语法：

1.向上取整：import math（math.ceil()） 向下取整：//

2.判断数据类型：isupper()、islower()、isdigit()（判断数字）、isalpha()（判断字母）、isalnum()（判断是否为字母或数字）、isspace()（判断是否为空格）。

3.判断素数：

def is\_prime(n):（判断一个数很便捷）

if n <= 1: # 素数必须大于 1

return False

for i in range(2, int(n \*\* 0.5) + 1): # 只需要判断到平方根

if n % i == 0: # 如果能被整除，说明不是素数

return False

return True # 如果没有能整除的数，则是素数

若要找出一个区间内的所有素数，则用埃拉托斯特尼筛法：

def sieve\_of\_eratosthenes(limit):

sieve = [True] \* (limit + 1)

sieve[0] = sieve[1] = False # 0 和 1 不是素数

for i in range(2, int(limit \*\* 0.5) + 1):

if sieve[i]:

for j in range(i \* i, limit + 1, i):

sieve[j] = False

return [i for i, is\_prime in enumerate(sieve) if is\_prime]

print(sieve\_of\_eratosthenes(100)) # 获取 100 以内的所有素数

4.判断完全平方数：

import math（返回的是布尔值）

def is\_perfect\_square(n):

if n < 0:

return False # 负数不是完全平方数

sqrt\_n = int(math.sqrt(n)) # 计算平方根并取整

return sqrt\_n \* sqrt\_n == n # 如果平方根的平方等于原数，则是完全平方数

5.保留几位小数：f”{x:.yf}”/"{:.2f}".format(number)/ round(number, 保留的位数)

6.sorted(目标，key,reverse=True(反向排序))，常用：sorted(a,key=lambda x:x[0],reverse=True)

7.remove(具体元素)：删除指定值的第一个匹配项/pop(索引):删除指定位置的元素并返回/del:删除指定索引的元素/列表推导式：numbers = [x for x in numbers if x != 3]

8.字典删除元素：my\_dict = {'a': 1, 'b': 2, 'c': 3}

del my\_dict['b'] # 删除键 'b' 对应的元素/removed\_value = my\_dict.pop('b') # 删除键 'b'，并返回其值/my\_dict = {key: value for key, value in my\_dict.items() if value >= 3}

9.字符串删除元素：replace(原元素，替换的新的元素，替换的次数)

10.一次性删除多个元素：若知道索引：for index in sorted(索引列表，reverse=True)

11.同时获取索引和值：for index, value in enumerate(fruits)

12.Counter的运用：首先要先引入：from collections import Counter

**统计频次**：s = "hello world"

counter = Counter(s)

print(counter)

# 输出: Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})

**获取出现次数最多的元素**：

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

counter = Counter(numbers)

# 获取出现次数最多的一个元素

most\_common = counter.most\_common(1)

print(most\_common) # 输出: [(3, 3)]，3 出现了 3 次

# 获取出现次数最多的两个元素

most\_common\_2 = counter.most\_common(2)

print(most\_common\_2) # 输出: [(3, 3), (2, 2)]，3 出现了 3 次，2 出现了 2 次

13.进制转换：int（要转换的字符串型，该字符串的原本进制）（可转化为十进制数，转化后为整型）

bin(number)[2:]（去掉 '0b' 前缀）（二）/oct(number)[2:]（去掉 '0o' 前缀）（八）/hex(number)[2:]（去掉 '0x' 前缀）（十六）

14.得到一个数所有的质因数：（返回的是一个列表，如：输入60，返回[2,2,3,5]）

def prime\_factors(n):

factors = []

while n % 2 == 0: # 处理2这个特殊情况

factors.append(2)

n //= 2

divisor = 3 # 处理奇数质因数，从3开始

while divisor \* divisor <= n: # 只需要检查到 sqrt(n)

while n % divisor == 0:

factors.append(divisor)

n //= divisor

divisor += 2 # 只检查奇数

# 如果剩下的是一个大于2的质数

if n > 2:

factors.append(n)

return factors

15.ASCII表转换：使用 ord() 将字符转换为 ASCII 码值/使用 chr() 将 ASCII 码值转换为字符/可以通过列表推导式或者 map() 函数将字符串中的字符批量转换为 ASCII 码值，或将 ASCII 码值批量转换为字符。

16.去除一个矩阵外围一圈的方法：[row[1:-1] for row in matrix[1:-1]]

