Cheat\_sheet

语法篇

数据结构：

字符串：

建立（转为）：数字转字符串：str()

列表转字符串：“（连接符）”.join(list)

处理：分割：str.split(“（分隔符）”)

大写：str.upper()

小写：str.lower()

首字母：str.title()

合并：+

特征：长度：len()

列表：

建立：list()

列表推导式：[声明变量 for 声明变量 in 某集合 if 共同满足的条件]

处理：切片：list[起点：终点+1]（不填默认起点为list[0],终点为列表末尾）

遍历：for （临时变量名） in list：

在末尾添加元素：list.append(元素)

添加多个元素：list.extend([a,b,c,...])

插入元素：list.insert(index,元素)；bisect库

删除已知元素：list.remove(元素)

删除已知索引的元素：del list[index]

弹出元素：list.pop(index)

顺序排序：list.sort()（ASCII码顺序）

倒序排序：list.sort(reverse=True)

指定顺序排序：list.sort(key= lambda s:排序指标（与s相关）)

拼接：list1+list2

特征：长度：len()

寻找索引：list.index(元素)

正数第n个元素：list[n-1]

倒数第n个元素：list[-n]

元素个数：list.count(元素)

itertools 库

判断共有特征：all(特征 for 元素 in 列表)

索引，元素元组：enumerate()函数（遍历方法：for index，元素代称 in enumerate(列表)）

字典：

建立：{}

dict(元组)

半有序：Ordereddict()

添加/修改键值对：dict[key]=value

遍历字典的键：for 元素 in dict() ； for 元素 in dict.keys()（注：一定要加s!）

遍历字典的值：for 元素 in dict.values()（一定要加s!）

删除键值对：del dict[键]

遍历键值对：for key,value in dict.items():

按顺序遍历：for key in sorted(dict.keys()):

元组：建立：直接定义：(...,...,...,...)

含元组的列表：zip(a,b,c,...)

访问：元组[index]

集合：建立：set()

向集合中添加元素：set.add()

添加多个元素：set.update()

删除元素：set.remove() 或set.discard()（前者有KeyError风险，后者没有）

随机删除：set.pop()

并集：set1 | set2（竖杠“|”在回车键上方）

交集：set1 & set2

差集（补集）：set1 - set2

对称差集（补集之交）：set1^set2

元素个数：len()

不可变集合：frozenset()

堆：heapq库

库：

注意：库中函数要先import之后才能使用。

math库：向上取整：math.ceil()

向下取整：math.floor()

阶乘：math.factoria()

数学常数：math.pi（圆周率），math.e（自然对数的底）

开平方：math.sqrt(x)

x的y次幂：math.pow(x,y)

e的x次幂：math.exp(x)

对数函数：math.log(真数，底数)（不填底数默认为自然对数）

三角：math.sin(),math.cos(),math.tan()

反三角：math.asin(),math.acos(),math.atan()

heapq库：列表转堆：最小值在上层：heapq.heapify(list)；最大值在上层：heapq.\_heapify\_max(list)

插入元素：heapq.heappush(堆名，被插元素)

弹出元素：heapq.heappop(堆名)（可被命名为其他变量临时调用）

（应用：堆排序：a=[heapq.heappop(b) for \_ in range(len(b))],返回排序后的b）

插入元素的同时弹出顶部元素：heapq.heappushpop(堆名，被插元素)

（或heapq.heapreplace(堆名，被插元素)）

以上操作在最大堆中应换为“\_a\_max”（a是它们中的任意一个）

建堆时，先定义一个空列表，然后一个一个往里面压入元素。

itertools库：

整数集：itertools.count(x,y)（从x开始往大数的整数，间隔为y）

循环地复制一组变量：itertools.cycle(list)

所有排列：itertools.permutations(集合，选取个数)

所有组合：itertools.combinations

拼接列表的另一种方式：itertools.chain(list1,list2)

已排序列表去重：[i for i,\_ in itertools.groupby(list)]（每种元素只能保留一个）

或者list(group)[:n]（group被定义为分组，保留每组的n个元素）

collections库：

双端队列：创建：a=deque(list)

从末尾添加元素：a.append(x)

从开头添加元素：a.appendleft(x)

从末尾删除元素：b=a.pop()

从开头删除元素：b=a.popleft()

