Cheating Paper！！！　张曦月

一、树相关模板，常见题型，思路，写法整理

1、树的基本操作（深度，子叶数目，前中后建树及输出，二叉搜索树，）

```
# class Treenode建树，求树的深度
class Treenode():
    def __init__(self,num):
        self.num=num
        self.left=None
        self.right=None

def find_height(x,cnt=0): #找深度
    if x==None:
        return 0
    lc=x.left
    rc=x.right
    return max(find_height(lc),find_height(rc))+1

n=int(input())
node=[Treenode(i) for i in range(n+1)] #该树1～n 根规定为1
for i in range(1,n+1):
    l,r=map(int,input().split()) #建树
    if l!=-1:
        node[i].left=node[l]
    if r!=-1:
        node[i].right=node[r]
print(find_height(node[1]))
```

```
#Treenode建树，求子叶数目，树的高度。
class Treenode():
    def __init__(self):
        self.left=None
        self.right=None

def find_height(node):#求树的高度
    if node==None:
        return 0
    return max(find_height(node.left),find_height(node.right))+1

def leave_num(node):#求子叶数目
    if node==None: #! !
        return 0
    if node.left==None and node.right==None: #! !
        return 1
    return leave_num(node.left)+leave_num(node.right)

n=int(input())
```

```
node=[Treenode() for i in range(n)]
has_parent=[False for i in range(n)]
for i in range(n):
    l,r=map(int,input().split())
    if l!=-1:
        node[i].left=node[l]
        has_parent[l]=True #! !
    if r!=-1:
        node[i].right=node[r]
        has_parent[r]=True #! !
root=has_parent.index(False)
print(f"{find_height(node[root])-1} {leave_num(node[root])}")
```

```
#中后转前
def iptp(ino,poo):
    if len(poo)==1:
        return [poo[0]]
    if len(poo)==0:
        return []
    root=poo[-1]
    x=ino.index(root)
    left_ino=ino[:x]
    right_ino=ino[x+1:]
    left_poo=poo[:len(left_ino)]
    right_poo=poo[len(left_ino):len(poo)-1]
    return [root]+iptp(left_ino,left_poo)+iptp(right_ino,right_poo)

in_order=list(input())
post_order=list(input())
print(''.join(iptp(in_order,post_order)))
```

```
def ptop(pre): ##二叉搜索树前序建树转后序
    if len(pre)==1:
        return [pre[0]]
    if len(pre)==0:
        return []
    root=pre[0]
    left=[i for i in pre if i<root]
    right=[i for i in pre if i>root]
    return ptop(left)+ptop(right)+[root]

n=int(input())
pre=list(map(int,input().split()))
print(' '.join(map(str,ptop(pre))))
```

```
##二叉搜索树的建立及层次遍历
class Treenode():
    def __init__(self,value):
        self.value=value
        self.left=None
```

```
        self.right=None

def insert(root,newvalue):
    if root==None:
        return Treenode(newvalue)
    elif newvalue<root.value:
        root.left=insert(root.left,newvalue)
    elif newvalue>root.value:
        root.right=insert(root.right,newvalue)
    return root

def level_order_traversal(root):  ##层次遍历就是用bfs!!