17.设置递归深度限制(dfsRE时可以用)：import sys(先引入)sys.setrecursionlimit(1<< 30)（<<指的是二进制数向左移动30位）

18. functools.lru\_cache（dfs超时的时候可用）是一个装饰器,用来缓存函数的返回结果，以优化递归和重复计算的性能。

import functools(首先引入)@functools.lru\_cache(maxsize=128) # 缓存最多 128 个结果

19.一次性读取输入：

①import sys

# 读取所有输入并按行分割

input\_lines = sys.stdin.read().splitlines()

for line in input\_lines:

print(line)

②# 读取多行输入，直到 EOF

inputs = []

while True:

try:

line = input()

inputs.append(line)

except EOFError:

break

# 打印所有输入

print(inputs)

20.join和end的用法：result = " ".join(map(str, numbers)) # 使用 map 将数字转换为字符串

end：指定打印输出后的结尾字符。默认为 \n，即换行。如果你希望输出后不换行，可以设置为 "" 或其他字符串。

21.bisect（用于**二分查找**和**插入操作**。这个模块提供了一些有用的函数，可以帮助你在已排序的列表中快速找到元素的位置，或者将新元素插入到合适的位置而保持列表的顺序。）（以下的使用都是bisect.xxx）

**bisect\_left(a, x, lo=0, hi=len(a))**：

查找一个元素 x 在已排序的列表 a 中插入的位置，使得列表在插入 x 后仍然保持排序。返回的位置是 **第一个大于等于 x** 的位置。lo 和 hi 用于指定搜索的范围，默认为整个列表。

**bisect\_right(a, x, lo=0, hi=len(a))**：

查找一个元素 x 在已排序的列表 a 中插入的位置，使得列表在插入 x 后仍然保持排序。返回的位置是 **第一个大于 x** 的位置。lo 和 hi 用于指定搜索的范围，默认为整个列表。

**insort\_left(a, x, lo=0, hi=len(a))**：

将元素 x 插入到已排序的列表 a 中，保证列表仍然保持排序。insort\_left 会将 x 插入到 **第一个大于等于 x** 的位置。lo 和 hi 用于指定插入的范围，默认为整个列表。

**insort\_right(a, x, lo=0, hi=len(a))**：

将元素 x 插入到已排序的列表 a 中，保证列表仍然保持排序。insort\_right 会将 x 插入到 **第一个大于 x** 的位置。lo 和 hi 用于指定插入的范围，默认为整个列表。

22.全排列：**使用 itertools.permutations函数**

Python 的 itertools 模块提供了一个非常方便的 permutations 函数，可以轻松地生成全排列。（）代码实现如下）

import itertools

def generate\_permutations(arr):

return list(itertools.permutations(arr))

# 示例

arr = [1, 2, 3]

permutations = generate\_permutations(arr)

print(permutations)（返回的是一个列表，其中的每个元素便是一种排列结果）

23.将元素插入到一个列表中：

insert（索引（要插入的位置），插入的元素）

列表可以拼接

算法：

1. 查找算法：

1.二分查找：

def binary\_search(arr, target):

low, high = 0, len(arr) - 1

while low <= high:

mid = (low + high) // 2 # 计算中间索引

if arr[mid] == target:

return mid # 找到目标元素，返回索引

elif arr[mid] < target:

low = mid + 1 # 目标元素在右半边

else:

high = mid - 1 # 目标元素在左半边

return -1 # 如果没有找到目标元素，返回 -1

二分查找的递归实现：

def binary\_search\_recursive(arr, target, low, high):

if low > high:

return -1 # 如果查找区间为空，返回 -1

mid = (low + high) // 2 # 计算中间索引

if arr[mid] == target:

return mid # 找到目标元素，返回索引

elif arr[mid] < target:

return binary\_search\_recursive(arr, target, mid + 1, high) # 目标在右半边

else:

return binary\_search\_recursive(arr, target, low, mid - 1) # 目标在左半边

2.哈希表（字典）查找

二．双指针：例子（最长回文子串，两个指针从中间开始）

def fuzhu(left,right):

while left>=0 and right<len(s) and s[left]==s[right]:

left-=1

right+=1

return s[left+1:right]

result=""

for i in range(len(s)):

a=fuzhu(i,i)

b=fuzhu(i,i+1)

result=max(result,a,b,key=len)

return result

1. 排序算法：（与sort函数大致相同）

1.冒泡排序：从头到尾遍历待排序序列，比较相邻的元素，如果顺序错误则交换它们。每一次遍历将一个最大（或最小）元素"冒泡"到序列的末尾。直到没有元素交换，排序完成。

