课程概述和导论

重载内置函数

__??__

le <=　　lt <　gt > gt >=

contains in　　getitem []

大O计时

得到函数运行时间（使用timeit模块）

```
#要先建立Timer对象，参数1要为之计时的语句，参数2为被试语句
t1 = Timer('test1()','from __main__ import test1')
print(t1.timeit(number=1000)) #重复执行1000次所用秒数
```

面对对象编程：定义类class

注意是双下划线！！！

```
class Fraction:
  def __init__(self,a,b):
      self.x = a
      self.y = b
  def show(self): #定义方法
      print(self.x,'/',self.y)
  def __add__(self,another): #重写默认的函数
        new_x = self.x*another.y+self.y*another.x
        new_y = self.y*another.y
        return Fraction(new_x,new_y)
```

继承inheritance

一些基础操作?

取质数

埃氏筛

```
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
```

欧拉筛

```
def euler(r):
    prime = [0 for i in range(r+1)]
    common = []
    for i in range(2, r+1):
        if prime[i] == 0:
            common.append(i)
        for j in common:
            if i*j > r:
                break
            prime[i*j] = 1
            if i % j == 0:
                break
    return prime
```

最大公因数GCD

欧几里得算法：对于整数m和n，如果m能被n整除，那么它们的最大公因数就是n。然而，如果m不能被n整除，那么结果是n与m除以n的余数的最大公因数

```
def gcd(m,n):
    while m%n != 0:
        oldm,oldn = m,n
        m,n = oldn,oldm%oldn
    return n
```

其他

str

```
str.upper()   str.lower()   str.title()
str.rstrip()  str.lstrip()  str.strip()
```

eval()将字段串表达的函数输出（例如'1+2'输出3）

lru_cache

```
from functools import lru_cache
@lru_cache(maxsize=128)#tle了加大，mle了减小
def func():
```

扩栈莫名其妙都可以用，不局限于RE

```
import sys
sys.setrecursionlimit(1<<30)
```

sys.exit()退出程序

format

取小数'{:.nf}'.format(num)

格式化输出

```
print("Name: {}, Age: {}".format(name, age))#或者
print(f"Name: {name}, Age: {age}")
```

%格式化输出

进制转换

1.利用内置函数

除10进制，其他进制转换**带有前缀**：

2进制--0b

8进制--0o

16进制--0x

注：int(str,base=10)将字符串视为base进制输入转化为10进制的数字

2.格式化数字操作

'{:b/o/x}'.format(num)

此种方法**不带前缀**

正则表达式

符号及解释（大写为小写相反意思）

\d任意数字　\w字母数字下划线 \b单词边界　\n换行　\r回车　\t制表符 \s所有空白字符（\n,\r,\t,\f换页）

\A字符串开始，\z字符串结束　^匹配字符串开始 $匹配字符串结束　.匹配除\n\r外所有单字符

[...]在...范围内字符 [^..]不在该范围内字符 {...}匹配次数。n-n次；n,m-n到m次；n,n次及以上

*匹配0到多次；+匹配1到多次；？匹配0到1次

(pattern)对匹配分组并记住匹配的文本（用group分别得到） \1...\9匹配第n个分组的内容

在函数参数加上**re.M**多行匹配

re库用法

pattern字符串格式，r'...'，r防止转义

re.match从起始位置匹配模式，匹配失败返回None

re.group(num=0)对于match结果，返回匹配到项目的组（若num有多个返回元组

re.groups返回匹配的元组

```
import re
n = 'Her name is Alice.'
x = re.match(r'([A-Z]?.*) name is ([A-Z]?.*).', n)
print(x.group())
print(x.group(1))
print(x.group(2))
'''
Her name is Alice.
Her
Alice
'''
```