    queue=[root]
    traversal=[]
    while queue:
        node=queue.pop(0)
        traversal.append(node.value)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return traversal

l=list(map(int,input().split()))
l=list(dict.fromkeys(l))
root=None
for num in l:
    root=insert(root,num)
traversal=level_order_traversal(root)
print(' '.join(map(str, traversal)))
```

```
#现在请你将一些一般的树用这种方法转换为二叉树，并输出转换前和转换后树的高度    dudduduudu
class Treenode():
    def __init__(self,value):
        self.value=value
        self.child=[]

def find_height1(node): ## 求高度！！！！！！
    return 1+max([find_height1(nod) for nod in node.child],default=-1)

def  find_height2(node):
    return 1+max([find_height2(node.child[i])+i for i in
range(len(node.child))],default=-1)

s=input()
s=list(s)
id=0
root=Treenode(0)
stack=[root]
for i in s:
    if i=='d':
```

```
            node=Treenode(id)
            stack[-1].child.append(node)
            stack.append(node)
            id+=1
        else:
            stack.pop()
print(f'{find_height1(root)} => {find_height2(root)}')
```

2、树的变体应用（表达式树，哈夫曼编码树，AVL「平衡二叉树」，）

```
#表达式树的建树及前后序输出： A(B(E),C(F,G),D(H(I)))
#在栈 temp 中保存的是当前节点的父节点，每遇到一个右括号")"，就从栈中弹出一个父节点，并继续处理该父节点的其他子节点。
class Treenode():
    def __init__(self,value):
        self.child=[]
        self.value=value

def build_tree(s):
    temp=[]
    node=None
    root=None
    for x in s:
        if x.isalpha():
            node=Treenode(x)
            if temp:
                temp[-1].child.append(node)
        if x=="(":
            if node:
                temp.append(node)
                node=None
        if x==")":
            if temp:
                root=temp.pop()
    return node if root==None else root

def pre_order_print(node):
    ans=[node.value]
    for i in node.child:
        ans+=pre_order_print(i)
    return ans

def post_order_print(node):
    ans=[]
    for i in node.child:
        ans+=post_order_print(i)
    return ans+[node.value]

s=input()
s=list(s)
root=build_tree(s)
```

```
print(''.join(pre_order_print(root)))
print(''.join(post_order_print(root)))
```

```
#文件结构图，本质是表达式建树。
class node():
    def __init__(self,name):
        self.name=name
        self.dir=[]
        self.file=[]

def print_tree(nod,level=0):
    indent='|     '*level
    print(indent+nod.name)
    for dirs in nod.dir:
        print_tree(dirs,level+1)
    for files in sorted(nod.file):
        print(indent+files)

datas=[]
setnum=1
temp=[]
while True:
    line=input()
    if line=='#':
        break
    if line=='*':
        datas.append(temp)
        temp=[]
    else:
        temp.append(line)
for data in datas:
    print(f'DATA SET {setnum}:')
    root=node('ROOT')
    stack=[root]
    for x in data:
        if x[0]=="d":
            nod=node(x)
            stack[-1].dir.append(nod)
            stack.append(nod)
        if x[0]=="f":
            stack[-1].file.append(x)
        if x==']':
            stack.pop()
    print_tree(root)
    if setnum<len(datas):
        print()
    setnum+=1
```

```
#给定一个后序表达式，请转换成等价的队列表达式。例如，"3 4 + 6 5 * -"的等价队列表达式就是"5 6 4 3 * + -
"
class Treenode():
```

```
    def __init__(self,num):
        self.left=None
        self.right=None
        self.num=num

def build_tree(s): #still表达式建树
    stack=[]
    for i in s:
        node=Treenode(i)
        if i.isupper():
            node.right=stack.pop()
            node.left=stack.pop()
        stack.append(node)
    return stack[0]

def level_print(root): #层次遍历！！——宽搜！！
    queue=[root]
    ans=[]
    while queue:
        x=queue.pop(0)
        ans.append(x.num)
        if x.left!=None:
            queue.append(x.left)
        if x.right!=None:
            queue.append(x.right)
    return ans[::-1]

n=int(input())
out=[]
for i in range(n):
    s=input()
    s=list(s)
    root=build_tree(s)
    out.append(level_print(root))
for i in out:
    print(''.join(i))
```

```
##哈夫曼编码树
import heapq
class Node:
    def __init__(self, weight, char=None):
        self.weight = weight
        self.char = char
        self.left = None
        self.right = None
    def __lt__(self, other):
        if self.weight == other.weight:
            return self.char < other.char
