工具：

int(str,base)进制转换，如int(‘101’,2)输出5

for key, value in dict.items()

for index, value in enumerate(list)

dict.get(key, default) 返回指定键的值，如果不存在则返回default

list(zip(a, b)) 将两个列表中的值一一对应打包

math.pow(m, n) m的n次方

math.log(m, n) 以n为底m的对数（默认以e为底）

math.ceil()

math.floor()

math.gcd()最大公约数

math.lcd()最小公倍数

from functools import lru\_cache

@lru\_cache(maxsize=None)

from sys import setrecursionlimit

sys.setrecursionlimit(300000)或者1<<30

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

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。

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

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

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。

读取空行：

x = input()

if not x:

balabala

多行读取：  
while True:  
 try:  
 ...

except EOFError:  
 break

找到对应值的键：

string = "Words are but wind"

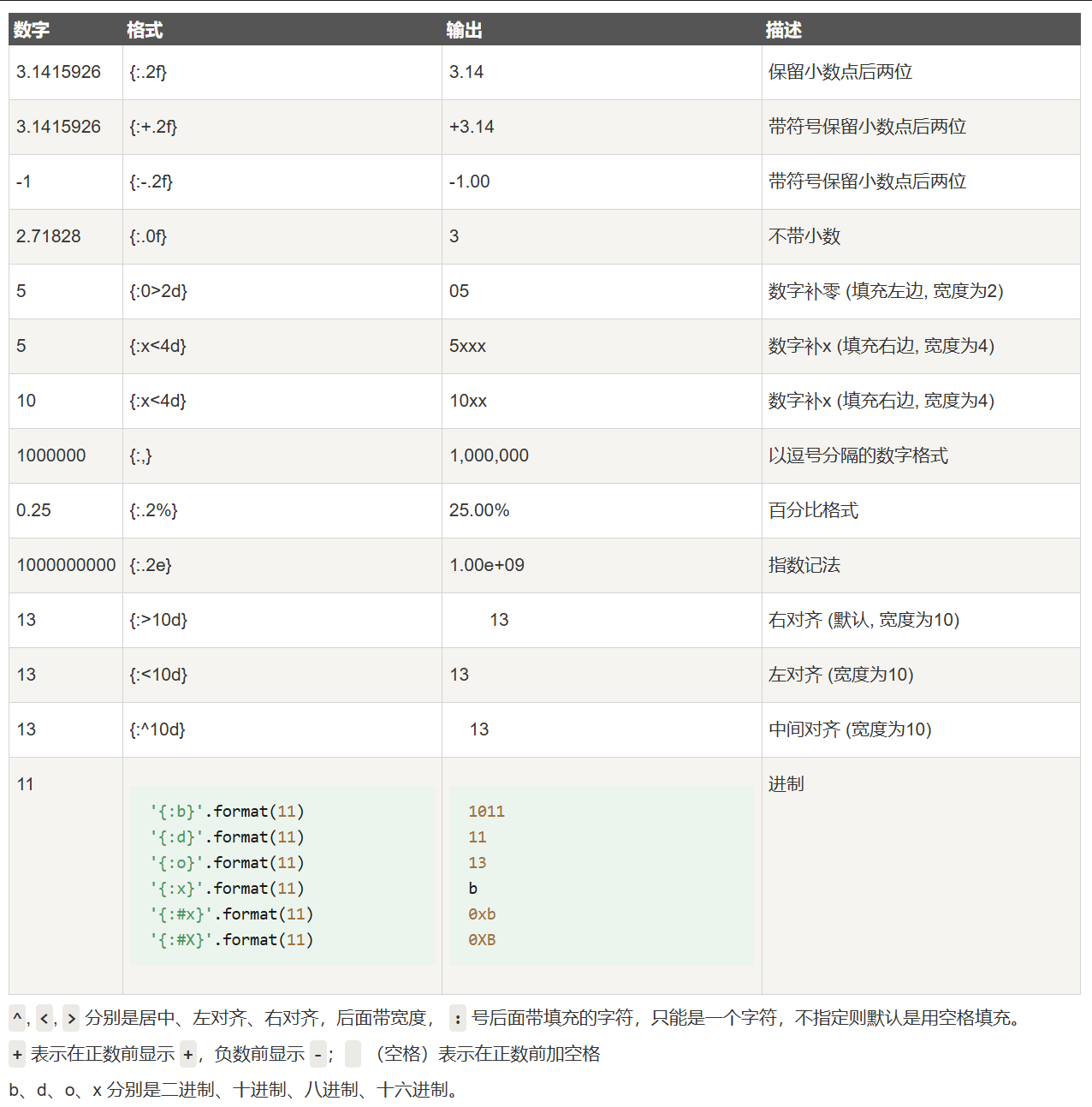
word\_order = {el: ind+1 for ind, el in enumerate(string.split())}