re.search扫描整个字符串并返回第一个成功的匹配，否则返回None

re.search().span()返回匹配的索引范围

.....start()/end()返回匹配开始或者结束的位置

re.sub(pattern, repl, string, count=0)替换匹配项，count默认0替换所有匹配

repl可以是某字符，或者函数（注意函数的输入应先用match.group()处理

```python
import re
def xiaoxie(matched):
    st = matched.group()
    return st.lower()
n = 'Her name is Alice.'
print(re.sub(r'\b[A-Z]{1}[a-z]*\b', xiaoxie, n))
print(re.sub(r'\b[A-Z]{1}[a-z]*\b', '??', n))
'''
her name is alice.
?? name is ??.
'''
```

pattern.findall(string,[pos],[endpos])找到所有匹配子串返回列表，若无匹配返回空列表（不包括endpos）

此处pattern应该用pattern = **re.compile(r'')**生成

pattern.finditer类似于上者，但是返回迭代器

pattern.split按照pattern分割字符串返回列表

常见正则表达式

汉字[\u4e00-\u9fa5]{0,}（莫名奇妙返回了很多空字符串?

math库

```python
import math
math.pi    math.e
math.sqrt(isqrt带取整)    math.pow(x,y)=x**y    math.log(x,base)
math.floor()    math.ceil()    math.factorial()#阶乘
math.gcd(*integers)#最大公约数
math.lcm(*integers)#最小公倍数
math.comb(n,k)#Cnk组合数
math.dist(p,q)#p,q为两个大小相同数组，计算欧氏距离
math.prod(iterable,start)#计算iterable所有数乘积在start上
math.modf(x)#返回（小数部分，整数部分）
```

ASCII码

ord()字符变码 **chr()**码变字符

计数--Counter

```python
from collections import Counter
Counter(item)  #输出Counter({数字:次数,})
Counter(item).most_common(n)  #前n个最常见的数，输出[(数,次数),]

x,y = Counter(item_a),Counter(item_b)
x.update(y)  #将x,y统计结果整合到一起
x.subtract(y)  #减去y的统计结果
```

算法

二分查找

代码：

```
low,high = 0,len(list)-1
answer = None
target = input()  #目标
while low <= high:
    mid = (low+high)/2  #向下取整
    guess = list[mid]
    if guess == target:
        answer = mid
        break
    if guess < target:
        low = mid+1
    else:
        high = mid-1
```

bisect库

bisect.bisect(list,num)返回num按序插入在list中的索引

bisect.insert(pos,num)在list中的pos位置插入num

以上2个可以用**bisect.insort(list,num)**合并完成

bisect.bisect/insort_left(list,num)当有相同大小元素时返回左边索引或者插入到左边（右边同理）

注意bisect--O(logn),insort--O(n)

DFS

可以用exit()结束函数，而避免多重return True

BFS

实现图--散列表，可以使用defaultdict(list)实现

思路：按照每层顺序向队列加入搜索对象

代码：（可以用于无向图）

```
from collections import deque
graph = {父节点：{子节点:权重}}  #散列表存储节点及其邻居
seach_queue = deque()  #记录已经搜索过的节点，否则可能无限循环
def bfs():
    searched = set()
    while search_queue:
        get = search_queue.popleft()
        if get in searched():
```

```
            continue
        if get == answer:
            return get
        search_queue += graph[get]  #加入下一层
        searched.add(get)
    return False
```

```

```

狄克斯特拉算法--加权图

思路：对已有的最小开销节点更新其邻居节点最小开销，直至更新完所有节点（不能对于无向图或者有环存在或者有负权边的图使用）

更新最小节点可以用heapq

代码：

```
import heapq
costs = [];heapq.heapify(costs)  #存为[(cost,node),]
graph = {父节点：{子节点:权重}}  #散列表存储节点及其邻居或者在计算costs过程中直接对父节点到子节点计算权重（爬山路）
#parent = {节点:父节点}，存储每个节点的父节点，如果不需要回溯可以不用建立该变量
processed = set()  #存储已经处理过的节点

small = heapq.heappop(costs)  #得到最小cost的节点
while small[1] != stop:  #当最小代价是终点时停止搜索
    s_cost,node = small[0],small[1]
    neighbors = graph[node]  #关于其邻居节点
    for neighbor in neighbors:
        if neighbor in processed:
            continue
        cost = costs[node] + graph[node][neighbor]
        heapq.heappush(costs,(cost,neighbor))  #不需要管是否新方法是比原邻居cost小还是大，因为heap会排序先给出小的。需要回溯时还是要判断一下
    processed.append(node)  #处理完该节点的所有子节点
    small = heapq.heappop(costs)  #通过cost找出当前最小开销节点
answer = small[0]
```