（其中b用于接收a弹出的元素）

有序字典：Ordereddict()

默认值字典：a=defaultdict(默认值)，如果键不在字典中，会自动添加值为默认值的键值对，而不报KeyError。

计数器：Counter(str)，返回以字符种类为键，出现个数为值的字典

sys库：sys.exit()用于及时退出程序

sys.setrecursionlimit()用于调整递归限制（尽量少用，递归层数过多会引起MLE）

statistics库：statistics 是 Python 标准库中用于统计学计算的模块，提供了各种用于处理统计数据的函数。以下是 statistics 模块中一些常用函数的简要介绍：

1.mean(data)：计算数据的平均值（均值）。

2.harmonic\_mean(data)：计算数据的调和平均数。

3.median(data)：计算数据的中位数。

4.median\_low(data)：计算数据的低中位数。

5.median\_high(data)：计算数据的高中位数。

6.median\_grouped(data, interval=1)：计算分组数据的估计中位数。

7.mode(data)：计算数据的众数。

8.pstdev(data)：计算数据的总体标准差。

9.pvariance(data)：计算数据的总体方差。

10.stdev(data)：计算数据的样本标准差。

11.variance(data)：计算数据的样本方差。

这些函数能够帮助你在 Python 中进行常见的统计计算。在使用这些函数时，你需要将数据作为参数传递给相应的函数，然后函数会返回计算结果。

数据处理：二进制：bin()，八进制：oct()，十六进制：hex()，整型：int()，浮点型：float(),

保留n位小数：round(原数字，保留位数)（如不写保留位数，则默认保留到整数）；’%.nf’%原数字；’{:.nf}’.format(原数字)；

n位有效数字：’%.ng’%原数字；’{:.ng}’.format(原数字)

最大值max(),最小值min()

ASCII转字符：chr();字符转ASCII：ord()

判断数据类型：isinstance(object,class)

其他：if，while循环；try，except 某error；

类的创建：class type(father):

def \_\_init\_\_(self,specific\_level):

