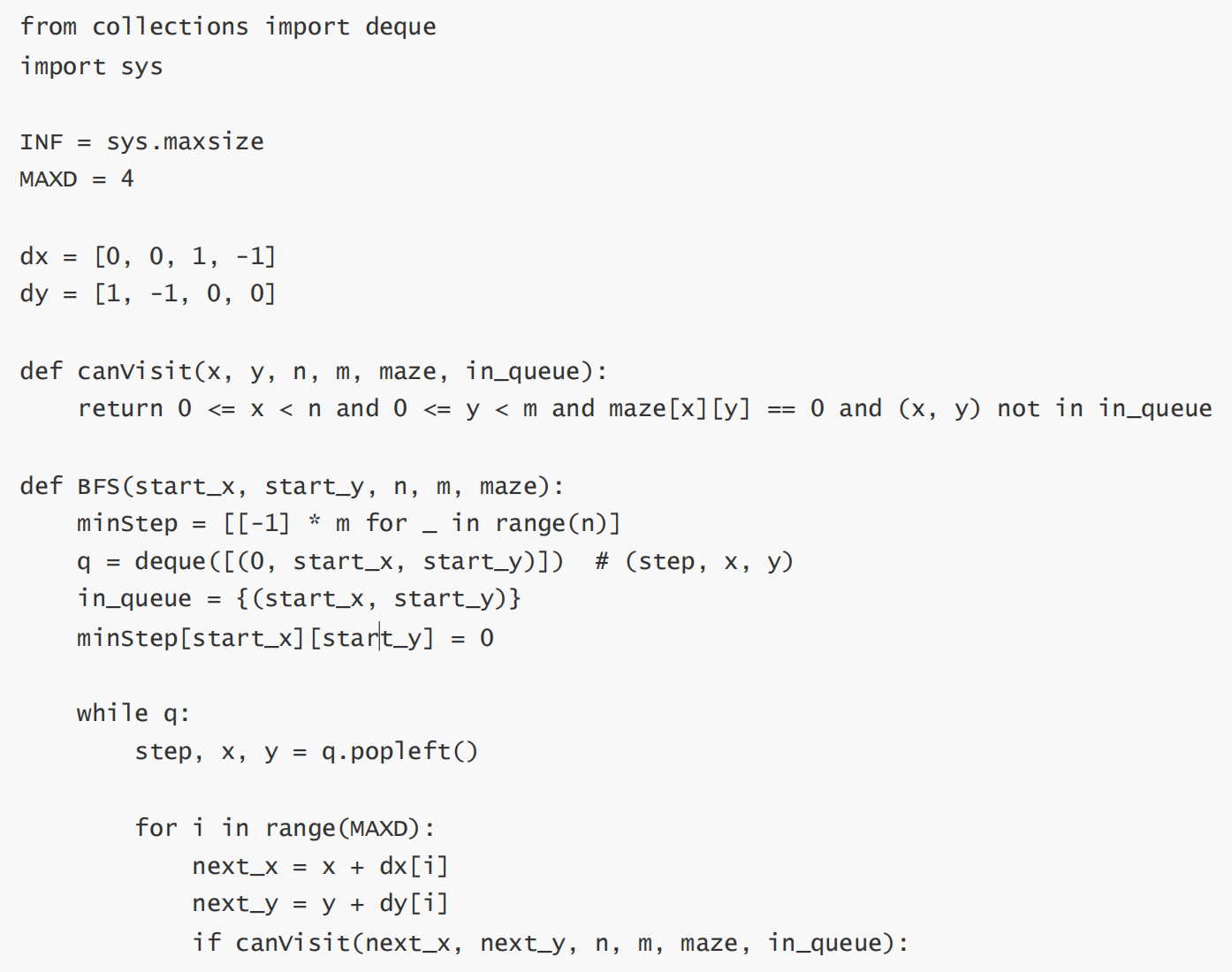


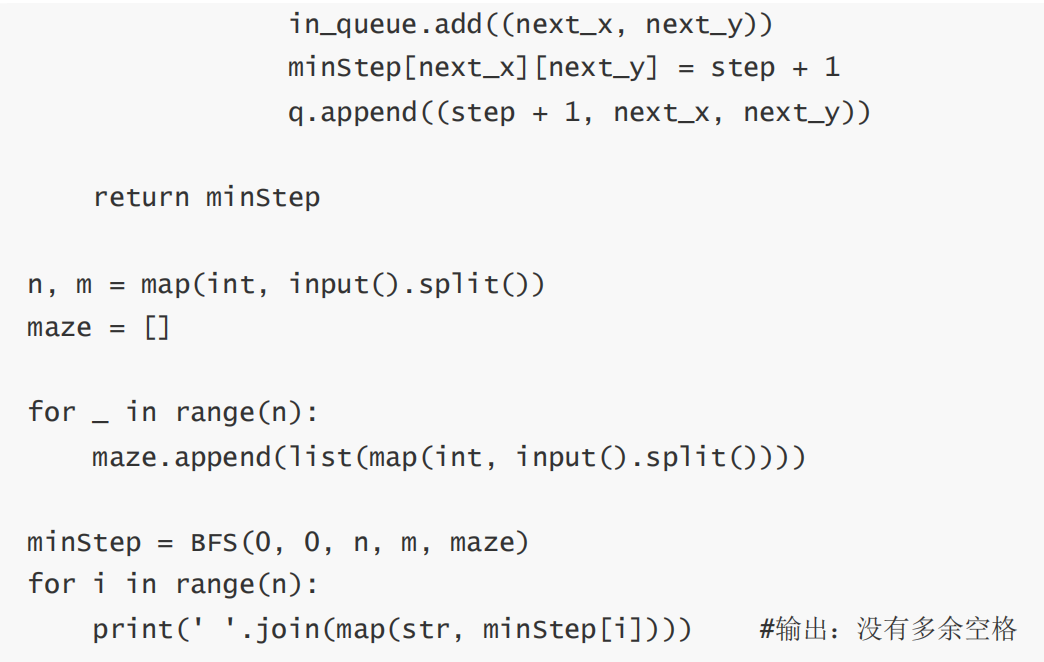


矩阵中的块：



多终点迷宫问题（现有一个 n\*m 大小的迷宫，其中 1 表示不可通过的墙壁， 0 表示平地。每次移动只能向上下左右移动一格，且只能移动到平地上。求从迷宫左上角到迷宫中每个位置的最小步数。输出n行m列个整数，表示从左上角到迷宫中每个位置需要的最小步数。如果无法到达，那么输出 -1 。注意，整数之间用空格隔开，行末不允许有多余的空格。）





螺旋矩阵：

n=int(input())

mx = [[401 for \_ in range(n+2)]for \_ in range(n+2)]

（因为1<=n<=20, 所以n\*n<=400；以及套保护圈）

for i in range(1,n+1):

mx[i][1:-1]=[0]\*n （把中间要赋值的那部分都赋为0）

directions=[[0,1],[1,0],[0,-1],[-1,0]]（排序是按向右向下向左向上来排的）

row=1 第一行

col=1 第一列

N=0

d\_row,d\_col=directions[0]

for j in range(1,n\*n+1): （遍历1到n\*n）

mx[row][col]=j

if mx[row+d\_row][col+d\_col]!=0:（碰到保护圈了）

N+=1

d\_row, d\_col = directions[N%4] （换方向）

row+=d\_row

col+=d\_col

for i in range(1,n+1):

print(' '.join(map(str,mx[i][1:-1])))（输出矩阵的方式）

（注：for row in square: 和 for value in row: 主要用于逐行逐元素地遍历二维矩阵的所有元素，是处理矩阵问题中最常见的遍历方式之一。For row in square; print(row)将矩阵的每一行单独作为一个列表输出；for row in square + for value in row + print(value, end=””) 输出该矩阵内的所有值）

