**数据结构基本选择与应用：**

一、**列表**list

1，特点：便于查找

2，基本用法

插入：list.insert(1,‘a’) 指定1位置插入a

去重：set(list) 生成的是字典

删除：list.pop() 空值为最后一个，可以加索引，返回该元素的值

list.remove(‘a’) 移除元素a

del list[0:3] 切片式删除

排序：小到大list=sorted（lis,reverse=Ture） 大到小list=sorted(lis) **特殊l=sorted(l, key=lambda x:x[1])**

获取数据：list.index(’a‘) a索引获取

list[1:4] 切片获取

反向：list.reverse() 反向排列

计算出现次数：list.count(obj) 计算obj在列表出现次数

追加值（如合并两个列表）：list.extend(seq) 列表末尾追加值

打包成元组：zip（list1，list2）

输出：str.join(list)

二、元组

1，特点：与列表相似，但不能修改

三、**字典**

1.特点：有键值对。键必须唯一。key:必须是可哈希的值，比如intmstring,float,tuple,但是，list,set,dict不行。

2.基本用法

增加与修改： dict['key'] = value

删除：del dict[“key”]

查找与输出：键值对dict.items 键dict.keys 值dict.values （均以列表形式返回）

四、**集合**

1.特点：查找很快，一般打表法都需要用。元素不重复，可以筛除。

2.基本用法：

创建：set（）

添加元素：s.add(x) 如果已经存在，则不操作。

移除元素：s.remove（x） s.discard（x）后者不存在也不会报错

集合运算的内置函数：intersection：交集 difference：差集 union：并集 issubset：检查一个集合是否为另一个子集issuperset：检查一个集合是否为另一个超集

五、**字符串**

大小写：lower() 【str.lower() 】 #将字符串转换为小写字符串 uppper()#将字符串转换为大写字符串 capitalize()#将字符串首字母变为大写title()#将字符串中每个单词的首字母都变为大写 swapcase()#将字符串中的字符大小写互换

判断类型：isupper()#是否全为大写 islower()#是否全为小写 isdigit()#是否全为数字 isalnum()#是否全为字母或汉字或数字 isalpha()#是否全为字母或汉字

ASCII：ord() # 字符转ASCII chr() # ASCII转字符

1. `str.lstrip() / str.rstrip()`: 移除字符串左侧/右侧的空白字符。

2. `str.find(sub)`: 返回子字符串`sub`在字符串中首次出现的索引，如果未找到，则返回-1。

3. `str.replace(old, new)`: 将字符串中的`old`子字符串替换为`new`。

4. `str.startswith(prefix) / str.endswith(suffix)`: 检查字符串是否以`prefix`开头或以`suffix`结尾。

5. `str.isalpha() / str.isdigit() / str.isalnum()`: 检查字符串是否全部由字母/数字/字母和数字组成。

**六、数字的基本处理**：

保留位数：向上取整：math.ceil（x） 向下取整：math.floor(x) 指定小数位数输出（包含四舍五入，会补齐）："{:.2f}".format(num)

单纯四舍五入：rounded\_number = round(number, ndigits)，其中 number 是要进行四舍五入的数字，ndigits 是要保留的小数位数。如果 ndigits 省略或是 0，则 round() 函数将返回最接近的整数。如果 ndigits 是负数，则四舍五入将应用于十 进制点左侧的数字。

运算法：import math

开根号：math.sqrt(x) 绝对值：abs（num）math.factorial(x) 返回x的阶乘，即x!，要求x是非负整数

math.gcd(a, b) 返回a和b的最大公约数，要求a和b都是整数。

math.sin(x) 返回x弧度的正弦值 math.log(x, [base]) 返回x的对数，如果指定了base，则以base为底，否则以e为底

math.fabs(x) 返回x的绝对值

进制：# 二进制转十进制

binary\_str = "1010"

decimal\_num = int(binary\_str, 2) # 第一个参数是字符串类型的某进制数，第二个参数是他的进制，最终转化为整数

print(decimal\_num) # 输出 10

#取二进制 bin(num)