        return self.weight < other.weight

def build_huffman_tree(characters):
```

```
    heap = []
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(left.weight + right.weight, min(left.char, right.char))
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]

def encode_huffman_tree(root):
    codes = {}
    def traverse(node, code):
        #if node.char:
        if node.left is None and node.right is None:
            codes[node.char] = code
        else:
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')
    traverse(root, '')
    return codes

def huffman_encoding(codes, string):
    encoded = ''
    for char in string:
        encoded += codes[char]
    return encoded

def huffman_decoding(root, encoded_string):
    decoded = ''
    node = root
    for bit in encoded_string:
        if bit == '0':
            node = node.left
        else:
            node = node.right
        if node.left is None and node.right is None:
            decoded += node.char
            node = root
    return decoded

n = int(input())
characters = {}
for _ in range(n):
    char, weight = input().split()
    characters[char] = int(weight)
huffman_tree = build_huffman_tree(characters)
codes = encode_huffman_tree(huffman_tree)
strings = []
while True:
```

```
    try:
        line = input()
        strings.append(line)
    except EOFError:
        break
results = []
for string in strings:
    if string[0] in ('0','1'):
        results.append(huffman_decoding(huffman_tree, string))
    else:
        results.append(huffman_encoding(codes, string))
for result in results:
    print(result)
```

```
#平衡二叉树的建立及先序输出
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def __init__(self):
        self.root = None
    def height(self, node):
        if node is None:
            return 0
        return node.height
    def balance_factor(self, node):
        if node is None:
            return 0
        return self.height(node.left) - self.height(node.right)
    def rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self.height(y.left), self.height(y.right))
        x.height = 1 + max(self.height(x.left), self.height(x.right))
        return x
    def rotate_left(self, x):
        y = x.right
        T2 = y.left
        y.left = x
        x.right = T2
        x.height = 1 + max(self.height(x.left), self.height(x.right))
        y.height = 1 + max(self.height(y.left), self.height(y.right))
        return y
    def insert(self, root, key):
        if root is None:
```

```
            return TreeNode(key)
        elif key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)
        root.height = 1 + max(self.height(root.left), self.height(root.right))
        balance = self.balance_factor(root)
        if balance > 1:
            if key < root.left.key:
                return self.rotate_right(root)
            else:
                root.left = self.rotate_left(root.left)
                return self.rotate_right(root)
        if balance < -1:
            if key > root.right.key:
                return self.rotate_left(root)
            else:
                root.right = self.rotate_right(root.right)
                return self.rotate_left(root)
        return root
    def pre_order_traversal(self, root):
        if root:
            return
[root.key]+self.pre_order_traversal(root.left)+self.pre_order_traversal(root.right)
        else:
            return []

if __name__ == "__main__":
    n = int(input())
    nums = list(map(int, input().split()))
    avl_tree = AVLTree()
    for num in nums:
        avl_tree.root = avl_tree.insert(avl_tree.root, num)
    ans=''
    for i in avl_tree.pre_order_traversal(avl_tree.root):
        ans+=str(i)+' '
    print(ans.strip())
```

```
from collections import defaultdict ##树的镜面映射
n=int(input())
l=list(input().split())
level=1
tree=defaultdict(list)
for i in l:
    a=i[0]
    x=int(i[1])
    if a!='$':
        tree[level].append(a)
    if x==1:
        level-=1
    else:
```

```
        level+=1

ans=''
for l in tree.values():
    ans+=' '.join(reversed(l))+' '
print(ans)
```

```
class Treenode(): #### 扩展二叉树（前序建！！）—— ABD..EF..G..C..