self.character=specific\_level

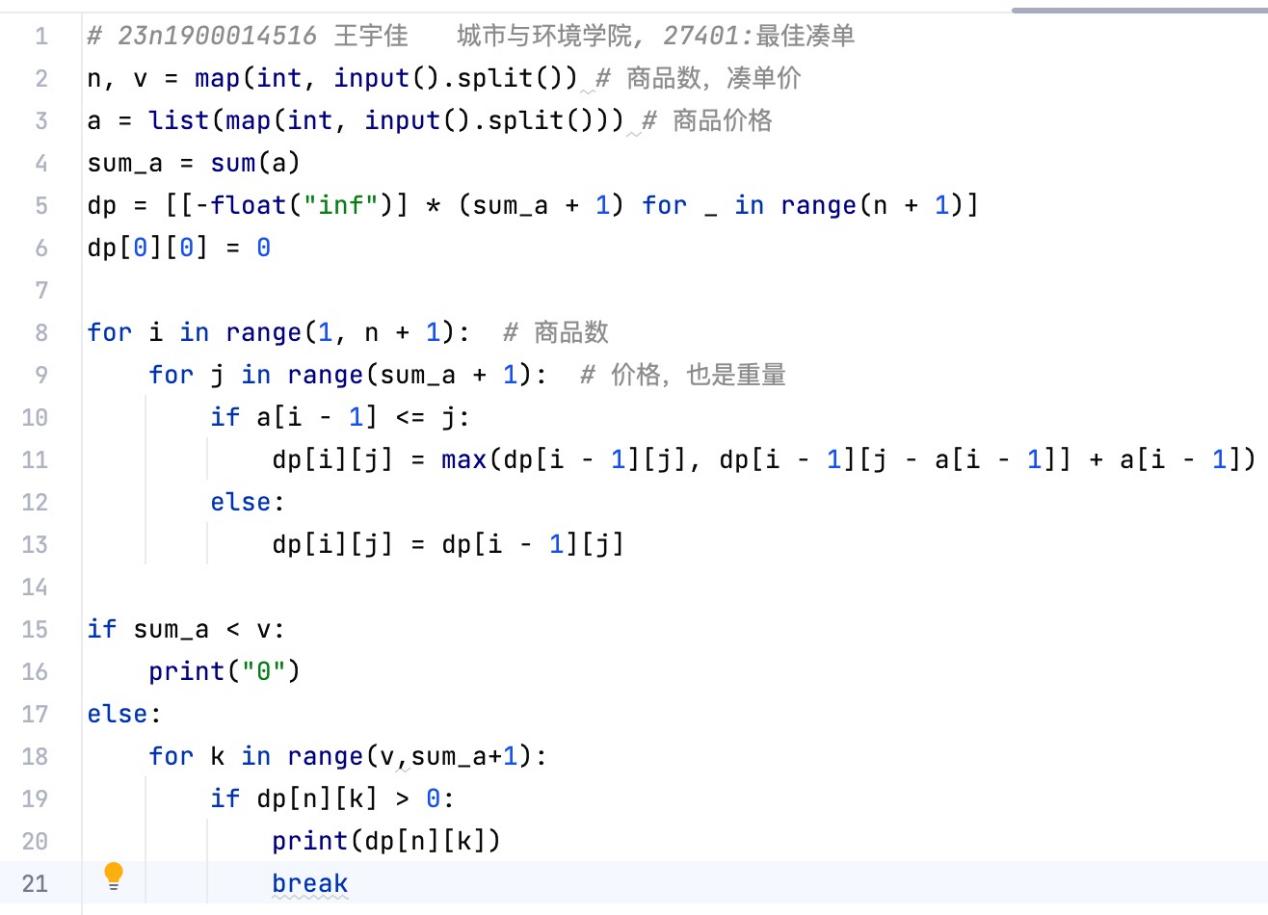
算法篇

dp（动态规划）

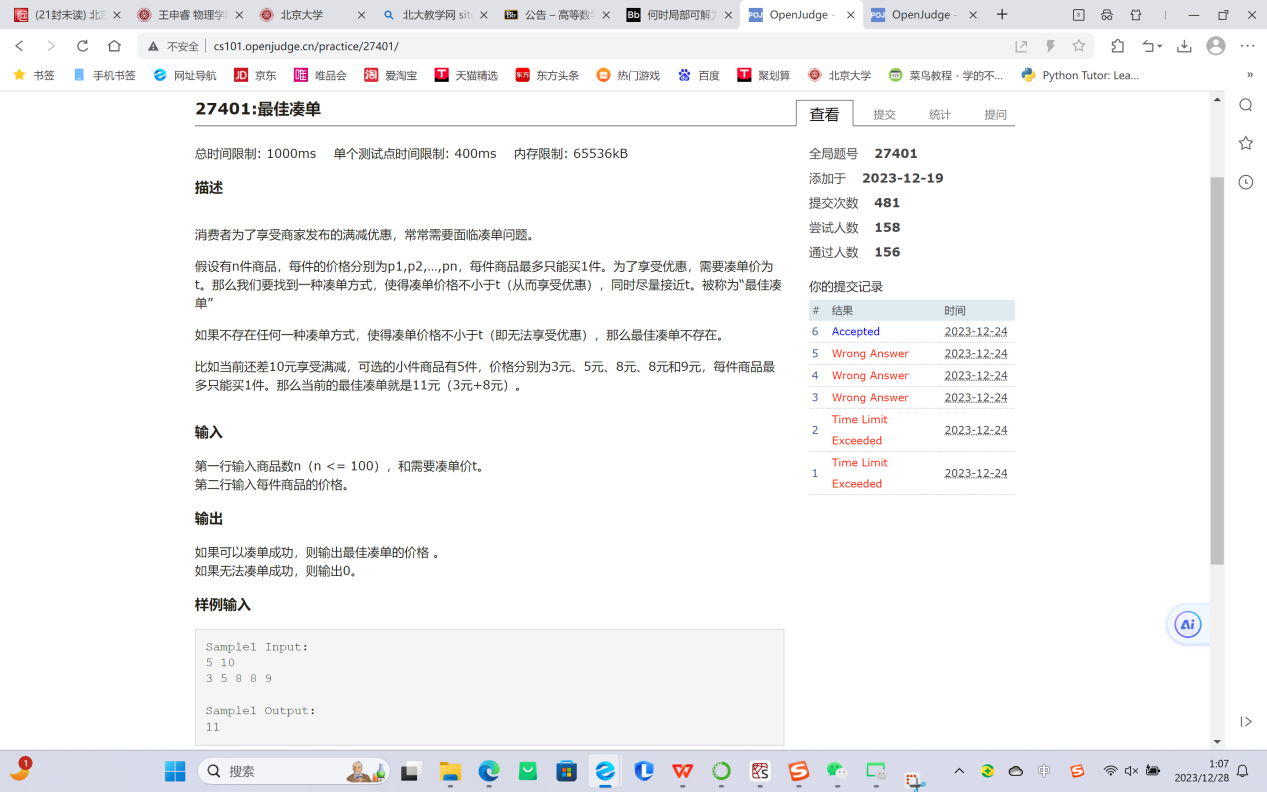
步骤：1、定义矩阵（全零或负无穷）

