并查集（按秩合并+路径压缩）

class UnionFind:

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

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, x):

if self.parent[x] != x:

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

return self.parent[x]

def union(self, x, y):

px, py = self.find(x), self.find(y)

if self.rank[px] > self.rank[py]:

self.parent[py] = px

else:

self.parent[px] = py

if self.rank[px] == self.rank[py]:

self.rank[py] += 1

def find(x):

if parent[x] != x: # 如果不是根结点，继续循环

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\_x] = root\_y

size[root\_y] += size[root\_x]

判断无向图是否连通有无回路

dfs能直接判断是否连通，过程中记录父亲节点，只要当前节点能够去到一个已经遍历过的节点，并且这个节点不是父亲节点，那么必然成环，以及连通和成环是可以同时判断的

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

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

for \_ in range(m):

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

edge[a].append(b)

edge[b].append(a)

cnt, flag = set(), False

def dfs(x, y):

global cnt, flag

cnt.add(x)

for i in edge[x]:

if i not in cnt:

dfs(i, x)

elif y != i:

flag = True

for i in range(n):

cnt.clear()

dfs(i, -1)

if len(cnt) == n:

break

if flag:

break

print("connected:"+("yes" if len(cnt) == n else "no"))

print("loop:"+("yes" if flag else 'no'))

中序表达式转后序表达式(通用模板)

def infixToPostfix(infixexpr):

prec = {}

prec["\*"] = 3

prec["/"] = 3

prec["+"] = 2

prec["-"] = 2

prec["("] = 1

opStack = [] # Stack()

postfixList = []

tokenList = infixexpr.split()

for token in tokenList:

if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":

postfixList.append(token)

elif token == '(':

#opStack.push(token)

opStack.append(token)

elif token == ')':

topToken = opStack.pop()

while topToken != '(':

postfixList.append(topToken)

topToken = opStack.pop()#这里可以把左括号pop出去

else:

#while (not opStack.is\_empty()) and (prec[opStack.peek()] >= prec[token]):

while opStack and (prec[opStack[-1]] >= prec[token]):

postfixList.append(opStack.pop())

#opStack.push(token)

opStack.append(token)

#while not opStack.is\_empty():

while opStack:

postfixList.append(opStack.pop())

return " ".join(postfixList)

print(infixToPostfix("( A + B ) \* C - ( D - E ) \* ( F + G )"))

后序求值

def evaluate\_postfix(expression):

stack = []

tokens = expression.split()

for token in tokens:

if token in '+-\*/':

# 弹出栈顶的两个元素

right\_operand = stack.pop()

left\_operand = stack.pop()

# 执行运算

if token == '+':

stack.append(left\_operand + right\_operand)

elif token == '-':

stack.append(left\_operand - right\_operand)

elif token == '\*':

stack.append(left\_operand \* right\_operand)

elif token == '/':

stack.append(left\_operand / right\_operand)

else:

# 将操作数转换为浮点数后入栈

stack.append(float(token))

# 栈顶元素就是表达式的结果

return stack[0]

# 读取输入行数

n = int(input())

# 对每个后序表达式求值

for \_ in range(n):

expression = input()

result = evaluate\_postfix(expression)

# 输出结果，保留两位小数

print(f"{result:.2f}")

中后序建树

"""

后序遍历的最后一个元素是树的根节点。然后，在中序遍历序列中，根节点将左右子树分开。

可以通过这种方法找到左右子树的中序遍历序列。然后，使用递归地处理左右子树来构建整个树。

"""

def build\_tree(inorder, postorder):

if not inorder or not postorder:

return []

root\_val = postorder[-1]

root\_index = inorder.index(root\_val)

left\_inorder = inorder[:root\_index]

right\_inorder = inorder[root\_index + 1:]

left\_postorder = postorder[:len(left\_inorder)]

right\_postorder = postorder[len(left\_inorder):-1]

root = [root\_val]

root.extend(build\_tree(left\_inorder, left\_postorder))

root.extend(build\_tree(right\_inorder, right\_postorder))

return root

def main():

inorder = input().strip()

postorder = input().strip()

preorder = build\_tree(inorder, postorder)

print(''.join(preorder))