**七、双向队列deque**

from collections import deque

d=deque()#可用len(deque),和list,str一样可以用iterable.count(x)计数x出现次数复杂度O(n)

d.pop()'右弹出' d.popleft()'左弹出'

d.append()'右添加' d.appendleft()'左添加'

d.extend()'右扩展' d.extendleft()'左扩展'

d.rotate()#将双向队列向右旋转 n 步。n 为正，右侧的元素会移动到左侧；n 为负,相反。

**其它重要的基本操作：**

**Iru\_cache：**需要注意的是，使用@lru\_cache装饰器时，应注意以下几点：

被缓存的函数的参数必须是可哈希的，这意味着参数中不能包含可变数据类型，如列表或字典。

缓存的大小会影响性能，需要根据实际情况来确定合适的大小或者使用默认值。

由于缓存中存储了计算结果，可能导致内存占用过大，需谨慎使用。

可以是多参数的。

from functools import lru\_cache

@lru\_cache(maxsize=128) #tle了加大，mle了减小

def func():

**排序 二分查找输出排位：**

import bisect

sorted\_list = [1,3,5,7,9] #[(0)1, (1)3, (2)5, (3)7, (4)9]

position = bisect.bisect\_left(sorted\_list, 6) **#查找元素应左插入的位置**

print(position) # 输出：3，因为6应该插入到位置3，才能保持列表的升序顺序

bisect.insort\_left(sorted\_list, 6) **#左插入元素**

print(sorted\_list) # 输出：[1, 3, 5, 6, 7, 9]，6被插入到适当的位置以保持升序顺序

sorted\_list=(1,3,5,7,7,7,9)

print(bisect.bisect\_left(sorted\_list,7))

print(bisect.bisect\_right(sorted\_list,7))

# 输出：3 6

**Calendar库**

1. `calendar.month(年, 月)`: 返回一个月份的日历字符串。它接受年份和月份作为参数，并以多行字符串的形式返回该月份的日历。

2. `calendar.calendar(年)`: 返回一个年份的日历字符串。这个函数生成整个年份的日历，格式化为多行字符串。

3. `calendar.monthrange(年, 月)`: 返回两个整数，第一个是该月第一天是周几（0-6表示周一到周日），第二个是该月的天数。

4. `**calendar.weekday**(年, 月, 日)`: 返回给定日期是星期几。0-6的返回值分别代表星期一到星期日。

5. `**calendar.isleap**(年)`: 返回一个布尔值，指示指定的年份是否是闰年。

6. `**calendar.leapdays**(年1, 年2)`: 返回在指定范围内的闰年数量，不包括第二个年份。

7. `calendar.monthcalendar(年, 月)`: 返回一个整数矩阵，表示指定月份的日历。每个子列表表示一个星期；天数为0表示该月份此天不在该星期内。

8. `calendar.setfirstweekday(星期)`: 设置日历每周的起始日。默认情况下，第一天是星期一，但可以通过这个函数更改。

9. `calendar.firstweekday()`: 返回当前设置的每周起始日。

**质数的欧拉筛**：（快但储存大）

def Euler\_sieve(n):

primes = [True for \_ in range(n+1)]

p = 2

while p\*p <= n:

if primes[p]:

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

primes[i] = False

print(primes)

p += 1

return primes[-1]

print(Euler\_sieve(1000)) # O(n)

**埃氏筛（慢但储存小）：**def SieveOfEratosthenes(n, prime):

p = 2

while (p \* p <= n): # If prime[p] is not changed, then it is a prime

if (prime[p] == True): # Update all multiples of p

for i in range(p \* 2, n+1, p):

prime[i] = False

p += 1

**矩阵要用深拷贝**from copy import deepcopy

matrix1=deepcopy（matrix）

**permutations：全排列**

from itertools import permutations as per

elements = [1, 2, 3]

permutations = list(per(elements))

**combinations：组合**

from itertools import combinations as com

elements = ['A', 'B', 'C', 'D']

# 生成所有长度为2的组合

combinations = list(com(elements, 2))

**reduce：累积**

import functools

numbers = [1, 2, 3, 4, 5]

# 使用 reduce 计算累积乘积

product = functools.reduce(lambda x, y: x \* y, numbers)

**product：笛卡尔积**

from itertools import product

# 创建两个可迭代对象

colors = ['red', 'blue']

numbers = [1, 2]

# 生成它们的笛卡尔积

cartesian\_product = list(product(colors, numbers))

# 创建一个可迭代对象

colors = ['red', 'blue']

# 生成它们的重复笛卡尔积（即三个位置上放元素，且可重复放）

repeat\_cartesian\_product = list(product(colors, repeat=3))