动态规划--背包问题

0-1背包

```
#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

```
#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

```
#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
```

线性数据结构

栈、队列、双端队列、列表--有序（顺序取决于放入先后相对位置保持不变）

栈

‘下推栈’，LIFO后进先出

```
#自己建立Stack对象
Stack() #创建空栈
push(item) #添加元素到顶端
pop() #移出顶端
peek() #返回顶端元素但不移除
isEmpty() #检查栈是否为空
size() #返回栈中元素数目
```

单调栈

例：输入 1 4 2 3 5

输出：大于某数字的最小下标否则输出0（2 5 4 5 0）

```
n = int(input())
a = list(map(int, input().split()))
stack = []

#f = [0]*n
for i in range(n):
    while stack and a[stack[-1]] < a[i]:
        #f[stack.pop()] = i + 1
        a[stack.pop()] = i + 1


    stack.append(i)

while stack:
    a[stack[-1]] = 0
    stack.pop()

print(*a)
```

前、中、后序表达式

队列

添加发生在尾部，移除发生在头部。FIFO先进先出

```
#自己建立Queue对象
Queue() #建立空队列
enqueue(item) #在尾部添加元素
dequeue() #从头部移出并返回一个元素
isEmpty() #检查队列是否为空
size() #返回队列中元素数目
```

双端队列deque

```
from collections import deque
d = deque(item)
d.append(x)    d.appendleft(x)
d.pop(x)    d.popleft(x)
reverse(d)
```

应用--约瑟夫问题

搜索和排序

映射

```
def __getitem__(self,key): #使映射通过[]索引
    return self.get(key)
def __setitem__(self,key,data): #使映射通过[]设置值
    self.put(key,data)
```

排序

冒泡排序

遍历依次交换前后大小（可以直接用a,b = b,a）

对于n个元素：遍历n-1遍（次数i只遍历到n-i+1即可，此时第n-i+1位已经为最大/小）

选择排序

每次遍历找到最大值并放在正确位置上，遍历n-1轮

插入排序

对每个数字向前查找并不停交换直至前面数字比它小为止

希尔排序（递减增量排序）

选取步长i构成子列表排序

然后再插入排序

归并排序

分治策略

关于两个排好序子序列的合并--**双指针**

```
i,j,k = 0
alist = []
while i < len(lefthalf) and j < len(reighthalf):
    if lefthalf[i] < righthalf[j]:
        alist[k] = lefthalf[i]
        i += 1
    else:
        alist[k] = righthalf[j]
        j += 1
       k += 1
while i < len(lefthalf):
```

```
        alist[k] = lefthalf[i]
        i += 1
        k += 1
    while j <len(righthalf):
        alist[k] = righthalf[j]
        j += 1
        k += 1
```

注意：该方法需要额外的空间存储分开成两部分的列表，当列表较大时可能出现问题

快速排序

分治策略

1）找到基准值的正确位置

2）对于基准值左右列表重复操作

当分割点偏向一端时可能增加时间复杂度--

三数取中法（考虑头、中、尾3个元素中偏中间的值）

树

树的遍历

前序遍历

访问根节点-递归前序遍历左子树-前序遍历右子树

```
def preorder(tree):
    if tree:
        print(tree.getRootVal)
        preorder(tree.getLeftChild)
        preorder(tree.getRightChild)
```