    def __init__(self,value):
        self.value=value
        self.left=None
        self.right=None

def build_tree(s):
    x=s.pop(0)
    if x=='.':
        return
    node=Treenode(x)
    node.left=build_tree(s)
    node.right=build_tree(s)
    return node

def order(node,pos): ## 0:前  1:中  2:后
    if node:
        sub=[order(nd,pos) for nd in (node.left,node.right)]
        sub.insert(pos,node.value)
        return ''.join(sub)
    return ''

pre=list(input())
root=build_tree(pre)
print(order(root,1))
print(order(root,2))
```

二、并查集

1、并查集的基本标程解析及例题演示（宗教信仰）

```
class DisjointSet():
    def __init__(self, n):
        self.parent = [i for i in range(n+1)] ##
        self.rank = [1] * (n+1) ##一定注意这里的下标！！！！

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
```

```
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1

def count(n):
    a=set()
    for i in range(1,n+1):
        a.add(unset.find(i)) ##看看有几个不连通的集（有几个根不一样的）
    a=list(a)
    return len(a)

cnt=0
ans=[]
while True:
    cnt+=1
    n,m=map(int,input().split())
    if n==0 and m==0:
        break
    unset=DisjointSet(n) ##建一个并查集
    for i in range(m):
        a,b=map(int,input().split())
        unset.union(a,b)  ##连接a b节点
    ans.append(f'Case {cnt}: {count(n)}')
for i in ans:
    print(i)
```

2、一些变体及补充

```
class DisjointSet():
    def __init__(self, n):
        self.parent = [i for i in range(n+1)]
        self.rank = [1] * (n+1)

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
      ##冰阔落——若原始编号为x 的阔落与原始编号为y的阔落已经在同一杯，请输出"Yes"；否则，我们将原始编号为y 所在杯子的所有阔落，          倒往原始编号为x 所在的杯子，并输出"No"。最后，老王想知道哪些杯子有冰阔落。
        root_u = self.find(u)
        root_v = self.find(v)
        self.parent[root_v] = root_u
```

```
class DisjointSet():
    def __init__(self, n):
        self.parent = [i for i in range(n)]##节点编号0～n-1
        self.rank = [1] * (n)
      #......
n,m=map(int,input().split())
unset=DisjointSet(n)
has_cycle=False ## 判断无向图是否连通有无回路
is_connected=True
for i in range(m):
    a,b=map(int,input().split())
    if unset.find(a)==unset.find(b):
        has_cycle=True
    else:
        unset.union(a,b)
root=unset.find(0)
for i in range(n):
    if unset.find(i)!=root:
        is_connected=False
        break
print(f"connected:{'yes' if is_connected else 'no'}")
print(f"loop:{'yes' if has_cycle else 'no'}")
```

三、图!

1、dfs，bfs，图的遍历等问题

```
move=[(-1,1),(-1,0),(-1,-1),(0,1),(0,-1),(1,0),(1,-1),(1,1)]
cnt=0
def dfs(x,y): ###最大联通域面积
    global cnt
    cnt+=1
    visited[x][y]=1
    for dx,dy in move:
        nx=x+dx
        ny=y+dy
        if 0<=nx<n and 0<=ny<m and gra[nx][ny]=='W' and visited[nx][ny]==0:
            dfs(nx,ny)
    return cnt

T=int(input())
out=[]
for _ in range(T):
    n,m=map(int,input().split())
    gra=[]
    ans=0
    visited=[[0]*m for i in range(n)]
    for i in range(n):
        gra.append(list(input()))
    for i in range(n):
        for j in range(m):
```

```
            if gra[i][j]=='W' and visited[i][j]==0:
                cnt=0
                dfs(i,j)
                ans=max(ans,cnt)
    out.append(ans)
for i in out:
    print(i)
```

```
a=0 ##最大权值联通块
def dfs(vertex):
    global a
    if visited[vertex]==True:
        return
    visited[vertex]=True
    a+=v_value[vertex]
    for i in edge[vertex]: ##! ! ! !