**Python字符串格式化输出:**



**在多组数据的多重问题时，注意一些变量的设定应该放在for循环内**

**函数的参数不能被设为全局变量**

**RE可能是爆栈，解决方法**：import sys

sys.setrecursionlimit(1<<30) （拓栈）

**从大到小排序**：sorted(original\_list, reverse=True)

**遍历的语法**：range(start, stop[, step]) 如果使用负数作为步长，则开始值必须大于结束值

**数字分解dfs**：

arr = [0] \* 100 # arr 用于记录方案

n, m = map(int, input().split()） #n个数，分成m个

def dfs(n, i, a):

if n == 0:

if len(arr[1:i])==m:

print(arr[1:i])

if i <= m:

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

arr[i] = j

dfs(n - j, i + 1, j) # 请仔细思考该行含义。

dfs(n, 1, 1)

**难题区模版：**

**DFS/BFS 图**

1. 八皇后
2. 滑雪（最长下滑路径）
3. 小游戏（简单迷宫寻路，但可跃迁到域外）
4. 城堡问题（给墙，寻找联通分量的数量和最大面积）
5. 变换的迷宫（墙在某些时刻可以穿过）
6. a knight‘s journey（象棋骑士巡逻遍历）
7. 海贼王之伟大航路（遍历的最短路程）
8. Roads（有限制的加权图的最短路径问题）
9. 迷宫问题 （基础走迷宫，但要输出路径）

**GREEDY**

**装箱问题：**乌鸦坐飞机：从大往小塞，尽量塞满。比较简单的贪心

OJ25566 CPU调度：容易猜到贪心策略是把w[i]长的尽量往前放，这样“不怎么挤占后面的空间”，但证明并不容易。对于这个题来说，**直觉**很重要。

OJ04137 最小新整数：转换思路，考虑留下l=n-k位。注意到留下的数要尽可能小，首先是首位要尽可能小（**分析最优解性质**），于是留下的第一个数一定是前k+1位中最小的（因为前k+1位中至少要留下一位），设为第t位。如果t=k+1则已经做完，否则前t位已经不用考虑（删t-1位，留1位），接下来要删k-t+1位，那么留下的第一个数一定是剩下前k-t+2位中最小的。依此类推。（这里其实有一点递归的思想，决定第一个要删掉的数之后化归为一个更小的同构问题）

OJ12559 最大最小整数：这里的麻烦在于有一个整数是另一个的前段的情形，不容易判断。需要想到把每个整数自循环足够多次后再排序。另一种我觉得更自然的想法是考虑何时最优，最优的时候任两个相邻数互换一定会变差（**局部调整**），这样可以得到任两个数之间的序关系；继而证明这是一个全序关系，依此排序。

OJ16067 电影节，OJ01328 Radar Installation，OJ27104世界杯只因，OJ04144畜栏保留问题：

这几个题是一个系列的，都属于**\*\*区间问题\*\***，基本涵盖了区间问题的所有常见类型。

这类区间问题普遍的办法是将区间排序（注意按左端点和右端点排序是有区别的！通常在**从左往右扫的时候按右端点排序**），然后按顺序考虑每一个区间，每次将**第一个满足要求**的区间加进来，同时**更新**剩下区间所需满足的要求（通常是区间右端点），完成一轮操作。

**DP**

**背包问题：**

#0-1背包的memory简化

f[i][l]=max(f[i-1][l],f[i-1][l-w[i]]+v[i])#这要二维数组i为进行到第i个物品，l为最大容量

for i in range(1, n + 1):#这时只需要一维，l为最大容量，通过反复更新维护

for l in range(W, w[i] - 1, -1):#必须这样逆序，要让每个f只被更新一次

f[l] = max(f[l], f[l - w[i]] + v[i])

#完全背包（每件物品可以选择任意次）

f[i][l]=max(f[i-1][l],f[i][l-w[i]]+v[i])#这要二维数组i为进行到第i个物品，l为最大容量

for i in range(1, n + 1):#这时只需要一维，l为最大容量，通过反复更新维护

for l in range(0, W - w[i] + 1):#此时要正序，根本原因是可以多次选择

f[l + w[i]] = max(f[l] + v[i], f[l + w[i]])

#多重背包（物品选择指定次）

#朴素想法转化为0-1背包，可能超时，因此考察二进制拆分（先尽力拆为1，2，4，8...)