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

main()

多行输入：

while True:

try:

except EOFError:

break

最短距离（dij）

import heapq

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

edges = {}

for i in range(n):

edges[i] = []

for \_ in range(m):

u, v, w = map(int, input().split())

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

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

def dijstra():

pq = [(0, s)]

dists = [float("inf")] \* n

visited = set()

while pq:

dist, node = heapq.heappop(pq)

visited.add(node)

if node == t:

return dist

for nbr, weight in edges[node]:

if nbr not in visited:

new\_dist = dist + weight

if new\_dist < dists[nbr]:

dists[nbr] = new\_dist

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

return -1

print(dijstra())

最短距离（spfa）

from collections import deque

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

edges = {}

for i in range(n):

edges[i] = []

for \_ in range(m):

u, v, w = map(int, input().split())

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

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

def spfa(): # dij算法去掉visited空间，并把堆换成队列即可

q = deque()

q.append((0, s))

dists = [float("inf")] \* n

while q:

dist, node = q.popleft()

for nbr, weight in edges[node]:

new\_dist = dist + weight

if new\_dist < dists[nbr]:

dists[nbr] = new\_dist

q.append((new\_dist, nbr))

return dists[t]

distance = spfa()

print(distance if distance != float("inf") else -1)

最小生成树（Kruskal）（最小堆+并查集，搜边）

class UnionFind:

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

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, x):

if self.parent[x] != x:

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

return self.parent[x]

def union(self, x, y):

px, py = self.find(x), self.find(y)

if self.rank[px] > self.rank[py]:

self.parent[py] = px

else:

self.parent[px] = py

if self.rank[px] == self.rank[py]:

self.rank[py] += 1

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

edges = []

for \_ in range(m):

u, v, w = map(int, input().split())

edges.append((v, u, w)) # 由于此算法特殊，无向图也只需存一个方向的边就行了

def kruskal():

uf = UnionFind(n)

edges.sort(key = lambda x: x[2])

res = 0

for u, v, w in edges:

if uf.find(u) != uf.find(v):

uf.union(u, v)

res += w

if len(set(uf.find(i) for i in range(n))) > 1:

return -1

return res

print(kruskal())

最小生成树（Prim）（堆，搜点）

import heapq

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

edges = {}

for i in range(n):

edges[i] = []

for \_ in range(m):

u, v, w = map(int, input().split())

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

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

def Prim():

visited = [False] \* n

result\_cost = 0

pq = [(0, 0)]

while pq:

weight, vertex = heapq.heappop(pq)

if visited[vertex]:

continue

visited[vertex] = True

result\_cost += weight

for i, w in edges[vertex]:

if visited[i]:

continue

heapq.heappush(pq, (w, i))

return result\_cost if all(visited) else -1

print(Prim())

拓扑排序（dfs）

n = int(input())

edges = {}

for i in range(n):

edges[i] = []

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

for j in line:

if j != 0:

edges[i].append(j-1) # 因为题目中的人编号从1开始，而字典的键是从0开始的，所以减1

visited = [False] \* n

pre = {}

post = {}

clock = 0

def dfs(i):

global clock

clock += 1

pre[i] = clock

for j in edges[i]:

if not visited[j]:

visited[j] = True

dfs(j)

clock += 1

post[i] = clock

return

# 处理每一个节点，防止有孤立的节点

for i in range(n):

if not visited[i]:

visited[i] = True

dfs(i)

result = sorted(post.keys(), key=lambda x: post[x], reverse=True)

result = [x+1 for x in result] # 恢复为题目中的编号从1开始

print(' '.join(map(str, result)))