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

# 每次遍历时标记是否有交换，如果没有交换则提前退出

swapped = False

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

swapped = True

if not swapped:

break

return arr

四.算法难！

1.贪心算法：选择一个合适的数据结构来存储问题的解；在每一步选择中，选出当前最优的元素或动作；根据选出的元素更新当前的解和问题的状态；直到满足终止条件时，输出当前解

（背包例题）**问题描述**：给定一组物品，每个物品有重量和价值，背包有一个最大承重，要求在不超过背包容量的情况下，选择物品使得背包中的物品总价值最大。**贪心策略**（对于分数背包问题）：

计算每个物品的价值/重量比率，优先选择价值/重量比率最高的物品。（代码实现如下）

def fractional\_knapsack(weights, values, capacity):

n = len(weights)

items = [(values[i] / weights[i], values[i], weights[i]) for i in range(n)]

items.sort(reverse=True, key=lambda x: x[0]) # 按照价值/重量比排序

total\_value = 0

for value\_ratio, value, weight in items:

if capacity >= weight: # 背包可以装下整个物品

capacity -= weight

total\_value += value

else: # 背包不能装下整个物品，只能装部分

total\_value += value \* (capacity / weight)

break

return total\_value

（活动选择）**问题描述**：给定一个包含多个活动的列表，每个活动都有开始时间和结束时间。要求选择最大数量的互不重叠的活动。**贪心策略**：每次选择结束时间最早的活动，排除已经选择的活动与之重叠的活动。（代码实现如下，该段代码返回的是所选活动的索引）

def activity\_selection(start, end):

n = len(start)

activities = sorted(range(n), key=lambda i: end[i]) # 按结束时间排序

selected = []

# 选择第一个活动

last\_end\_time = -1

for i in activities:

if start[i] >= last\_end\_time: # 如果当前活动的开始时间不与前一个重叠

selected.append(i)

last\_end\_time = end[i] # 更新最后结束时间

return selected

（硬币找零）**问题描述**：给定不同面额的硬币和一个总金额，求使用最少的硬币数目来达到这个金额。如果没有任何一种硬币组合能组成总金额，返回 -1。**贪心策略**：每次选择当前面额最大的硬币，直到总金额为 0。（贪心策略只适用于某些货币系统，如1，2，10，25的系统）（代码实现如下）

def coin\_change(coins, amount):

coins.sort(reverse=True) # 从大到小排序硬币面额

coin\_count = 0

for coin in coins:

if amount >= coin:

count = amount // coin

coin\_count += count

amount -= count \* coin

if amount == 0:

break

return coin\_count if amount == 0 else -1

（机智的股民老张）直接看代码，就是贪心的基本步骤

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

def ans(prices):

min\_price=float('inf')

max\_price=0

for price in prices:

min\_price=min(min\_price,price)

max\_price=max(max\_price,price-min\_price)

return max\_price

print(ans(a)

2.动态规划：(定义状态，状态转移方程（重要），边界条件，计算顺序（从下到上，或从上到下）)

两种方式：（以斐波那契为例）

自顶向下，带备忘录： 第二种是自底向上，则较为常见：

def fib(n, memo={}): def fib(n):

if n in memo: if n <= 1:

return memo[n] return n

if n <= 1: dp=[0]\*(n+1)

return n dp[1]=1

memo[n] = fib(n-1, memo) + fib(n-2, memo) for i in range(2,n+1):

return memo[n] dp[i]=dp[i-1]+dp[i-2]

典型例题： return dp[n]

①背包问题：给定 n 个物品，每个物品有一个重量 w[i] 和价值 v[i]，还有一个背包，最大承重为 W。选择一些物品放入背包，使得背包中的物品的总价值最大，并且总重量不超过背包的最大承重。

状态转移方程：dp[i][w]=max(dp[i−1][w],dp[i−1][w−w[i]]+v[i]) （其中 dp[i-1][w] 表示不选第 i 个物品的最大价值，dp[i-1][w-w[i]] + v[i] 表示选第 i 个物品后的最大价值。）（代码如下：）

def knapsack(weights, values, W):

n = len(weights)

dp = [[0] \* (W + 1) for \_ in range(n + 1)]