import math

k=int(math.log(x,2))

for i in range(k+2):

if x>=2\*\*i:

x-=2\*\*i

coi.append(y\*(2\*\*i))

else:

coi.append(x\*y)

break

**Matrix**

一，Input

1、加保护层：（***矩阵运算****2*）

l = []

l.append([-1 for x in range(m+2)])

for \_ in range(n):

l.append([-1] + [int(\_) for \_ in input().split()] + [-1])

l.append([-1 for x in range(m+2)])

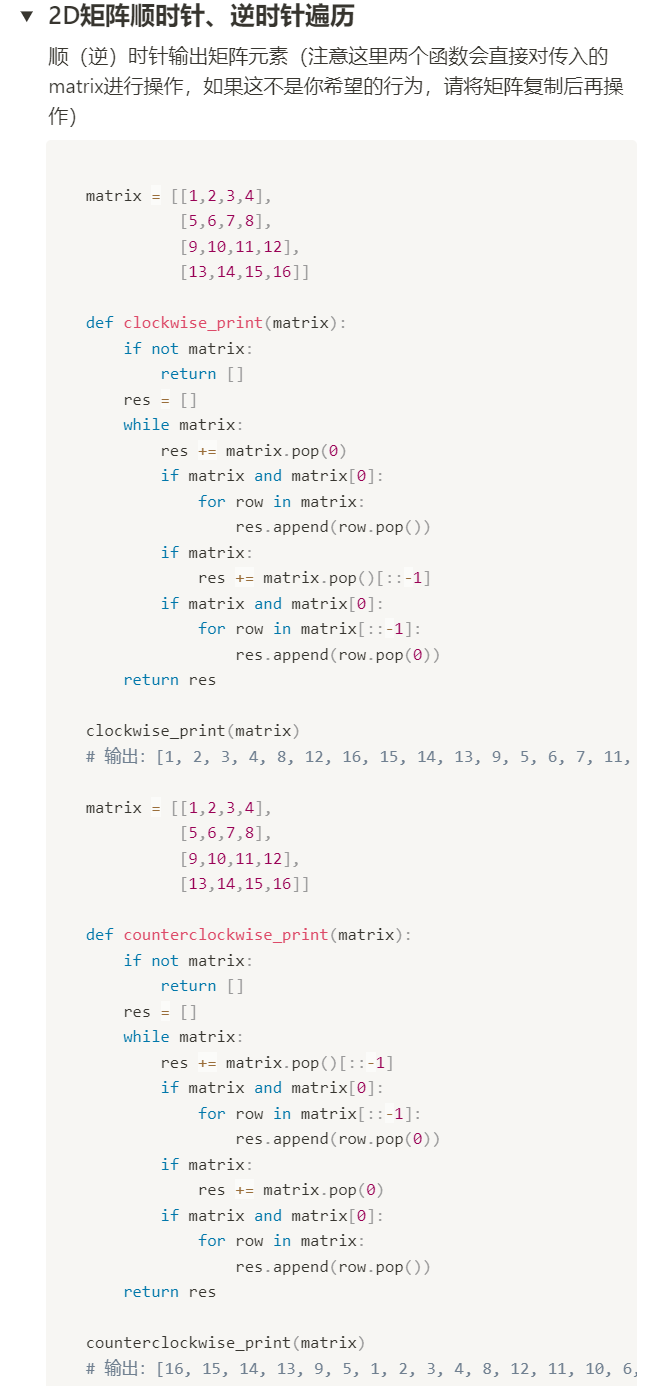
二，基本遍历操作（如下，顺时针逆时针）

三，输出整个矩阵：

for \_ in matrix:

print(" ".join(map(str, \_)))

四，



**矩阵运算(matrices)**

**图**

**DFS**：基本模版：（***DFS***）

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

maze = []

maze.append( [-1 for x in range(m+2)] ) #加保护圈操作

for \_ in range(n):

maze.append([-1] + [int(\_) for \_ in input().split()] + [-1])

maze.append( [-1 for x in range(m+2)] )