动态规划：

双dp（土豪购物）——两种情况分析，取或不取，放或不放：

a=input()

merch=[int(items) for items in a.split(',') ]

n=len(merch)

dp1=[0]\*n

dp2=[0]\*n

dp1[0]=merch[0]

dp2[0]=merch[0]

for i in range(1,n):

dp1[i]=max(dp1[i-1]+merch[i],merch[i]) #不放回

dp2[i]=max(dp1[i-1],dp2[i-1]+merch[i],merch[i])

print(max(dp2))

拦截导弹：

def max\_intercepted\_missiles(k, heights):

dp = [1] \* k

for i in range(1, k):

for j in range(i):

if heights[i] <= heights[j]:

dp[i] = max(dp[i], dp[j] + 1)

return max(dp)

if \_\_name\_\_ == "\_\_main\_\_":

k = int(input())

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

result = max\_intercepted\_missiles(k, heights)

print(result)

0-1背包问题（二维dp）：（N是物品件数，B是背包最大承重）

N,B=map(int,input().split())

price=[0]+list(map(int,input().split()))

weight=[0]+list(map(int,input().split()))

dp=[[0]\*(B+1) for \_ in range(N+1)]

for i in range(N+1):

for j in range(B+1):

if weight[i]<=j:

dp[i][j]=max(dp[i-1][j],dp[i-1][j-weight[i]]+price[i])

else:

dp[i][j]=dp[i-1][j]

print(dp[-1][-1])

最长公共子序列：

while True:

try:

a, b = input().split()

except EOFError:

break

alen = len(a)

blen = len(b)

dp = [[0]\*(blen+1) for i in range(alen+1)]

for i in range(1, alen+1):

for j in range(1, blen+1):

if a[i-1]==b[j-1]:

dp[i][j] = dp[i-1][j-1] + 1

else:

dp[i][j] = max(dp[i-1][j], dp[i][j-1])

print(dp[alen][blen])

多重背包问题（一个物品可以拿多个）：（二进制优化版，虽然不是很懂）

def bounded\_knapsack\_binary(v, w, m, V):

# v: 物品体积列表； w: 物品价值列表； m: 每种物品最大数量； V: 背包容量

n = len(v)

items = []

# 二进制拆分

for i in range(n):

count = m[i]

k = 1

while k <= count:

items.append((k \* v[i], k \* w[i]))

count -= k

k \*= 2

if count > 0:

items.append((count \* v[i], count \* w[i]))

# 动态规划

dp = [0] \* (V + 1)

for volume, value in items:

for j in range(V, volume - 1, -1):

dp[j] = max(dp[j], dp[j - volume] + value)

return dp[V]

# 示例数据

v = [1, 2, 3] # 物品体积

w = [6, 10, 12] # 物品价值

m = [3, 2, 1] # 每种物品最大数量

V = 5 # 背包容量

print(bounded\_knapsack\_binary(v, w, m, V)) # 输出：30

非优化版：当做0-1背包做

def bounded\_knapsack\_dp(v, w, m, V):

n = len(v)

dp = [0] \* (V + 1)

for i in range(n):

if m[i] \* v[i] >= V:

# 当物品的总量大于背包容量时，直接按照完全背包处理

for j in range(v[i], V + 1):

dp[j] = max(dp[j], dp[j - v[i]] + w[i])

else:

# 否则按照多重背包处理

for j in range(V, 0, -1):

for k in range(1, min(m[i], j // v[i]) + 1):

dp[j] = max(dp[j], dp[j - k \* v[i]] + k \* w[i])

return dp[V]

完全背包问题：

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

dp = [-1] \* (n + 1)

dp[0] = 0 # 长度为 0 时可以切分 0 段

for i in range(1, n + 1):

for j in (a,b,c):

if j<=i and dp[i-j]!=-1:

dp[i]=max(dp[i],dp[i-j]+1)

print(dp[n])