中序遍历

递归前序遍历左子树-访问根节点-递归前序遍历右子树

后序遍历

递归后序遍历右子树-递归后序遍历左子树-访问根节点

**不同的遍历方式仅区别于根节点的访问位置

中序表达式转后序表达式

OpenJudge - 24591:中序表达式转后序表达式

```
#赵语涵2300012254
op,comp = ['+','-','*','/','(',')'],{'(':1,'+':2,'-':2,'*':3,'/':3}
def change(x):
    results = []
    ind,ops,stop = 0,[],False
    while ind < len(x):
        num = ''
        while (a:=x[ind]) not in op:
            num += a
            ind += 1
            if ind==len(x):
                stop = True
                break
        if num != '':
            results.append(num)
        if stop:
            break
        if a == ')':
            while (b:=ops.pop()) != '(':
                results.append(b)
        elif a == '(':
            ops.append(a)
```

```
        else:
            while True:
                if ops == []:
                    break
                if comp[(b:=ops.pop())]>=comp[a]:
                    results.append(b)
                else:
                    ops.append(b)
                    break
            ops.append(a)
        ind += 1
    while ops:
        results.append(ops.pop())
    return ' '.join(results)
n = int(input())
for i in range(n):
    print(change(input()))
```

后序表达式求值

```
#赵语涵2300012254
def ope(a,b,o):
    if o=='+':
        return a+b
    elif o=='-':
        return a-b
    elif o=='*':
        return a*b
    elif o=='/':
        return a/b

def calcu(data):
    ind,nums = 0,[]
    while ind < len(data):
        try:
            nums.append(float(data[ind]))
        except:
            b,a = nums.pop(),nums.pop()
            nums.append(ope(a,b,data[ind]))
        ind += 1
    return nums
n = int(input())
for i in range(n):
    x = calcu(list(input().split()))[0]
    print('%.2f'%x)
```

二叉树转换（左儿子右兄弟）

```python
from collections import defaultdict
n=int(input())
nodes=[(x,int(i)) for x,i in input().split()]
output=defaultdict(list)
t=0
last=0
ma=0
for x,i in nodes:
    #print(last,x,i)
    if last == 1:
        t-=1
    else:
        t+=1
    if x != '$':
        output[t].append(x)
        ma=max(ma,t)
    last=i
print(' '.join([' '.join(output[i][::-1])for i in range(1,ma+1)]))
```

二叉堆实现优先级队列

完全树，可以直接用列表表示（索引从1开始，对位置p的元素，其左节点在2p，右节点在2p+1）

```python
def percUp(self,i):
    while i//2 > 0:
        if self.heapList[i] < self.heapList[i//2]:
            tmp = self.heapList[i//2]
            self.heapList[i//2] = self.heapList[i]
            self.heapList[i] = tmp
        i = i//2
```

二叉搜索树

1）右旋/左旋

2）右旋或左旋时若子节点右/左倾，先对子节点左/右倾

并查集

发现它，抓住它

```python
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):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1
```

Huffmann编码树

OpenJudge - 22161:哈夫曼编码树

```
import heapq
class Node():
    def __init__(self,char,freq):
        self.key=char
        self.freq=freq
        self.left,self.right=None,None
    def __lt__(self,other):
        return self.freq < other.freq

def build_huff():#构建哈夫曼树
    n=int(input())
    chars=[]#存贮字母节点的最小堆
    for _ in range(n):
        char,freq=input().split()#各字母及使用频率
        chars.append(Node(char,int(freq)))
    heapq.heapify(chars)
    while len(chars) > 1:
        a,b=heapq.heappop(chars),heapq.heappop(chars)#合并频率最低的两个字母
        parent=Node(a.key+b.key,a.freq+b.freq)
        parent.left,parent.right=a,b
        heapq.heappush(chars,parent)
    return chars[0]
```