        dfs(i)

n,m=map(int,input().split())
v_value=list(map(int,input().split()))
edge={i:[] for i in range(n)}
for i in range(m):
    a,b=map(int,input().split())
    edge[a].append(b)
    edge[b].append(a)
ans=0
for i in range(n):
    visited=[False]*n ##写得不好，只是看一下字典见图的类推
    a=0
    dfs(i)
    ans=max(ans,a)
print(ans)
```

```
##Warnsdorff规则的核心是每一步选择当前可达格子中，下一步可达格子数目最少的格子，具体实现通过排序来完成。这种策略有效地避免了死胡同情况的发生，从而提高了搜索效率。
move=[(1,2),(1,-2),(-1,2),(-1,-2),(2,1),(2,-1),(-2,1),(-2,-1)]
ans=0
def getNeighbor(x,y):
    return [(x+dx,y+dy) for dx,dy in move if 0<=x+dx<n and 0<=y+dy<n and visited[x+dx]
[y+dy]==0]

def dfs(x,y,cnt):
    visited[x][y]=1
    global ans
    if cnt==n*n:
        return True
    for x2,y2 in sorted(getNeighbor(x,y),key=lambda c:len(getNeighbor(c[0],c[1]))):
 ##! ! ! !
        if dfs(x2,y2,cnt+1):
            return True  ##!!!!!!!!!
        visited[x2][y2]=0
```

```
##不在最后加 return False 也是对的，因为如果找不到成功的路径，函数会在所有递归调用都失败时自然返回False。
n=int(input())
x,y=map(int,input().split())
visited=[[0]*n for i in range(n)]
ans=0
print('success' if dfs(x,y,1) else 'fail')
```

```
def bfs(x): ###Find The Multiple（找101010101倍数）——抽象宽搜也要记得找一个visited的标记！！！！
    q=[(1,'1')]
    while q:
        mol,cur=q.pop(0)
        for i in ['0','1']:
            if mol==0:
                return cur
            if visited[(mol*10+int(i))%x]==0:
                q.append(((mol*10+int(i))%x,cur+i))
                visited[(mol*10+int(i))%x]=1
ans=[]
while True:
    x=int(input())
    visited=[0]*x
    if x==0:
        break
    ans.append(bfs(x))
for i in ans:
    print(i)
```

```
dire=[(-1,0),(0,-1),(1,0),(0,1)]  ###鸣人与佐助 最少需要花费多少时间；可以打查克拉
ans=0
flag=0
def bfs(x,y,t):
    global ans,flag
    q=[]
    visited=set()
    q.append((t,x,y,0))
    while q:
        t,x,y,ans=q.pop(0)
        for dx,dy in dire:
            nx=x+dx
            ny=y+dy
            if 0<=nx<m and 0<=ny<n:
                if gra[nx][ny]!="#":
                    nt=t
                else:
                    nt=t-1
                if nt>=0 and (nt,nx,ny) not in visited:
                    nans=ans+1
                    if gra[nx][ny]=="+":
                        flag=1
```

```
                        return flag,nans ##最先找到的就是最快的！！因为是层次遍历
                    q.append((nt, nx, ny, nans))
                    visited.add((nt,nx,ny)) ###！！！！
    return flag,ans

m,n,t=map(int,input().split())
gra=[]
        if gra[i][j]=='@':
            x=i
            y=j
flag,ans=bfs(x,y,t)
```

```
##深度优先遍历无向图
def dfs(graph, visited, node):
    visited[node] = True
    print(node, end=" ")
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs(graph, visited, neighbor)

def main():
    n, m = map(int, input().split())
    graph = [[] for _ in range(n)]
    visited = [False] * n
    for _ in range(m):
        a, b = map(int, input().split())
        graph[a].append(b)
        graph[b].append(a)
    for i in range(n):
        if not visited[i]:
            dfs(graph, visited, i)
if __name__ == "__main__":
    main()
```

2、最小生成树（Prim和Kruskal）& dijkstra &Kahn等等其他算法应用

```
##Prim!!（兔子与星空） （A 2 B 12 I 25）
import heapq
def prim(graph,start):
    mst=[]
    used=set([start])
    edges=[(cost, start, to) for to,cost in graph[start].items()]
    heapq.heapify(edges)
    while edges:
        cost,frm,to=heapq.heappop(edges)
        if to not in used:
            used.add(to)
            mst.append((frm,to,cost))
            for to_next,cost2 in graph[to].items():
```

```
                if to_next not in used:
                    heapq.heappush(edges,(cost2,to,to_next))
    return mst ##返回的是权值和最小时的路径

n = int(input())
graph = {chr(i+65): {} for i in range(n)}
for i in range(n-1):
    data = input().split()
    star = data[0]
    m = int(data[1])
    for j in range(m):
        to_star = data[2+j*2]
        cost = int(data[3+j*2])
        graph[star][to_star] = cost
        graph[to_star][star] = cost
mst = prim(graph, 'A')
print(sum(x[2] for x in mst))
```

```
def kruskal(n,edges): ##krusal (繁忙的厦门)
    uf=UnionFind(n)
    edges.sort(key=lambda x:x[2]) ##! ! !