贪心代码：（看一下以右边序列排序的怎么做）

区间问题：

按照右端点排序：重叠区间问题

（1）不相交区间数目最大

（2）区间选点问题（给出一堆区间，取尽量少的点，使得每个区间内至少有一个点）——尽量选择当前区间最右边的点，同不相交区间数目最大

(3)区间覆盖问题：给出一堆区间和一个目标区间，问最少选择多少区间可以覆盖掉题中给出的这段目标区间。

按照左端点排序：合并区间问题

（1）区间合并（左端点由小到大排序，维护前面区间右端点ed）

（2）区间分组问题：从前往后依次枚举每个区间，判断当前区间能否被放到某个现有组里面。

合并区间：给出一个区间的集合，请合并所有重叠的区间。

class Solution:

def merge(self, intervals):

result = []

if len(intervals) == 0:

return result # 区间集合为空直接返回

intervals.sort(key=lambda x: x[0]) # 按照区间的左边界进行排序

result.append(intervals[0]) # 第一个区间可以直接放入结果集中

for i in range(1, len(intervals)):

if result[-1][1] >= intervals[i][0]: # 发现重叠区间

#合并区间只需要更新结果集最后一个区间的右边界，因为根据排序，左边界已经是最小的

result[-1][1] = max(result[-1][1], intervals[i][1])

else:

result.append(intervals[i]) # 区间不重叠

return result

无重叠区间：给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

class Solution:

def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:

if not intervals:

return 0

intervals.sort(key=lambda x: x[0]) # 按照左边界升序排序

count = 0 # 记录重叠区间数量

for i in range(1, len(intervals)):

if intervals[i][0] < intervals[i - 1][1]: # 存在重叠区间

intervals[i][1] = min(intervals[i - 1][1], intervals[i][1]) # 更新重叠区间的右边界

count += 1

return count

充实的寒假生活：

n=int(input())

activities=[]

for \_ in range(n):

start,end=map(int,input().split())

activities.append((start,end))

activities.sort(key=lambda x:x[1])

ed=-1

num=0

for i in range(n):

if ed<activities[i][0]（即activities这个列表里第i个元素的start）:

num+=1

ed=activities[i][1]（即activities这个列表里第i个元素的end）

print(num)

划分字母区间：字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。

s=input()

last\_occurrence = {} # 存储每个字符最后出现的位置

for i, ch in enumerate(s):

last\_occurrence[ch] = i

print(last\_occurrence)

result = []

start = 0

end = 0

for i, ch in enumerate(s):

end = max(end, last\_occurrence[ch]) # 找到当前字符出现的最远位置

if i == end: # 如果当前位置是最远位置，表示可以分割出一个区间

result.append(end - start + 1)

start = i + 1

print(result)

应用Heapq的示例（毒药）

n = int(input()) # 输入药水数量

potion = list(map(int, input().split())) # 输入药水的效果值

health = 0 # 初始健康值

count = 0 # 能喝的药水数量

min\_heap = [] # 小根堆，用来存储已饮用的药水效果

for i in range(n):

health += potion[i] # 喝掉当前的药水

heapq.heappush(min\_heap, potion[i]) # 将药水效果加入堆中

count += 1 # 增加已喝的药水数量

# 如果健康值小于0，说明健康值不合法，需要丢掉一个药水

if health < 0:

health -= heapq.heappop(min\_heap) # 弹出最小的药水，回退健康值

count -= 1 # 减少已喝的药水数量

print(count)

Bisect：import bisect:用于在已排序的列表中找到插入点，或者对列表进行插入操作，同时保持列表的顺序。常见的应用包括二分查找（binary search）、在排序列表中插入元素等。

**a=bisect.bisect\_left(list, item, lo=0, hi=len(list))**：在已排序的 list 中查找插入点，使得插入 item 后仍保持列表的有序性。返回的插入点是从列表的左侧开始，第一个大于等于 item 的位置。