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

dy = [ 0, 1, 0, -1] #创建可能的移动

def dfs(maze, x, y):

global cnt #可能的变量，如路径数，路径步数

for i in range(4): #四向搜索

nx = x + dx[i]

ny = y + dy[i]

if maze[nx][ny] == 'e': #终止目标条件

cnt += 1

continue

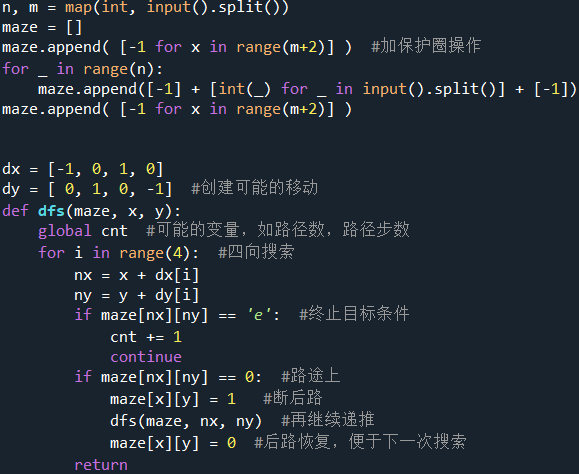
if maze[nx][ny] == 0: #路途上

maze[x][y] = 1 #断后路

dfs(maze, nx, ny) #再继续递推

maze[x][y] = 0 #后路恢复，便于下一次搜索

return



**BFS:**

① 定义队列 q，并将起点s⼊队。 ② 写⼀个 while 循环，循环条件是队列q⾮空。 ③ 在 while 循环中，先取出队⾸元素 top，然后访问它（访问可以是任何事情，例如将其输出）。访问完后将其出 队。 ④ 将top 的下⼀层结点中所有未曾⼊队的结点⼊队，并标记它们的层号为 now 的层号加1，同时设置这些⼊队的结 点已⼊过队。

⑤ 返回 ② 继续循环。

 再强调⼀点,在BFS 中设置的 inq 数组的含义是判断结点是否已⼊过队，⽽不是结点是否已被访问。区别在于:如果 设置成是否已被访问，有可能在某个结点正在队列中(但还未访问)时由于其他结点可以到达它⽽将这个结点再次⼊ 队，导致很多结点反复⼊队，计算量⼤⼤增加。因此BFS 中让每个结点只⼊队⼀次，故需要设置 inq 数组的含义为 结点是否已⼊过队⽽⾮结点是否已被访问。

基本模版：

from collections import deque

# Constants

MAXN = 100 #构建inqueue的参数，一般也可用mn代替

MAXD = 4 #可走方向

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

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

# Functions

def can\_visit(x, y):

return 0 <= x < n and 0 <= y < m and matrix[x][y] == 1 and not in\_queue[x][y]

def bfs(x, y):

q = deque([(x, y)])

in\_queue[x][y] = True

while q:#这里是基本情况，路1墙0，如果不是，要加判断

front = q.popleft()#出队

#本题属于区块探索，如果是走迷宫，则需要加if判断是否到达目标并返回

for i in range(MAXD):#四种尝试

next\_x = front[0] + dx[i] #走

next\_y = front[1] + dy[i]

if can\_visit(next\_x, next\_y):#同样加判断

in\_queue[next\_x][next\_y] = True #过河拆桥

q.append((next\_x, next\_y)) #下一波入队

#如果是迷宫，一般还要加变量的改变，如步数

# Input

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

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

in\_queue = [[False] \* MAXN for \_ in range(MAXN)]

# Main process

counter = 0

for i in range(n):

for j in range(m):

if matrix[i][j] == 1 and not in\_queue[i][j]:

bfs(i, j)

counter += 1

# Output

print(counter)

### 1.1 迷宫可行路径数

https://sunnywhy.com/sfbj/8/1/313

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（不允许移动到曾经经过的位置），且只能移动到平地上。求从迷宫左上角到右下角的所有可行路径的条数。

#### 加保护圈，原地修改

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

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

def dfs(maze, x, y):

global cnt

for i in range(4):

nx = x + dx[i]

ny = y + dy[i]

if maze[nx][ny] == 'e':

cnt += 1

continue

if maze[nx][ny] == 0:

maze[x][y] = 1

dfs(maze, nx, ny)

maze[x][y] = 0

return

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

maze = []

maze.append( [-1 for x in range(m+2)] )

for \_ in range(n):

maze.append([-1] + [int(\_) for \_ in input().split()] + [-1])

maze.append( [-1 for x in range(m+2)] )

maze[1][1] = 's'

maze[n][m] = 'e'

cnt = 0

dfs(maze, 1, 1)

print(cnt)