    mst,max_edge=0,0
    for u, v, w in edges:
        if uf.find(u)!=uf.find(v):
            uf.union(u, v)
            mst+=1   ## mst.append((u, v, weight)), 就能输出路径
            max_edge=max(max_edge,w)
            if mst==n-1:
                break
    return mst, max_edge
```

```
##堆实现 走山路
import heapq
def dijkstra(graph,start):
    m,n=len(graph),len(graph[0])
    distances={(i,j):float('inf') for i in range(m) for j in range(n)}
    distances[(start[0],start[1])]=0
    priority_queue=[(0,start)]
    while priority_queue:
        current_distance,(cx, cy)=heapq.heappop(priority_queue)
        if current_distance>distances[(cx, cy)]:
            continue
        for dx,dy in [(0,-1),(0,1),(1,0),(-1,0)]:
            nx,ny=cx+dx,cy+dy
            if 0<=nx<m and 0<=ny<n and graph[nx][ny]!='#':
                new_distance=current_distance+abs(int(graph[nx][ny])-int(graph[cx][cy]))
                if new_distance<distances[(nx, ny)]:
                    distances[(nx,ny)]=new_distance
                    heapq.heappush(priority_queue,(new_distance,(nx,ny)))
    return distances
```

```
m,n,p=map(int,input().split())
graph=[]
for _ in range(m):
    row=input().split()
    graph.append(row)
for _ in range(p):
    x1,y1,x2,y2=map(int,input().split())
    if graph[x1][y1]== '#' or graph[x2][y2]=='#':
        print('NO')
        continue
    distances=dijkstra(graph, (x1, y1))
    if (x2, y2) in distances and distances[(x2, y2)]!=float('inf'):
        print(distances[(x2, y2)])
    else:
        print('NO')
```

```
##正常djikstra
dire=[(0,-1),(0,1),(1,0),(-1,0)]
def bfs(x,y):
    q=[(x,y)]
    distances={(x,y):0}
    while q:
        cx,cy=q.pop(0)
        for dx,dy in dire:
            nx,ny=cx+dx,cy+dy
            if 0<=nx<m and 0<=ny<n and gra[nx][ny]!='#':
                new_distance=distances[(cx, cy)]+abs(int(gra[nx][ny])-int(gra[cx][cy]))
                if (nx,ny) not in distances or new_distance<distances[(nx,ny)]:
                    distances[(nx,ny)]=new_distance
                    q.append((nx,ny))
    return distances
```

```
##兔子与樱花的djikstra（（ difference?