例：**arr = [1, 3, 4, 10, 12]；index = bisect.bisect\_left(arr, 5)；print(index)**

# 输出 3，因为 5 应该插入在 10 之前，索引 3

**bisect.bisect\_right(list, item, lo=0, hi=len(list)) 或 bisect.bisect()**：类似于 bisect\_left，但返回的插入点是从列表的右侧开始，第一个大于 item 的位置。

**bisect.insort\_left(list, item, lo=0, hi=len(list))**：将 item 插入到 list 中，使其保持有序性，插入点使用 bisect\_left 查找的结果。

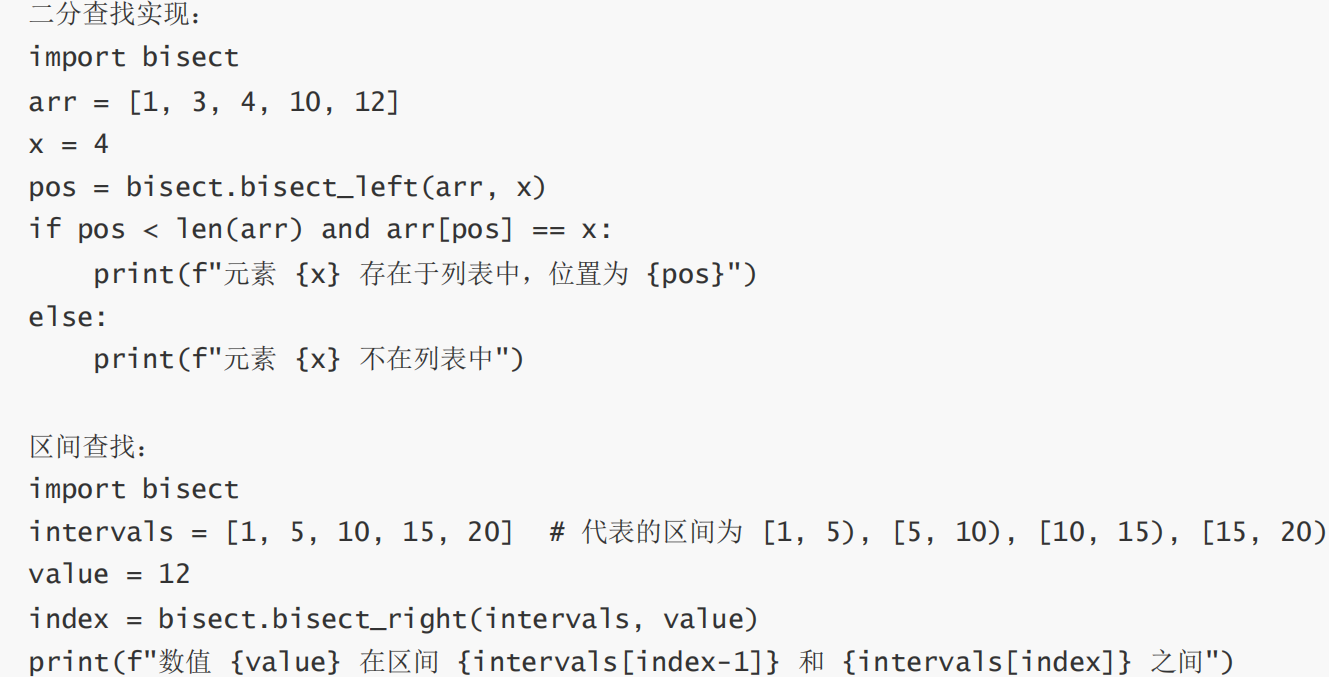
例：**arr = [1, 3, 4, 10, 12]；bisect.insort(arr, 5) # 将 5 插入到正确的位置；print(arr)**