### 1.2 指定步数的迷宫问题

### 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（不允许移动到曾经经过的位置），且只能移动到平地上。现从迷宫左上角出发，问能否在恰好第步时到达右下角。

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

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

canReach = False

def dfs(maze, x, y, step):

global canReach

if canReach:

return

for i in range(4):

nx = x + dx[i]

ny = y + dy[i]

if maze[nx][ny] == 'e':

if step==k-1:

canReach = True

return

continue

if maze[nx][ny] == 0:

if step < k:

maze[x][y] = -1

dfs(maze, nx, ny, step+1)

maze[x][y] = 0

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

maze = []

maze.append( [-1 for x in range(m+2)] )

for \_ in range(n):

maze.append([-1] + [int(\_) for \_ in input().split()] + [-1])

maze.append( [-1 for x in range(m+2)] )

maze[1][1] = 's'

maze[n][m] = 'e'

dfs(maze, 1, 1, 0)

print("Yes" if canReach else "No")

### 1.3 矩阵最大权值

现有一个 n\*m 大小的矩阵，矩阵中的每个元素表示该位置的权值。现需要从矩阵左上角出发到达右下角，每次移动只能向上下左右移动一格（不允许移动到曾经经过的位置）。求最后到达右下角时路径上所有位置的权值之和的最大值。

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

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

maxValue = float("-inf")

def dfs(maze, x, y, nowValue):

global maxValue

if x==n and y==m:

if nowValue > maxValue:

maxValue = nowValue

return

for i in range(4):

nx = x + dx[i]

ny = y + dy[i]

if maze[nx][ny] != -9999:

tmp = maze[x][y]

maze[x][y] = -9999

nextValue = nowValue + maze[nx][ny]

dfs(maze, nx, ny, nextValue)

maze[x][y] = tmp

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

maze = []

maze.append( [-9999 for x in range(m+2)] )

for \_ in range(n):

maze.append([-9999] + [int(\_) for \_ in input().split()] + [-9999])

maze.append( [-9999 for x in range(m+2)] )

dfs(maze, 1, 1, maze[1][1])

print(maxValue)

### 1.4 矩阵最大权值路径

现有一个 n\*m 大小的矩阵，矩阵中的每个元素表示该位置的权值。现需要从矩阵左上角出发到达右下角，每次移动只能向上下左右移动一格（不允许移动到曾经经过的位置）。假设左上角坐标是(1,1)，行数增加的方向为增长的方向，列数增加的方向为增长的方向。求最后到达右下角时路径上所有位置的权值之和最大的路径。

MAXN = 5

INF = float('inf')

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

maze = []

for \_ in range(n):

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

maze.append(row)

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

maxValue = -INF

tempPath, optPath = [], []

MAXD = 4

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

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

def is\_valid(x, y):

return 0 <= x < n and 0 <= y < m and not visited[x][y]

def DFS(x, y, nowValue):

global maxValue, tempPath, optPath

if x == n - 1 and y == m - 1:

if nowValue > maxValue:

maxValue = nowValue

optPath = list(tempPath)

return

visited[x][y] = True

for i in range(MAXD):

nextX = x + dx[i]

nextY = y + dy[i]

if is\_valid(nextX, nextY):

nextValue = nowValue + maze[nextX][nextY]

tempPath.append((nextX, nextY))

DFS(nextX, nextY, nextValue)

tempPath.pop()

visited[x][y] = False

tempPath.append((0, 0))

DFS(0, 0, maze[0][0])

for pos in optPath:

print(pos[0] + 1, pos[1] + 1)

1.5 迷宫最大权值 现有一个大小的迷宫，其中1表示不可通过的墙壁，0表示平地。现需要从迷宫左上角出发到达右下角，每次移动只能向上下左右移动一格（不允许移动到曾经经过的位置），且只能移动到平地上。假设迷宫中每个位置都有权值，求最后到达右下角时路径上所有位置的权值之和的最大值。

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

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

maxValue = float("-inf")

def dfs(maze, x, y, nowValue):

global maxValue

if x==n and y==m:

if nowValue > maxValue:

maxValue = nowValue

return

for i in range(4):

nx = x + dx[i]

ny = y + dy[i]

if maze[nx][ny] == 0:

maze[nx][ny] = -1

tmp = w[x][y]

w[x][y] = -9999

nextValue = nowValue + w[nx][ny]

dfs(maze, nx, ny, nextValue)

maze[nx][ny] = 0

w[x][y] = tmp

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

maze = []

maze.append( [-1 for x in range(m+2)] )

for \_ in range(n):

maze.append([-1] + [int(\_) for \_ in input().split()] + [-1])

maze.append( [-1 for x in range(m+2)] )

w = []

w.append( [-9999 for x in range(m+2)] )

for \_ in range(n):

w.append([-9999] + [int(\_) for \_ in input().split()] + [-9999])

w.append( [-9999 for x in range(m+2)] )

dfs(maze, 1, 1, w[1][1])

print(maxValue)