图

拓扑排序

Kahn算法

Kahn算法的基本思想是通过不断地移除图中的入度为0的顶点，并将其添加到拓扑排序的结果中，直到图中所有的顶点都被移除。具体步骤如下：

1. 初始化一个队列，用于存储当前入度为0的顶点。
2. **遍历图中的所有顶点，计算每个顶点的入度，并将入度为0的顶点加入到队列中。**
3. **不断地从队列中弹出顶点，并将其加入到拓扑排序的结果中。同时，遍历该顶点的邻居，并将其入度减1。如果某个邻居的入度减为0，则将其加入到队列中。**
4. 重复步骤3，直到队列为空。

Kahn算法的时间复杂度为O(V + E)，其中V是顶点数，E是边数。它是一种简单而高效的拓扑排序算法，在有向无环图（DAG）中广泛应用。

拓扑排序

题目：给出一个图的结构，输出其拓扑排序序列，要求在同等条件下，编号小的顶点在前。

题解中graph是邻接表，形如graph[1]=[2,3,4]，由于本题要求顺序，因此不用队列而用优先队列。

```
from collections import defaultdict
from heapq import heappush,heappop
def Kahn(graph):
    q,ans=[],[]
    in_degree=defaultdict(int)
    for lst in graph.values():
        for vert in lst:
            in_degree[vert]+=1

    for vert in graph.keys():
        if vert not in in_degree or in_degree[vert]==0:
            heappush(q,vert)

    while q:
        vertex=heappop(q)
        ans.append('v'+str(vertex))
        for neighbor in graph[vertex]:
            in_degree[neighbor]-=1
            if in_degree[neighbor]==0:
                heappush(q,neighbor)
    return ans

v,a=map(int,input().split())
graph={}
```

```
for _ in range(a):
    f,t=map(int,input().split())
    if f not in graph:graph[f]=[]
    if t not in graph:graph[t]=[]
    graph[f].append(t)

for i in range(1,v+1):
    if i not in graph:graph[i]=[]

res=Kahn(graph)
print(*res)
```

最小生成图

Prim算法

步骤：

1. 起点入堆。
2. 堆顶元素出堆（排序依据是到该元素的开销），如已访问过，continue；否则标记为visited。
3. 访问该节点相邻节点，（访问开销（排序依据），相邻节点）入堆。
4. 相邻节点前驱设置为当前节点（如需）。
5. 当前节点入树

全部精要在于：每次走出下一步的开销都是当前最小的。

Agri-net

题目：用邻接矩阵给出图，求最小生成树路径权值和。

```
4
0 4 9 21
4 0 8 17
9 8 0 16
21 17 16 0
        # 注意这一步continue很关键，因为一个节点会同时很多存在于pq中（这是由出队标记决定的）
        # 如果不设计这一步continue，则会重复加路径长。
```

```
from heapq import heappop, heappush
def prim(matrix):
    ans=0
    pq,visited=[(0,0)],[False for _ in range(N)]
    while pq:
        c,cur=heappop(pq)
        if visited[cur]:continue
        visited[cur]=True
        ans+=c
        for i in range(N):
            if not visited[i] and matrix[cur][i]!=0:
                heappush(pq,(matrix[cur][i],i))
```

```
    return ans

while True:
    try:
        N=int(input())
        matrix=[list(map(int,input().split())) for _ in range(N)]
        print(prim(matrix))
    except:break
```

Kruskal算法（能写Prim建议写Prim）

Agri-net

```
class DisJointSet:
    def __init__(self,num_vertices):
        self.parent=list(range(num_vertices))
        self.rank=[0 for _ in range(num_vertices)]

    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):
        root_x=self.find(x)
        root_y=self.find(y)
        if root_x!=root_y:
            if self.rank[root_x]<self.rank[root_y]:
                self.parent[root_x]=root_y
            elif self.rank[root_x]>self.rank[root_y]:
                self.parent[root_y]=root_x
            else:
                self.parent[root_x]=root_y
                self.rank[root_y]+=1

# graph是邻接表
def kruskal(graph:list):
    res,edges,dsj=[],[],DisJointSet(len(graph))
    for i in range(len(graph)):
        for j in range(i+1,len(graph)):
            if graph[i][j]!=0:
                edges.append((i,j,graph[i][j]))

    for i in sorted(edges,key=lambda x:x[2]):
        u,v,weight=i
        if dsj.find(u)!=dsj.find(v):
            dsj.union(u,v)
            res.append((u,v,weight))
    return res

while True:
```

```
    try:
        n=int(input())
        graph=[list(map(int,input().split())) for _ in range(n)]
        res=kruskal(graph)
        print(sum(i[2] for i in res))
    except EOFError:break
```