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

for w in range(W + 1):

if w >= weights[i - 1]:

dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])

else:

dp[i][w] = dp[i - 1][w]

return dp[n][W]

②最长公共子序列：给定两个字符串 A 和 B，求它们的最长公共子序列的长度。子序列是指从字符串中删除某些字符（不改变其顺序）得到的字符串。

状态转移方程：（用 dp[i][j] 表示 A 的前 i 个字符和 B 的前 j 个字符的最长公共子序列长度。）

dp[i][j]=dp[i-1][j-1]+1(当A[i-1]=B[j-1])

dp[i][j]=max(dp[i-1][j],dp[i][j-1])(当A[i-1]!=B[j-1])(代码如下：)

def lcs(A, B):

m, n = len(A), len(B)

dp = [[0] \* (n + 1) for \_ in range(m + 1)]

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

for j in range(1, n + 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])

return dp[m][n]

③最短路径问题：给定一个加权有向图，找到从源点 S 到目标点 T 的最短路径。

状态转移方程（Floyd-Warshall算法）：dp[k][i][j]=min(dp[k−1][i][j],dp[k−1][i][k]+dp[k−1][k][j])（代码如下：）

def floyd\_warshall(graph):

n = len(graph)

dp = [row[:] for row in graph] # 深拷贝原图

for k in range(n):

for i in range(n):

for j in range(n):

if dp[i][j] > dp[i][k] + dp[k][j]:

dp[i][j] = dp[i][k] + dp[k][j]

return dp

④硬币找零：给定不同面额的硬币和一个总金额 amount，要求找出最少的硬币个数使得总金额为 amount。如果没有办法凑成该金额，返回 -1（用 dp[i] 表示金额 i 所需的最小硬币数，初始化 dp[0] = 0，其他为无穷大。）

状态转移方程：dp[i]=min(dp[i],dp[i−coin]+1) 其中 coin 是硬币的面额（代码如下）

def coinChange(coins, amount):

dp = [float('inf')] \* (amount + 1)

dp[0] = 0

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

for coin in coins:

if i - coin >= 0:

dp[i] = min(dp[i], dp[i - coin] + 1)

return dp[amount] if dp[amount] != float('inf') else -1

3.bfs（广度优先搜素）：模板

（一般的）from collections import deque

def bfs(start\_node, graph):

# 记录每个节点是否被访问过

visited = set() # 也可以使用列表或数组

# 队列，初始时将起始节点加入队列

queue = deque([start\_node])

# 将起始节点标记为已访问

visited.add(start\_node)

# 队列非空时，继续执行

while queue:

# 弹出队首元素

node = queue.popleft()

print(node) （或是到了目标点） # 处理当前节点（可以替换为其他逻辑）

# 遍历当前节点的所有邻居（通常是四个方向）

for neighbor in graph[node]:

if neighbor not in visited:

# 如果邻居没有被访问过，加入队列并标记为已访问

visited.add(neighbor)

queue.append(neighbor)

（优先队列，求最短路径的）

import heapq

n = int(input())

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

yi = [] #存储一的位置

for i in range(n):

for j in range(n):

if grid[i][j] == 1:

yi.append((i, j))

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

def fu(grid, x, y): #辅助函数，用来标记除起始点以外的点距起始点的位置都是正无穷

distance = {(x, y): 0}

for i in range(len(grid)):

for j in range(len(grid[i])):

if i != x or j != y:

distance[(i, j)] = float('inf')

return distance

def bfs(x, y, grid,tx,ty): #运用优先队列，记录下目标点到起始点的距离

pqueue = []

heapq.heappush(pqueue, (0, (x, y)))

seen = set()

distance = fu(grid, x, y)

while pqueue:

pair = heapq.heappop(pqueue)

dist = pair[0]

vertex = pair[1]

if vertex==(tx,ty):

return dist-1

break

seen.add(vertex)

for dx, dy in directions:

nx, ny = vertex[0] + dx, vertex[1] + dy

if 0 <= nx < n and 0 <= ny < n and (nx, ny) not in seen:

if dist + 1 < distance[(nx, ny)]:

heapq.heappush(pqueue, (dist + 1, (nx, ny)))

distance[(nx, ny)] = dist + 1

r=[]

for i in range(len(yi)):

r.append(bfs(yi[i][0], yi[i][1], grid,yi[-1][0],yi[-1][1]))

for i in r:

if i<=0:

r.remove(i)