2.1 数字操作（一维BFS）

从整数1开始，每轮操作可以选择将上轮结果加1或乘2。问至少需要多少轮操作才能达到指定整数。

from collections import deque

def bfs(n):

vis = set()

vis.add(1)

q = deque()

q.append((1, 0))

while q:

front, step = q.popleft()

if front == n:

return step

if front \* 2 <= n and front \* 2 not in vis:

vis.add(front \*2)

q.append((front \* 2, step+1))

if front + 1 <= n and front + 1 not in vis:

vis.add(front + 1)

q.append((front + 1, step+1))

n = int(input())

print(bfs(n))

### 2.2 矩阵中的块

现有一个 n\*m 的矩阵，矩阵中的元素为`0`或`1`。然后进行如下定义：

1. 位置(x,y)与其上下左右四个位置 $(x,y + 1)、(x,y - 1)、(x + 1,y)、(x-1,y)$ 是相邻的；

2. 如果位置 (x1,y1) 与位置 (x2,y2) 相邻，且位置 (x2,y2) 与位置 (x3,y3) 相邻，那么称位置(x1,y1)与位置(x3,y3)也相邻；

3. 称个数尽可能多的相邻的`1`构成一个“块”。

求给定的矩阵中“块”的个数。

from collections import deque

# Constants

MAXD = 4

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

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

def bfs(x, y):

q = deque([(x, y)])

inq\_set.add((x,y))

while q:

front = q.popleft()

for i in range(MAXD):

next\_x = front[0] + dx[i]

next\_y = front[1] + dy[i]

if matrix[next\_x][next\_y] == 1 and (next\_x,next\_y) not in inq\_set:

inq\_set.add((next\_x, next\_y))

q.append((next\_x, next\_y))

# Input

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

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

inq\_set = set()

# Main process

counter = 0

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

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

if matrix[i][j] == 1 and (i,j) not in inq\_set:

bfs(i, j)

counter += 1

# Output

print(counter)

### 2.3 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。求从迷宫左上角到右下角的最小步数。

from collections import deque

# 声明方向变化的数组，代表上下左右移动

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

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

def bfs(x, y):

q = deque()

q.append((x, y))

inq\_set.add((x, y))

step = 0

while q:

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

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

if cur\_x == n and cur\_y == m:

return step

for direction in range(4):

next\_x = cur\_x + dx[direction]

next\_y = cur\_y + dy[direction]

if maze[next\_x][next\_y] == 0 and (next\_x,next\_y) not in inq\_set:

inq\_set.add((next\_x, next\_y))

q.append((next\_x, next\_y))

step += 1

return -1

if \_\_name\_\_ == '\_\_main\_\_':

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

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

inq\_set = set()

step = bfs(1, 1)

print(step)

2.4 迷宫最短路径

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。假设左上角坐标是(1,1)，行数增加的方向为增长的方向，列数增加的方向为增长的方向，求从迷宫左上角到右下角的最少步数的路径。

from queue import Queue

MAXN = 100

MAXD = 4

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

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

def canVisit(x, y):

return x >= 0 and x < n and y >= 0 and y < m and maze[x][y] == 0 and not inQueue[x][y]

def BFS(x, y):

q = Queue()

q.put((x, y))

inQueue[x][y] = True

while not q.empty():

front = q.get()

if front[0] == n - 1 and front[1] == m - 1:

return

for i in range(MAXD):

nextX = front[0] + dx[i]

nextY = front[1] + dy[i]

if canVisit(nextX, nextY):

pre[nextX][nextY] = (front[0], front[1])

inQueue[nextX][nextY] = True

q.put((nextX, nextY))

def printPath(p):

prePosition = pre[p[0]][p[1]]

if prePosition == (-1, -1):

print(p[0] + 1, p[1] + 1)

return

printPath(prePosition)

print(p[0] + 1, p[1] + 1)

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

maze = []

for \_ in range(n):

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

maze.append(row)

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

pre = [[(-1, -1)] \* m for \_ in range(n)]

BFS(0, 0)

printPath((n - 1, m - 1))