from heapq import heappop,heappush
from collections import defaultdict
def dijkstra(start,end):
    heap=[(0,start,[start])]
    vis=set()
    while heap:
        (cost,u,path)=heappop(heap)
        if u not in vis:
            vis.add(u)
            if u==end:
                return [cost,path]
            for v in graph[u]:
                if v not in vis:
                    heappush(heap,(cost+graph[u][v],v,path+[v]))

n=int(input())
name=[]
for i in range(n):
```

```
    name.append(input())
graph=defaultdict(dict)
for i in range(int(input())):
    x,y,l=input().split()
    graph[x][y]=graph[y][x]=int(l)
N=int(input())
for i in range(N):
    start,end=input().split()
    cost,path=dijkstra(start,end)
    for j in range(len(path)-1):
        print(f'{path[j]}->({graph[path[j]][path[j+1]]})->',end='')
    print(path[-1])
```

```
def isCyclicKahn(graph,V):  ##舰队海域 Kahn算法
    in_degree=[0]*V
    for u in range(V):
        for v in graph[u]:
            in_degree[v]+=1
    queue=[]
    for i in range(V):
        if in_degree[i]==0:
            queue.append(i)
    count=0
    while queue:
        u=queue.pop(0)
        count+=1
        for v in graph[u]:
            in_degree[v]-=1
            if in_degree[v]==0:
                queue.append(v)
    return count!=V

T=int(input())
ans=[]
for _ in range(T):
    N, M=map(int,input().split())
    graph=[[] for _ in range(N)]
    for _ in range(M):
        u,v=map(int,input().split())
        graph[u-1].append(v-1)
    ans.append("Yes" if isCyclicKahn(graph,N) else "No")
for i in ans:
    print(i)
```

```
n,m=map(int,input().split())  ##最小奖金方案（Khan算法拓扑排序例子～）
gra=[[] for i in range(n)]
award=[0 for i in range(n)]
inDegree=[0 for i in range(n)]
for i in range(m):
  a,b=map(int,input().split())
  gra[b].append(a)
```

```
  inDegree[a]+=1
q=[]
for i in range(n):
  if inDegree[i]==0:
    q.append(i)
    award[i]=100
while len(q)>0:
  u=q.pop(0)
  for v in gra[u]:
    inDegree[v]-=1
    award[v]=max(award[v],award[u]+1) ##注意是max!
    if inDegree[v]==0:
      q.append(v)
total=sum(award)
print(total)
```

四、栈，堆，队列

```
decimal = int(input())  # 读取十进制数
stack = []
if decimal == 0:
    print(0)
else:
    while decimal > 0: # 不断除以8，并将余数压入栈中
        remainder = decimal % 8
        stack.append(remainder)
        decimal = decimal // 8
    octal = ""
    while stack:
        octal += str(stack.pop())
    print(octal)
```

```
def infix_to_postfix(expression): ##调度场算法
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''
    for char in expression:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char] <=
precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
```

```
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()
    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)
    while stack:
        postfix.append(stack.pop())
    return ' '.join(str(x) for x in postfix)

n = int(input())
for _ in range(n):
    expression = input()
    print(infix_to_postfix(expression))
```

要解决这个问题，需要理解如何计算后序表达式。后序表达式的计算可以通过使用一个栈来完成，按照以下步骤：

1. 从左到右扫描后序表达式。 ##后序表达式的计算
2. 遇到数字时，将其压入栈中。
3. 遇到运算符时，从栈中弹出两个数字，先弹出的是右操作数，后弹出的是左操作数。将这两个数字进行相应的运算，然后将结果压入栈中。
4. 当表达式扫描完毕时，栈顶的数字就是表达式的结果。

```
s = input()  # 读取输入的括号字符串
a = [-1]     # 初始化栈，并放入一个初始位置 -1
ans = 0      # 初始化最长长度为 0
for i, c in enumerate(s):
    if c == '(':  # 如果是左括号
        a.append(i)  # 将索引位置压入栈
    else:  # 如果是右括号
        a.pop()  # 从栈中弹出一个位置
        if a:  # 如果栈不为空
            ans = max(ans, i - a[-1])  # 计算当前有效子串长度，并更新最长长度
        else:  # 如果栈为空
            a.append(i)  # 将当前索引位置压入栈
print(ans)  # 输出最长格式正确的括号子串的长度
```