# 输出 [1, 3, 4, 5, 10, 12]

**bisect.insort\_right(list, item, lo=0, hi=len(list))** 或 **bisect.insort()：**将 item 插入到 list 中，插入点使用 bisect\_right 查找的结果。

bisect.bisect\_left() 和 bisect.bisect\_right() 用于查找插入位置。

bisect.insort\_left() 和 bisect.insort\_right() 用于在保持列表有序的情况下插入元素。



递归：（找到base case）

汉诺塔问题：

def tower(n,a,b,c):

if n==1:

print(a+'->'+c)

return

if n>1:

tower(n-1,a,c,b)

print(a+'->'+c)

tower(n-1,b,a,c)

height=int(input())

print(2\*\*height-1)

tower(height,'A','B','C')

全排列

（1）使用 Python 内置的 itertools.permutations 方法

from itertools import permutations

k = int(input())

l = [i + 1 for i in range(k)]

# 生成全排列

def whole\_list(l):

return permutations(l)

# 打印每种排列

for perm in whole\_list(l):

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

（2）递归的做法：

def whole\_list(l):

# 递归终止条件：当列表只有一个元素时，返回该元素的单个排列

if len(l) == 1:

return [l]

# 保存所有排列

permutations = []

# 遍历列表中的每个元素

for i in range(len(l)):

# 当前选择的元素

current = l[i]

# 剩下的元素（去掉当前选中的元素）

remaining = l[:i] + l[i+1:]

# 对剩下的元素递归求全排列

for perm in whole\_list(remaining): （这就是递归调用自身吧）

# 把当前选中的元素加到每个排列的开头

permutations.append([current] + perm)

return permutations

# 输入和调用

k = int(input())

l = [i + 1 for i in range(k)]

# 打印每种排列

for perm in whole\_list(l):

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

排序：（最大整数类问题OJ27373：假设有n个正整数，要求从中选取几个组成一个位数不超过m的新正整数，现需要求出最大可能数值是多少。比如有4个数，993，923，3817，3807，组成新数不超过7位，那么新数最大值为9933817。）

n = int(input())

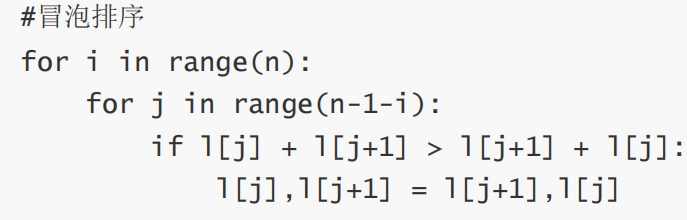
num=input.split()

for i in range(n-1):

for j in range(i+1,n):

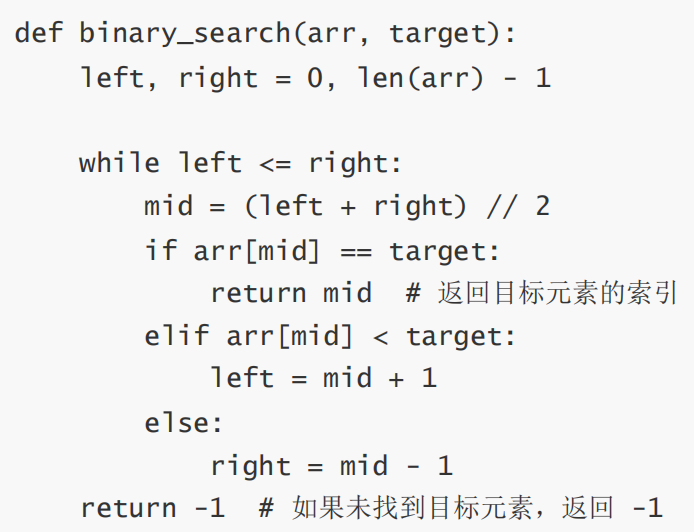
if nums[i]+nums[j]<nums[j]+nums[i]:

nums[i],nums[j]=nums[j],nums[i]（字典序的比较）



双指针

（河中跳房子）（对撞指针：回文串）（滑动窗口：最大子数组和、无重复字符的最长子字符串）



回文串：

def second(x):

left = 0

right = len(x) - 1

while left <= right:

if x[left] == x[right]:

left += 1

right -= 1

else:

return 'No'

return 'Yes'