Kosaraju's算法（有向图连通域）

用于查找有向图中强连通分量（任意两个节点都可到达的一组节点）

```
def dfs1(graph, node, visited, stack):# 第一个深度优先搜索函数，用于遍历图并将节点按完成时间压入栈中
  visited[node] = True # 标记当前节点为已访问
  for neighbor in graph[node]: # 遍历当前节点的邻居节点
        if not visited[neighbor]: # 如果邻居节点未被访问过
      dfs1(graph, neighbor, visited, stack) # 递归调用深度优先搜索函数
  stack.append(node) # 将当前节点压入栈中，记录完成时间

def dfs2(graph, node, visited, component):# 第二个深度优先搜索函数，用于在转置后的图上查找强连通分量
  visited[node] = True # 标记当前节点为已访问
  component.append(node) # 将当前节点添加到当前强连通分量中
  for neighbor in graph[node]: # 遍历当前节点的邻居节点
    if not visited[neighbor]: # 如果邻居节点未被访问过
      dfs2(graph, neighbor, visited, component) # 递归调用深度优先搜索函数
def kosaraju(graph):# Kosaraju's 算法函数
  # Step 1: 执行第一次深度优先搜索以获取完成时间
  stack = [] # 用于存储节点的栈
  visited = [False] * len(graph) # 记录节点是否被访问过的列表
  for node in range(len(graph)): # 遍历所有节点
    if not visited[node]: # 如果节点未被访问过
      dfs1(graph, node, visited, stack) # 调用第一个深度优先搜索函数

    # Step 2: 转置图
  transposed_graph = [[] for _ in range(len(graph))] # 创建一个转置后的图
    for node in range(len(graph)): # 遍历原图中的所有节点
      for neighbor in graph[node]: # 遍历每个节点的邻居节点
        transposed_graph[neighbor].append(node) # 将原图中的边反向添加到转置图中

    # Step 3: 在转置后的图上执行第二次深度优先搜索以找到强连通分量
  visited = [False] * len(graph) # 重新初始化节点是否被访问过的列表
  sccs = [] # 存储强连通分量的列表
  while stack: # 当栈不为空时循环
    node = stack.pop() # 从栈中弹出一个节点
    if not visited[node]: # 如果节点未被访问过
      scc = [] # 创建一个新的强连通分量列表
      dfs2(transposed_graph, node, visited, scc) # 在转置图上执行深度优先搜索
      sccs.append(scc) # 将找到的强连通分量添加到结果列表中
  return sccs # 返回所有强连通分量的列表
```

另：判断无向图是否连通有无回路

思路：比较简单的并查集方法，直接将祖先中大的应该指向小的，如果在过程中有遇到某边的两个连接点是指向同一祖先的说明出现了连通情况loop=yes，最后统计根节点的数量如果只有1个根节点说明只有一个回路connected=yes

代码

```
#赵语涵2300012254
n,m = map(int,input().split())
parent = [x for x in range(n)]
def find(x):
    if parent[x] != x:
        return find(parent[x])
    return parent[x]
loop = 'no'
for _ in range(m):
    a,b = map(int,input().split())
    x,y = find(a),find(b)
    if x == y:
        loop = 'yes'
    else:
        if x < y:
            parent[y] = x
        else:
            parent[x] = y
ancient = set(find(x) for x in range(n))
if len(ancient)==1:
    print('connected:yes')
else:
    print('connected:no')
print(f'loop:{loop}')
```