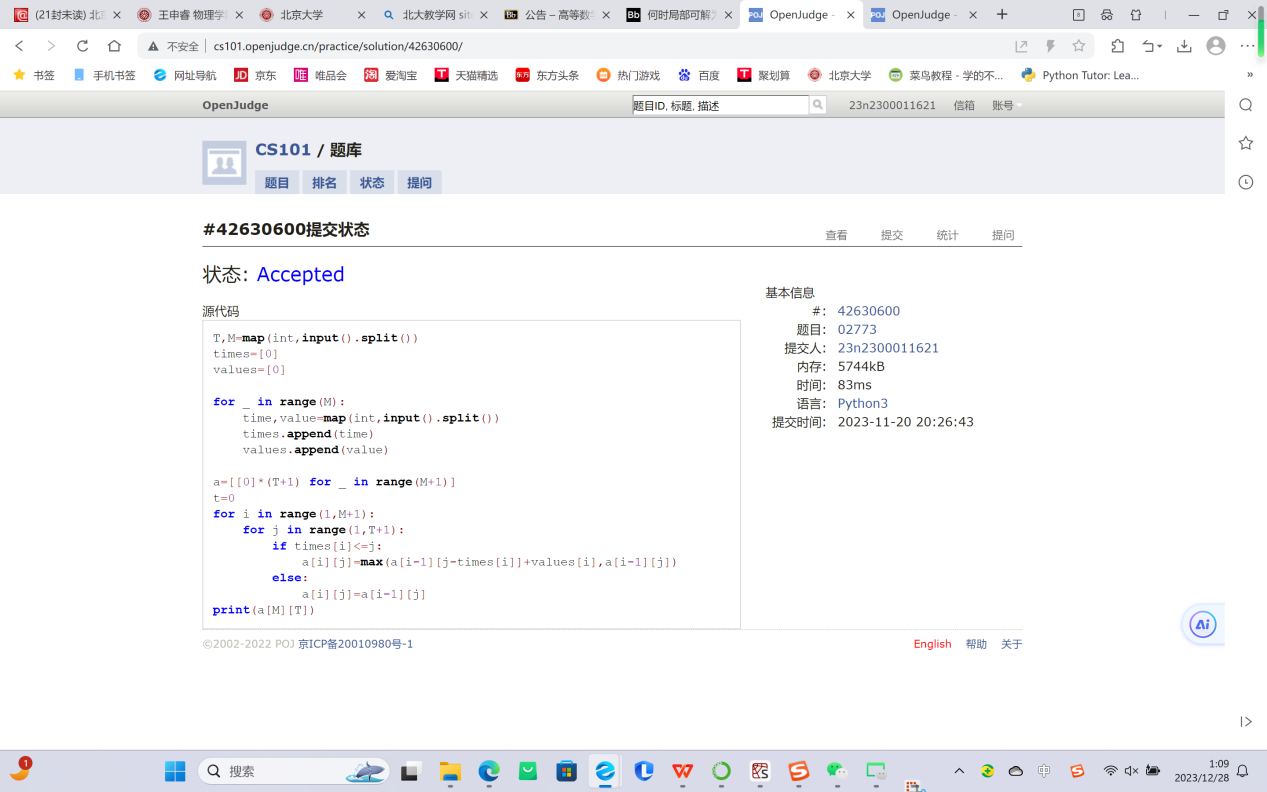
1. 遍历矩阵（顺带遍历可选项）
2. 遇到可放入：状态转移方程：a[i][j]=max(a[i-1][j-t]+value[t],a[i-1][j]),t为物品对空间的占用
3. 按情况输出矩阵的一个格子（通常是a[-1][-1]）