移除重复的元素（原地修改）：给定一个有序数组，要求原地删除重复的元素，并返回新数组的长度。（其中一个慢指针 slow 用于记录已去重的数组的尾部，另一个快指针 fast 用于遍历整个数组。）

**def remove\_duplicates(arr):**

**if not arr:**

**return 0** 如果数组为空，直接返回 0，表示没有元素。

**slow = 0** 慢指针初始化为 0，表示去重后的新数组的尾部索引。

**for fast in range(1, len(arr)):** 从索引 1 开始遍历数组，fast 是快指针，用于扫描整个数组。

**if arr[fast] != arr[slow]:** 如果快指针指向的元素与慢指针指向的元素不同，说明这是一个新的、没有重复的元素。

**slow += 1**

**arr[slow] = arr[fast]** 慢指针移动到下一个位置，并将快指针当前的值赋给慢指针所在位置，表示找到一个新的非重复元素。

**return slow + 1** 慢指针指向的是最后一个非重复元素的索引，所以新数组的长度是 slow + 1。

滑动窗口

初始化：维护一个窗口 [start + 1, i] ，表示当前的无重复子串。使用一个字典 char\_index 来记录每个字符最近一次出现的位置。

扩展窗口：遍历字符串，逐个字符地扩展窗口的右边界 i 。

收缩窗口：如果当前字符 c 在字典中且其上次出现的位置在当前窗口内，则需要收缩窗口的左边界 start，使其不包含重复字符。

例：最长的无重复子序列串



Stack（后进先出）

1. 入栈 (Push): 使用 append() 方法将元素压入栈。

2. 出栈 (Pop): 使用 pop() 方法将栈顶元素弹出。

3. 栈顶元素 (Peek): 使用索引 [-1] 访问栈顶元素。

4. 检查栈是否为空: 使用 if not stack 来判断栈是否为空。

应用：快速堆猪

stack=[]

min\_stack=[] （一个主栈一个辅助栈）

def push(x):

stack.append(x)

if not min\_stack or x<= min\_stack[-1]:

min\_stack.append(x)

def pop():

if stack:

top=stack.pop()

if top==min\_stack[-1]:

min\_stack.pop()

def get\_min():

if min\_stack:

return min\_stack[-1]

return

while True:

try:

command=input().strip()

if command.startswith('push'):

value=int(command.split()[1])

push(value)

elif command.startswith('pop'):

pop()

elif command.startswith('min'):

if get\_min() is not None: （不能直接写if get\_min()因为如果返回值为0的话也会视为False）

print(get\_min())

except EOFError:

break

其它：

与平方数相关的题，平方数可以这么找：

squares=set()

i=1

while i\*i<=10\*\*9:

squares.add(i\*i)

i+=1

或者列表推导式？squares=[i\*\*2 for i in range(10)]

用dp判断某一个字符串是否均由平方数组成：

dp=[False]\*(n+1)

dp[0]=True

for i in range(1,n+1):

for j in range(i):

if int(digits[j:i]) in squares and dp[j]==True:

dp[i]=True

break

在一定范围内筛选该范围内所有质数的代码：

N = 10005

Is\_prime = [True] \* N

Is\_prime[0] = is\_prime[1] = False

p = 2

while p \* p <= N:

if is\_prime[p]:

for i in range(p \* 2, N, p):

Is\_prime[i] = False

p += 1

找平方素数（恰好有且仅有三个不同的正除数）只需判断该数的平方根是否为素数即可。

前缀和：flower

t,k=map(int,input().split())