#这一段，就是直接算出所有一的位置距离最后一个一的位置的距离，如果小于0.说明是最后一个点本身，如果等于0，说明是和最后一个点连在一起的岛

print(min(r))

4.dfs（深度优先搜索）

（一般模板）def dfs(node, graph, visited):

# 处理当前节点

print(node) # 可以根据需求修改为其他操作

# 将当前节点标记为已访问

visited.add(node)

# 遍历当前节点的所有邻居

for neighbor in graph[node]:

if neighbor not in visited:

dfs(neighbor, graph, visited)

def dfs\_init(graph, start):

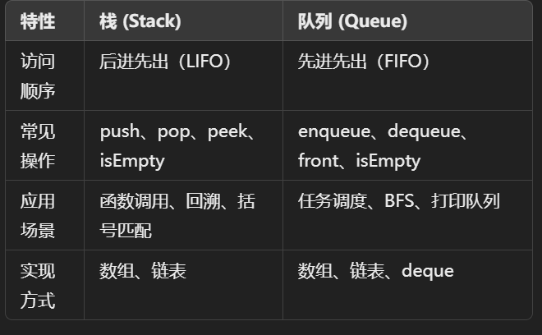
visited = set() # 用集合记录已访问的节点

dfs(start, graph, visited)

数据结构：

1.栈（stack）：栈是一种线性数据结构，遵循 **后进先出（LIFO）** 的原则，即最后加入的元素最先被移除。可以将其想象为一个堆叠的盘子，每次只能从顶部插入或移除元素。（可以直接理解为列表）

2.队列（queue）：主要运用于bfs （下图是区别） 3.栈运用于dfs



4.双端队列（deque）：

append：将元素添加到队列的尾部。

appendleft：将元素添加到队列的头部。

pop：从队列的尾部移除元素。

popleft：从队列的头部移除元素。

rotate：将队列的元素循环移动。

（双端队列和下面的优先队列在搜索中模板）

5.优先队列（priority queue）:首先要import heapq

heapq.heappush(pq, item)：将一个元素 item 插入到优先队列pq中。

heapq.heappop(pq)：删除并返回队列中的最小元素（对于最大优先队列，返回负数最小的元素）。

heapq.nsmallest(1, pq)：返回队列中优先级最小的一个元素，但不删除它。

heapq.merge(\*iterables)：将多个已排序的迭代器合并为一个排序后的迭代器。

6.堆（heapq）：import heapq

**①heapq.heapify(list)**：将一个列表转化为堆，时间复杂度是 O(N)。

**heapq.heappush(heap, item)**：将一个元素 item 插入堆中，时间复杂度是 O(log N)。

**heapq.heappop(heap)**：弹出并返回堆顶元素，时间复杂度是 O(log N)。

**heapq.heappushpop(heap, item)**：先将元素 item 插入堆中，然后弹出并返回堆顶元素，时间复杂度是 O(log N)。

**heapq.heapreplace(heap, item)**：弹出堆顶元素，并将 item 插入堆中，时间复杂度是 O(log N)。

②**实现最大堆**： Python heapq 默认是最小堆，如果要模拟最大堆，可以通过将元素取负数来实现：

相关问题：

一、前缀和，后缀和问题：

1.前缀和：对于一个数组，计算其从数组开头到任意位置的和，并将其存储起来。这样可以将区间和的查询问题转化为常数时间的查询。（给定一个数组 arr，前缀和 prefix[i] 表示数组 arr[0] 到 arr[i] 的和。）

def prefix\_sum(arr):

n = len(arr)

prefix = [0] \* (n + 1) # 用一个额外的空间来存储前缀和

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

prefix[i] = prefix[i - 1] + arr[i - 1] # prefix[i] = arr[0] + arr[1] + ... + arr[i-1]

return prefix

从l到r的区间和可用prefix[r+1]-prefix[l]来计算

2.后缀和：后缀和与前缀和类似，只不过它是计算从某个位置到数组末尾的和。（给定一个数组 arr，后缀和 suffix[i] 表示从 arr[i] 到 arr[n-1] 的和。）

def suffix\_sum(arr):

n = len(arr)

suffix = [0] \* (n + 1) # 用一个额外的空间来存储后缀和

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

suffix[i] = suffix[i + 1] + arr[i] # suffix[i] = arr[i] + arr[i+1] + ... + arr[n-1]

return suffix

从l到r的区间和可用suffix[l]-suffix[r+1]