模版：



（原题）





（原题）



dfs（深度优先搜索）

步骤：1、定义函数；

1. 在函数内部，判断是否终了，若是，存下状态，return；
2. 一层层地判断：符合条件就打标记，递归调用，进入下一层；记得在递归调用完后抹除标记，以便搜索其他分支；
3. 在函数外调用函数，注意实参是否正确；
4. 输出可能结果

模版：

（八皇后）

ans = []

def queen\_dfs(A, cur=0): #考虑放第cur行的皇后

if cur == len(A): #如果已经放了n个皇后，一组新的解产生了

ans.append(''.join([str(x+1) for x in A])) #注意避免浅拷贝

return

for col in range(len(A)): #将当前皇后逐一放置在不同的列，每列对应一组解

for row in range(cur): #逐一判定，与前面的皇后是否冲突

#因为预先确定所有皇后一定不在同一行，所以只需要检查是否同列，或者在同一斜线上

if A[row] == col or abs(col - A[row]) == cur - row:

break

else: #若都不冲突

A[cur] = col #放置新皇后，在cur行，col列

#对下一个皇后位置进行递归

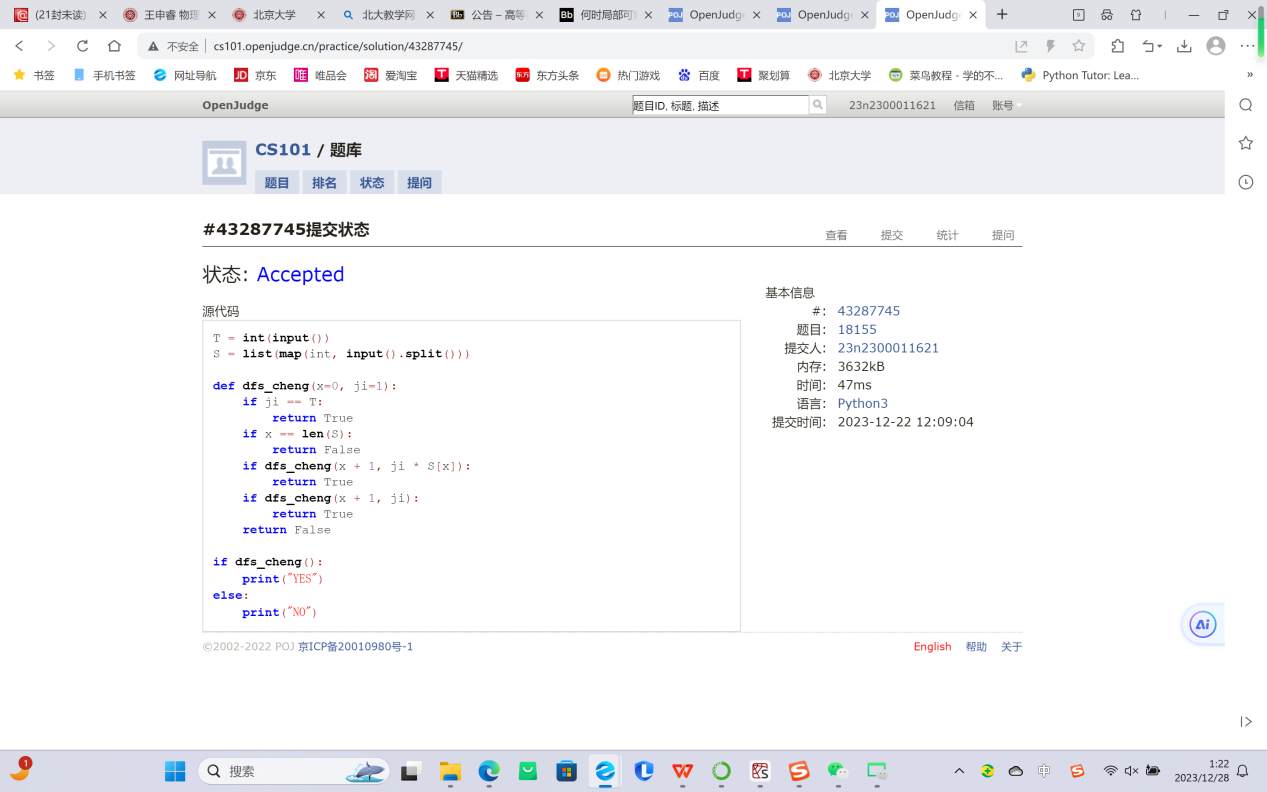
queen\_dfs([None]\*8)