MAXN=100001

dp=[0]\*MAXN

dp[0]=1

sum\_flower=[0]\*MAXN

MOD = 10\*\*9+7

for i in range(1,MAXN):

if i<k:

dp[i]=dp[i-1]

else:

dp[i]=(dp[i-1]+dp[i-k])% MOD

sum\_flower[i]=(sum\_flower[i-1]+dp[i])% MOD

for \_ in range(t):

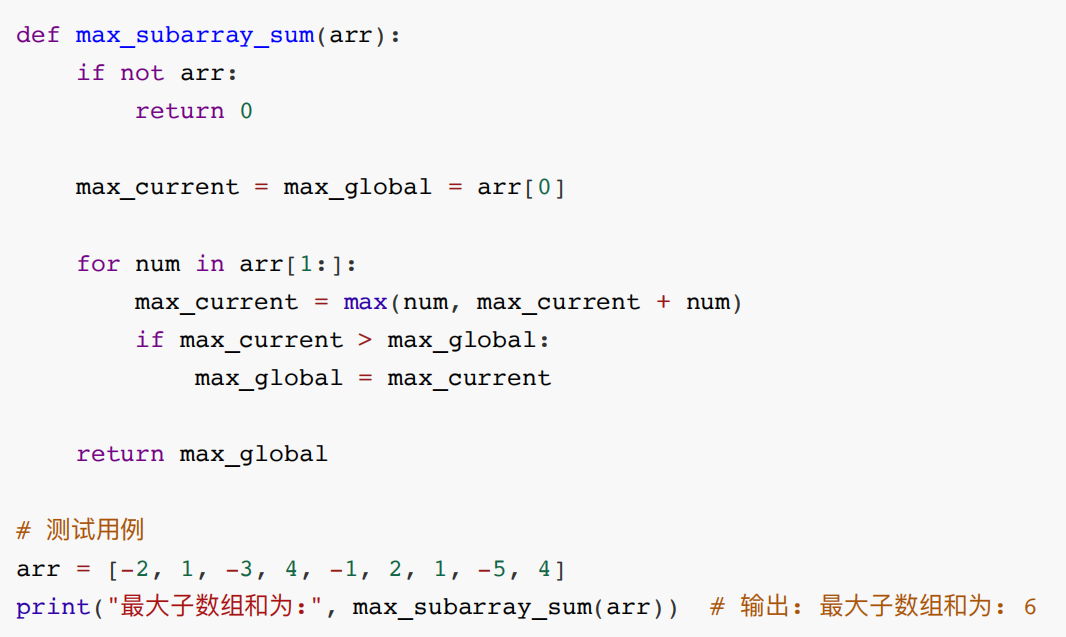
a, b = map(int, input().split())

print((sum\_flower[b]-sum\_flower[a-1]+MOD)% MOD)

求xx的最大值之前=都要先设置一个maxValue = float("-inf")，给最大值的初始值赋为无穷小，这样就可以通过max(a,b)得出最大值。

将每一个数字转换成单个字符组成的列表e.g.123→[’1’, ’2’, ’3’]：a = list(str(num))

Kadane's algorithm：Kadane's 算法是⼀种⽤于解决最⼤⼦数组问题（Maximum Subarray Problem）的⾼效算法，即在⼀个⼀维整数数组中找到具有最⼤和的连续⼦数组。



两种语言转换类问题：先创建两个字典（分别为字符：数字；数字：字符）；先从第一种语言转换成数字，然后再从数字转换成第二种语言。

english=input()

english\_to\_num\_map={'two':2, 'three':3, 'four':4, 'five':5, 'six':6, 'seven':7,'hundred':100, 'thousand':1000, 'million':1000000}

def english\_to\_num(s):

total = 0

current=0

words = s.split()

for word in words:

if word in english\_to\_num\_map:

value=english\_to\_num\_map[word]

if value==100:

current\*=value

elif value>=1000:

current\*=value

total+=current

current=0

else:

current+=value

total+=current

return total

if "negative" in english:

print("-"+str(english\_to\_num(english)))

else:

print(english\_to\_num(english))