二、最长递增/递减子序列问题：（动态规划）

1.最长递增子序列：给定一个整数序列，求该序列的 **最长递增子序列** 的长度。子序列不要求是连续的，但需要保持原有的相对顺序。

def length\_of\_lis(nums):

if not nums:

return 0

n = len(nums)

dp = [1] \* n # 初始化，所有元素的LIS至少为1

for i in range(1, n):

for j in range(i):

if nums[i] > nums[j]: # 如果 nums[i] > nums[j]，可以扩展子序列

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

return max(dp) # 最长递增子序列的长度是dp数组的最大值

二分查找优化：

import bisect

def length\_of\_lis(nums):

sub = [] # 用来保存构造的LIS的候选值

for num in nums:

pos = bisect.bisect\_left(sub, num) # 在sub中找到第一个不小于num的位置

if pos == len(sub):

sub.append(num) # 如果没有找到合适的位置，就追加到sub的末尾

else:

sub[pos] = num # 如果找到了合适的位置，更新sub中的值

return len(sub) # 最终的sub的长度就是LIS的长度

2.最长递减子序列：要找到最长递减子序列，我们可以对原始数组进行 **逆序处理**，然后使用求解最长递增子序列的方法。（直接nums.reverse()逆序即可，再套用上面的最长递增子序列）

三、搜索加上回溯：

1.马走日：马在中国象棋以日字形规则移动。请编写一段程序，给定n\*m大小的棋盘，以及马的初始位置(x，y)，要求不能重复经过棋盘上的同一个点，计算马可以有多少途径遍历棋盘上的所有点。

def dfs(x,y,visited,n,m,count):

if count==n\*m:

return 1

visited[x][y]=True

a=0

directions=[(-2,1),(2,1),(-2,-1),(2,-1),(-1,2),(-1,-2),(1,-2),(1,2)]

for dx,dy in directions:

nx,ny=x+dx,y+dy

**if** 0<=nx<n and 0<=ny<m and not visited[nx][ny]:

a+=dfs(nx,ny,visited,n,m,count+1)

visited[x][y]=False（这个回溯的意思就是如果遍历了directions之后，还是没有找到合适的点，那么就跳过这个点，不走这个点了，进行回溯）

return a

T=int(input())

for \_ in range(T):

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

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

print(dfs(x,y,visited,n,m,1))

2.矩阵最大权值路径：（从左上角到右下角）

第一种（dfs加上回溯）

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

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

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

# 用来记录当前路径的最大权值和

max\_path = []

max\_sum = float('-inf') # 初始设置为负无穷

# 深度优先搜索，寻找最大权值和的路径

def dfs(grid, x, y, current\_path, current\_sum, n, m):

global max\_path, max\_sum

# 记录当前节点和路径的权值

current\_path.append((x + 1, y + 1)) # 使用1-based索引

current\_sum += grid[x][y] # 累加当前节点的权值

# 如果到达终点，检查当前路径的权值和

if (x, y) == (n - 1, m - 1):

if current\_sum > max\_sum: # 如果当前路径的权值和更大，更新最大路径

max\_sum = current\_sum

max\_path = current\_path[:]

visited[x][y] = True # 标记当前节点已访问

# 尝试四个方向

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

for dx, dy in directions:

nx, ny = x + dx, y + dy

# 如果下一个位置合法且未访问过

if 0 <= nx < n and 0 <= ny < m and not visited[nx][ny]:

dfs(grid, nx, ny, current\_path, current\_sum, n, m)

visited[x][y] = False # 回溯，标记当前节点未访问

current\_path.pop() # 回溯时，从路径中移除当前节点

# 调用 DFS 函数进行深度优先搜索

dfs(grid, 0, 0, [], 0, n, m)

# 输出最大权值和路径

for x, y in max\_path:

print(x, y)

第二种：（bfs+优先队列）

import heapq

# 输入

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

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

# BFS 函数，用优先队列来实现最大权值路径

def bfs(grid, n, m):