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

print(ans[int(input()) - 1])

（组合乘积）





bfs（广度优先搜索）

定义好所有方向，按方向遍历就行了。记得标记好走过的地点。

（步骤：1）：from collections import deque；2）：def bfs(start,end):；3）：初始化一个包含起点和步数的双端队列；4）：；5）：；）

模版：

## 19930: 寻宝

bfs, http://cs101.openjudge.cn/practice/19930

Billy获得了一张藏宝图，图上标记了普通点（0），藏宝点（1）和陷阱（2）。按照藏宝图，Billy只能上下左右移动，每次移动一格，且途中不能经过陷阱。现在Billy从藏宝图的左上角出发，请问他是否能到达藏宝点？如果能，所需最短步数为多少？\*\*输入\*\*

第一行为两个整数m,n，分别表示藏宝图的行数和列数。(m<=50,n<=50)

此后m行，每行n个整数（0，1，2），表示藏宝图的内容。

\*\*输出\*\*

如果不能到达，输出‘NO’。

如果能到达，输出所需的最短步数（一个整数）。

样例输入

```

样例输入1：

3 4

0 0 2 0

0 2 1 0

0 0 0 0

样例输出1：

5

```

样例输出

```

样例输入2：

2 2

0 2

2 1

样例输出2:

NO

```

提示

每张藏宝图有且仅有一个藏宝点。

输入保证左上角（起点）不是陷阱。

来源：by cs101-2009 邵天泽

其实所有求最短、最长的问题都能用heapq实现，在图搜索中搭配bfs尤其好用。

```python

#23 工学院 苏王捷

import heapq

def bfs(x,y):d=[[-1,0],[1,0],[0,1],[0,-1]]

queue=[]

heapq.heappush(queue,[0,x,y])

check=set()

check.add((x,y))

while queue:

step,x,y=map(int,heapq.heappop(queue))

if martix[x][y]==1:

return step

for i in range(4):

dx,dy=x+d[i][0],y+d[i][1]

if martix[dx][dy]!=2 and (dx,dy) not in check:

heapq.heappush(queue,[step+1,dx,dy])

check.add((dx,dy))

return "NO"

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

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

print(bfs(1,1))

```

找最短路径，是bfs，借助队列queue实现，head和tail分别指向队列的头和尾。

```python

q = []

step = [[0, 1], [1, 0], [-1, 0], [0, -1]]

vis = [[0] \* 52 for \_ in range(52)]

g = []

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

for i in range(m):

g.append([int(x) for x in input().split()])

def check(x, y):

if (x < 0 or y < 0 or x >= m or y >= n):

return False

if (vis[x][y] or g[x][y] == 2):

return False

return True

q.append((0, 0))

head = 0

tail = 1

level = 0

while (head < tail):

# i = head

# j = tailfor k in range(head, tail):

x, y = q[head]

head += 1

if (g[x][y] == 1):

print(level)

exit(0)

for z in range(4):

newx = x + step[z][0]

newy = y + step[z][1]

if (check(newx, newy)):

vis[newx][newy] = 1

q.append((newx, newy))

tail += 1

level += 1

print('NO')

```

二分查找

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ### （3）bisect（二分查找） |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| ```python |  |  |  |  |  |  |  |  |
| 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) |  |  |  |  |  |  |  |  |
| 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) |  |  |  |  |  |  |  |  |
| 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 |  |  |  |  |  |  |  |  |

欧拉筛

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N=20 |  |  |  |  |  |  |
| primes = [] |  |  |  |  |  |  |
| is\_prime = [True]\*N |  |  |  |  |  |  |
| is\_prime[0] = False;is\_prime[1] = False |  |  |  |  |  |  |
| for i in range(2,N): |  |  |  |  |  |  |
| if is\_prime[i]: |  |  |  |  |  |  |
| primes.append(i) |  |  |  |  |  |  |
| for p in primes: #筛掉每个数的素数倍 |  |  |  |  |  |  |
| if p\*i >= N: |  |  |  |  |  |  |
| break |  |  |  |  |  |  |
| is\_prime[p\*i] = False |  |  |  |  |  |  |
| if i % p == 0: #这样能保证每个数都被它的最小素因数筛掉！ |  |  |  |  |  |  |
| break |  |  |  |  |  |  |
| print(primes) |  |  |  |  |  |  |
| # [2, 3, 5, 7, 11, 13, 17, 19] |  |  |  |  |  |  |