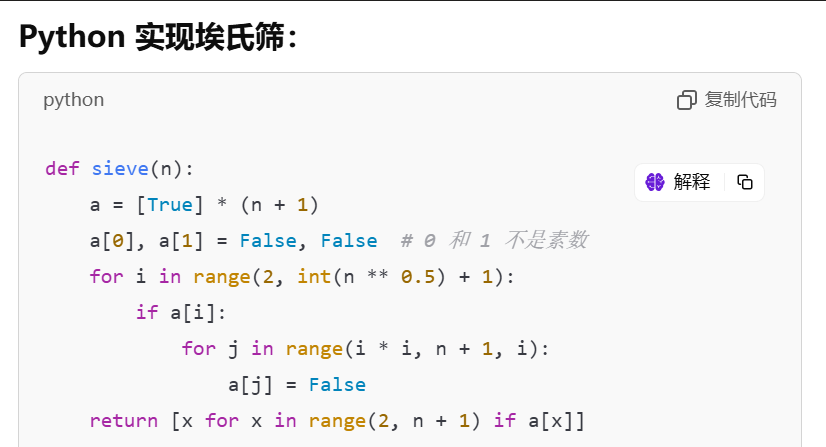
print(word\_order)

sort()中可以加入次级优先级：courses.sort(key = lambda x: (x[2],x[0]), reverse = True)

zip（）函数，将两个列表（字符串）各个元素依次连接为元祖后成为新的列表 list（zip（a， b））

用extend连接两个列表





calendar

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。

counter：计数

from collections import Counter

a=['red', 'blue', 'red', 'green', 'blue', 'blue']

a=Counter(a)

Counter()

主要功能：可以支持方便、快速的计数，将元素数量统计，然后计数并返回一个字典，键为元素，值为元素个数。

from collections import Counter

list1 = ["a", "a", "a", "b", "c", "c", "f", "g", "g", "g", "f"]

dic = Counter(list1)

print(dic)

#结果:次数是从高到低的

#Counter({'a': 3, 'g': 3, 'c': 2, 'f': 2, 'b': 1})

print(dict(dic))

#结果:按字母顺序排序的

#{'a': 3, 'b': 1, 'c': 2, 'f': 2, 'g': 3}

print(dic.items()) #dic.items()获取字典的key和value

#结果:按字母顺序排序的

#dict\_items([('a', 3), ('b', 1), ('c', 2), ('f', 2), ('g', 3)])

print(dic.keys())

#结果:

#dict\_keys(['a', 'b', 'c', 'f', 'g'])

print(dic.values())

#结果：

#dict\_values([3, 1, 2, 2, 3])

print(sorted(dic.items(), key=lambda s: (-s[1])))

#结果:按统计次数降序排序

#[('a', 3), ('g', 3), ('c', 2), ('f', 2), ('b', 1)]

most\_common()

返回一个列表，包含counter中n个最大数目的元素，如果忽略n或者为None，most\_common()将会返回counter中的所有元素，元素有着相同数目的将会选择出现早的元素

list1 = ["a", "a", "a", "b", "c", "f", "g", "g", "c", "11", "g", "f", "10", "2"]

print(Counter(list1).most\_common(3))

#结果：[('a', 3), ('g', 3), ('c', 2)]

#"c"、"f"调换位置，结果变化

list2 = ["a", "a", "a", "b", "f", "c", "g", "g", "c", "11", "g", "f", "10", "2"]

print(Counter(list2).most\_common(3))

#结果：[('a', 3), ('g', 3), ('f', 2)]

update()

从一个可迭代对象（可迭代对象是一个元素序列，而非(key,value)对构成的序列）中或者另一个映射（或counter）中所有元素相加，是数目相加而非替换它们

dic1 = {'a': 3, 'b': 4, 'c': 0, 'd': -2, "e": 0}

dic2 = {'a': 3, 'b': 4, 'c': 0, 'd': 2, "e": -1, "f": 6}

a = Counter(dic1)

print(a)

#结果:Counter({'b': 4, 'a': 3, 'c': 0, 'e': 0, 'd': -2})

b = Counter(dic2)

print(b)

#结果:Counter({'f': 6, 'b': 4, 'a': 3, 'd': 2, 'c': 0, 'e': -1})

a.update(b)

print(a)