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

max\_path = []

max\_sum = float('-inf') # 初始最大路径权值和设置为负无穷

# 使用最大堆来存储状态，堆中的元素为 (-current\_sum, x, y, path)，负值是为了模拟最大堆

pq = []

heapq.heappush(pq, (-grid[0][0], 0, 0, [(1, 1)])) # 初始状态，(权值和，x，y，路径)

# 记录每个位置的最大权值和

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

visited[0][0] = True # 起点标记为已访问

while pq:

current\_sum, x, y, path = heapq.heappop(pq)

current\_sum = -current\_sum # 还原为正数

# 如果到达终点，更新最大路径

if (x, y) == (n - 1, m - 1):

if current\_sum > max\_sum:

max\_sum = current\_sum

max\_path = path

# 探索相邻节点

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < n and 0 <= ny < m and not visited[nx][ny]:

visited[nx][ny] = True

new\_path = path + [(nx + 1, ny + 1)] # 记录路径，1-based indexing

heapq.heappush(pq, (-(current\_sum + grid[nx][ny]), nx, ny, new\_path))

return max\_path

# 调用 BFS 函数

max\_path = bfs(grid, n, m)

# 输出结果

for x, y in max\_path:

print(x, y)

四、滑动窗口：滑动窗口的基本思想是利用两个指针维护一个“窗口”，并在遍历数组的过程中，通过调整窗口的大小或位置来满足问题的条件。（左指针（left）表示窗口开始，右则亦然）

1.类型：固定和动态（前者是长度不变，后者则变化）

2.经典问题：

①无重复字符的最长子串（给定一个字符串 s，找出其中没有重复字符的最长子串的长度。）

解法（使用滑动窗口的思想，维护一个窗口，窗口内不包含重复字符。通过移动右指针扩大窗口，并使用左指针收缩窗口来确保窗口内的字符唯一。）

代码实现：

def lengthOfLongestSubstring(s: str) -> int:

# 使用字典来记录字符的最新位置

char\_set = set()

left = 0

max\_len = 0

for right in range(len(s)):

# 如果字符已经在窗口内，左指针右移直到字符不重复

while s[right] in char\_set:

char\_set.remove(s[left])

left += 1

# 当前字符加入窗口

char\_set.add(s[right])

# 更新最长长度

max\_len = max(max\_len, right - left + 1)

return max\_len

②最大子数组和（给定一个整数数组 nums 和一个整数 k，找到所有长度为 k 的子数组的和的最大值。）

解法：可以使用滑动窗口维护当前窗口内 k 个元素的和。当右指针向右移动时，我们加上新元素的值；当窗口大小达到 k 时，左指针向右移动，删除窗口最左边的元素，从而保持窗口的大小为 k。

代码实现如下：

def maxSum(nums: list, k: int) -> int:

window\_sum = sum(nums[:k]) # 初始窗口的和

max\_sum = window\_sum

for i in range(k, len(nums)):

# 滑动窗口，加入新的元素并删除最左边的元素

window\_sum += nums[i] - nums[i - k]

max\_sum = max(max\_sum, window\_sum)

return max\_sum

③最小覆盖子串（给定一个字符串 s 和一个字符串 t，返回 s 中覆盖 t 所有字符的最小子串。）

解法：我们可以使用滑动窗口和字符频率计数的方法来解这个问题。我们维护一个窗口，窗口内包含 t 中所有的字符，并尝试通过移动左指针来找出最小的覆盖子串。

代码如下：

from collections import Counter

def minWindow(s: str, t: str) -> str:

if not s or not t:

return ""

# 记录t中的字符频率

t\_count = Counter(t)

window\_count = Counter()

left = 0

right = 0

required = len(t\_count) # 需要匹配的不同字符数

formed = 0 # 当前窗口匹配的字符数

min\_len = float("inf")

min\_left = 0

while right < len(s):

# 扩大窗口，加入右边字符

window\_count[s[right]] += 1

# 如果窗口内的字符数满足t中某个字符的要求

if window\_count[s[right]] == t\_count[s[right]]:

formed += 1

# 缩小窗口，保持窗口仍然包含t中的所有字符

while left <= right and formed == required:

if right - left + 1 < min\_len:

min\_len = right - left + 1

min\_left = left

# 收缩窗口，删除左边的字符

window\_count[s[left]] -= 1

if window\_count[s[left]] < t\_count[s[left]]:

formed -= 1

left += 1

right += 1

return s[min\_left:min\_left + min\_len] if min\_len != float("inf") else ""