树：

准备工作：

class TreeNode():

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

self.value=val

self.left=None

self.right=None

N=int(input())

nodes=[TreeNode(i) for i in range(N)]#存放节点，防止重复创建实例

（如果实在难以避免重复引用TreeNode（i），可以采用以下方法创建类：

class TreeNode:

\_instances = {}

def \_\_new\_\_(cls, value):

if value not in cls.\_instances:

instance = super().\_\_new\_\_(cls)

cls.\_instances[value] = instance

return instance

else:

return cls.\_instances[value]

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

if not hasattr(self, 'initialized'):

self.value = value

self.left = None

self.right = None

self.initialized = True

def \_\_eq\_\_(self, other):

return isinstance(other, TreeNode) and self.value == other.value

def \_\_hash\_\_(self):

return hash(self.value)

）

常见建树函数：

前中建后：

def find\_node(a,b):

if (not a) and (not b):

return

root=a[0]

for i in range(len(b)):

if b[i].value==root.value:

root2=i

b\_left=b[:root2]

b\_right=b[root2+1:]

a\_left=a[1:root2+1]

a\_right=a[root2+1:]

root.left=find\_node(a\_left,b\_left)

root.right=find\_node(a\_right,b\_right)

return root

中后建前：

def find\_node(a,b):

if (not a) and (not b):

return

root=b[-1]

for i in range(len(a)):

if a[i].value==root.value:

root2=i

break

a\_left=a[:root2]

a\_right=a[root2+1:]

b\_left=b[:root2]

b\_right=b[root2:len(b)-1]

root.right=find\_node(a\_right,b\_right)

root.left=find\_node(a\_left,b\_left)

return root

表达式：

优先表达式：以前序为例：

def qianxu(x):

res=x.value

if x.left is not None:

res+=qianxu(x.left)

if x.right is not None:

res+=qianxu(x.right)

return res

层级遍历表达式：BFS

并查集：

步骤：

1. 初始化：将每一个节点的归属设为其自身；
2. 判断：判断两个节点是否同归属，在判断的过程中路径压缩；
3. 合并：将节点的归属按要求重新设置；
4. 关于路径压缩：只要节点的归属不是其父节点，就将其父节点设为总的根节点

模版：

（列表表示对象法）

def find(x):

if parent[x] != x:

parent[x] = find(parent[x])

return parent[x]

def union(x, y):

root\_x = find(x)

root\_y = find(y)

if root\_x != root\_y:

parent[root\_y] = root\_x

while True:

try:

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

parent = list(range(n + 1))

for \_ in range(m):

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

if find(a) == find(b):

print('Yes')

else:

print('No')

union(a, b)

unique\_parents = set(find(x) for x in range(1, n + 1)) # 获取不同集合的根节点

ans = sorted(unique\_parents) # 输出有冰阔落的杯子编号

print(len(ans))

print(\*ans)

except EOFError:

break

（OOP——类的写法）

class Cola():

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

self.value=val

self.parent=self

def find(self):

if self.parent==self:

return self

else:

self.parent=self.parent.find()

return self.parent

def union(self,other):

root\_self = self.find() # 找到当前对象所在集合的根节点

root\_other = other.find() # 找到其他对象所在集合的根节点

if root\_self != root\_other: # 如果两个对象不属于同一个集合

root\_other.parent = root\_self

print("No")

else:

print('Yes')

while True:

try:

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

colas=[0]+[Cola(i) for i in range(1,n+1)]

for \_ in range(m):

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

colas[x].union(colas[y])

counter=set()

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

counter.add(colas[i].find().value)

counter=list(counter)

counter.sort()

counter=list(map(str,counter))

print(len(counter))

print(' '.join(counter))

except EOFError:

break

图及其算法：

Dijkstra：

import heapq

def dijkstra(N, G, start):