#结果：Counter({'b': 8, 'a': 6, 'f': 6, 'c': 0, 'd': 0, 'e': -1})

subtract()

从一个可迭代对象中或者另一个映射（或counter）中，元素相减，是数目相减而不是替换它们

dic1 = {'a': 3, 'b': 4, 'c': 0, 'd': -2, "e": 0}

dic2 = {'a': 3, 'b': 4, 'c': 0, 'd': 2, "e": -1, "f": 6}

a = Counter(dic1)

print(a)

#结果：Counter({'b': 4, 'a': 3, 'c': 0, 'e': 0, 'd': -2})

b = Counter(dic2)

print(b)

#结果：Counter({'f': 6, 'b': 4, 'a': 3, 'd': 2, 'c': 0, 'e': -1})

a.subtract(b)

print(a)

#结果：Counter({'e': 1, 'a': 0, 'b': 0, 'c': 0, 'd': -4, 'f': -6})

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))

bisect

import bisect

# 创建一个已排序的列表

sorted\_list = [1, 3, 3, 6, 7, 9]

# 使用 bisect\_left 查找元素应插入的位置

insert\_index = bisect.bisect\_left(sorted\_list, 4)

print("Insert at index:", insert\_index)

# 使用 insort\_left 插入元素并保持有序

bisect.insort\_left(sorted\_list, 4)

print("Updated list:", sorted\_list)

dijkstra

这个版本的Dijkstra算法使⽤了⼀个集合 visited 来记录已经访问过的节点，这样可以避免对同⼀个节点的重复处

理。当我们从优先队列中取出⼀个节点时，如果这个节点已经在 visited 集合中，那么我们就跳过这个节点，处理

下⼀个节点。这样可以提⾼算法的效率。

此外，这个版本的Dijkstra算法还在找到⽬标节点 t 时就⽴即返回结果，⽽不是等到遍历完所有节点。这是因为

Dijkstra算法保证了每次从优先队列中取出的节点就是当前距离最短的节点，所以当我们找到⽬标节点 t 时，就已

经找到了从起始节点 s 到 t 的最短路径，⽆需再继续搜索。

堆中存储的所有节点的距离是递增的，且不会再变小。

def dijkstra(n, edges, s, t):

graph = [[] for \_ in range(n)]

for u, v, w in edges:

graph[u].append((v, w))

graph[v].append((u, w))

pq = [(0, s)] # (distance, node)

visited = set()

distances = [float('inf')] \* n

distances[s] = 0

while pq:

dist, node = heapq.heappop(pq)

if node == t:

return dist

if node in visited:

continue

visited.add(node)

for neighbor, weight in graph[node]:

if neighbor not in visited:

new\_dist = dist + weight

if new\_dist < distances[neighbor]:

distances[neighbor] = new\_dist

heapq.heappush(pq, (new\_dist, neighbor))

return -1

**走山路**

import heapq

def dijkstra():

heap = []

heapq.heappush(heap, (0, start\_x, start\_y))

min\_cost = [[float('inf')] \* n for \_ in range(m)]

min\_cost[start\_x][start\_y] = 0

while heap:

num, x, y = heapq.heappop(heap)

if num > min\_cost[x][y]:

continue

if x == end\_x and y == end\_y:

return num

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < m and 0 <= ny < n and matrix[nx][ny] != '#':

cost = num + abs(int(matrix[nx][ny]) - int(matrix[x][y]))

if cost < min\_cost[nx][ny]:

min\_cost[nx][ny] = cost

heapq.heappush(heap, (cost, nx, ny))

return 'NO'

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

matrix = [input().split() for \_ in range(m)]

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

for \_ in range(p):

start\_x, start\_y, end\_x, end\_y = map(int, input().split())

if matrix[start\_x][start\_y] == '#' or matrix[end\_x][end\_y] == '#':

print('NO')

else:

print(dijkstra())

冒泡排序

def BubbleSort(arr):

for i in range(len(arr) - 1):

for j in range(len(arr) - i - 1):

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

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

return arr

dp

1. k-Tree

双列表dp

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

mod = 10\*\*9 + 7

dp1 = [1] + [0] \* n

dp2 = [1] + [0] \* n

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

for j in range(1, min(i, k) + 1):

dp1[i] = (dp1[i] + dp1[i-j]) % mod

for j in range(1, min(d, i + 1)):

dp2[i] = (dp2[i] + dp2[i-j]) % mod

print((dp1[n]-dp2[n])%mod)