INF = float('inf')

dist = [INF] \* (N + 1) # 存储源点到各个节点的最短距离

dist[start] = 0 # 源点到自身的距离为0

pq = [(0, start)] # 使用优先队列，存储节点的最短距离

while pq:

d, node = heapq.heappop(pq) # 弹出当前最短距离的节点

if d > dist[node]: # 如果该节点已经被更新过了，则跳过

continue

for neighbor, weight in G[node]: # 遍历当前节点的所有邻居节点

new\_dist = dist[node] + weight # 计算经当前节点到达邻居节点的距离

if new\_dist < dist[neighbor]: # 如果新距离小于已知最短距离，则更新最短距离

dist[neighbor] = new\_dist

heapq.heappush(pq, (new\_dist, neighbor)) # 将邻居节点加入优先队列

return dist

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

G = [[] for \_ in range(N + 1)] # 图的邻接表表示

for \_ in range(M):

s, e, w = map(int, input().split())

G[s].append((e, w))

start\_node = 1 # 源点

shortest\_distances = dijkstra(N, G, start\_node) # 计算源点到各个节点的最短距离

print(shortest\_distances[-1]) #

Prim（最小生成树）：

#兔子与樱花（顺便复习路径的字典表示，图的邻接表表示）

import heapq

def prim(graph):

# 初始化最小生成树的顶点集合和边集合

mst = set()

edges = []

visited = set()

total\_weight = 0

# 随机选择一个起始顶点

start\_vertex = list(graph.keys())[0]

# 将起始顶点加入最小生成树的顶点集合中

mst.add(start\_vertex)

visited.add(start\_vertex)

# 将起始顶点的所有边加入边集合中

for neighbor, weight in graph[start\_vertex]:

heapq.heappush(edges, (weight, start\_vertex, neighbor))

# 循环直到所有顶点都加入最小生成树为止

while len(mst) < len(graph):

# 从边集合中选取权重最小的边

weight, u, v = heapq.heappop(edges)

# 如果边的目标顶点已经在最小生成树中，则跳过

if v in visited:

continue

# 将目标顶点加入最小生成树的顶点集合中

mst.add(v)

visited.add(v)

total\_weight += weight

# 将目标顶点的所有边加入边集合中

for neighbor, weight in graph[v]:

if neighbor not in visited:

heapq.heappush(edges, (weight, v, neighbor))

return total\_weight

n = int(input())

graph = {}

for \_ in range(n - 1):

alist = list(input().split())

if alist[0] not in graph.keys():

graph[alist[0]] = []

for i in range(1, int(alist[1]) + 1):

if alist[2 \* i] not in graph.keys():

graph[alist[2 \* i]] = []

graph[alist[0]].append((alist[2 \* i], int(alist[2 \* i + 1])))

graph[alist[2 \* i]].append((alist[0], int(alist[2 \* i + 1])))

print(prim(graph))

拓扑排序：

from collections import deque, defaultdict

#实际应用中可能需要import heapq

def topological\_sort(graph):

indegree = defaultdict(int)

result = []

queue = deque()

# 计算每个顶点的入度

for u in graph:

for v in graph[u]:

indegree[v] += 1

# 将入度为 0 的顶点加入队列

for u in graph:

if indegree[u] == 0:

queue.append(u)

# 执行拓扑排序

while queue:

u = queue.popleft()

result.append(u)

for v in graph[u]:

indegree[v] -= 1

if indegree[v] == 0:

queue.append(v)

# 检查是否存在环

if len(result) == len(graph):

return result

else:

return None

单调栈：

n=int(input())

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

stack=[]

for i in range(len(lst)):

while stack and lst[stack[-1]]<lst[i]:

lst[stack.pop()]=str(i+1)

stack.append(i)

while stack:

lst[stack.pop()]='0'

print(' '.join(lst))#实际操作中，可以再单独开一个数组存放结果，以避免数据覆盖

字典树（dict dict型）：

class TrieNode:

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

self.children = {}

self.is\_end\_of\_word = False

class Trie:

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

self.root = TrieNode()

def insert(self, word):

node = self.root

for char in word:

if char not in node.children:

node.children[char] = TrieNode()

node = node.children[char]

node.is\_end\_of\_word = True

def search(self, word):

node = self.root

for char in word:

if char not in node.children:

return False

node = node.children[char]

return node.is\_end\_of\_word

def starts\_with(self, prefix):

node = self.root

for char in prefix:

if char not in node.children:

return False

node = node.children[char]

return True