**·归并排序&求逆序数/交换次数：**

def MergeSort(arr):

n=len(arr)

if n<=1:

return arr,0

middle=n//2

left,left\_swap=MergeSort(arr[:middle])

right,right\_swap=MergeSort(arr[middle:])

swaps=left\_swap+right\_swap

result=[]

i=j=0

while i<len(left) and j<len(right):

if left[i]<right[j]:

result.append(left[i])

i+=1

else:

result.append(right[j])

j+=1

swaps+=(middle-i)

if i==len(left):

result.extend(right[j:])

else:

result.extend(left[i:])

return result,swaps

·**堆排序&堆的构建:**

def Big\_Heap(arr,start,end): #构建大根堆

root=start

child=root\*2+1

while child<=end:

if child+1<=end and arr[child]<arr[child+1]:

child+=1

if arr[root]<arr[child]:

arr[root],arr[child]=arr[child],arr[root]

root=child

child=root\*2+1

else:

break

def HeapSort(arr): #堆排序

first=len(arr)//2-1

for start in range(first,-1,-1):

Big\_Heap(arr,start,len(arr)-1)

for end in range(len(arr)-1,0,-1):

arr[0],arr[end]=arr[end],arr[0]

Big\_Heap(arr,0,end-1)

return arr

·**中序表达式转后序表达式(调度场算法)：**

def infix\_to\_postfix(infix):

prec={‘+’:1,‘-’:1,‘\*’:2,‘/’:2}

stack=[]

postfix=[]

for token in infix:

if token.isdigit():

postfix.append(token)

elif token==‘(’:

stack.append(token)

elif token==‘)’:

while stack[-1]!=‘(’:

postfix.append(stack.pop())

stack.pop()

else:

while stack and stack[-1]!=‘(’ and prec[token]<=prec[stack[-1]]:

postfix.append(stack.pop())

stack.append(token)

while stack:

postfix.append(stack.pop())

return postfix

**·中序表达式转前序表达式：**

def infix\_to\_prefix(infix):

prec={‘+’:1,‘-’:1,‘\*’:2,‘/’:2}

stack=[]

prefix=[]

for token in infix[::-1]:

if token.isdigit():

prefix.append(token)

elif token==‘)’:

stack.append(token)

elif token==‘(’:

while stack[-1]!=‘)’:

prefix.append(stack.pop())

stack.pop()

else:

while stack and stack[-1]!=‘)’and prec

[token]<prec[stack[-1]]:

prefix.append(stack.pop())

stack.append(token)

while stack:

prefix.append(stack.pop())

return prefix[::-1]

**Tips:**方便起见可对输入数据进行如下处理

infix=input().replace(‘+',‘ + ').replace(‘-',‘ - '). replace(‘\*',‘ \* ').replace(‘/',‘ / ').replace(‘(', ‘ ( ').replace(‘)',‘ ) ’).split() #增加空格,便于分离

**·计算前序表达式：**

方法一：栈

def calculate(prefix):

stack=[]

for token in prefix[::-1]:

if token.isdigit():

stack.append(token)

else:

a=int(stack.pop())

b=int(stack.pop())

if token==‘+’:

stack.append(a+b)

elif token==‘-’:

stack.append(a-b)

elif token==‘\*’:

stack.append(a\*b)

elif token==‘/’:

stack.append(a/b)

return stack[0]

方法二：用函数写递归

index=-1

def exp():

global index

index+=1

a=string[index]

if a=='+':

return exp()+exp()

if a=='-':

return exp()-exp()

if a=='\*':

return exp()\*exp()

if a=='/':

return exp()/exp()

else:

return float(a)

**·计算后序表达式：**

def calculate(postfix):

stack=[]

for token in postfix:

if token.isdigit():

stack.append(token)

else:

a=int(stack.pop())

b=int(stack.pop())

if token==‘+’:

stack.append(b+a)

elif token==‘-’:

stack.append(b-a)

elif token==‘\*’:

stack.append(b\*a)

elif token==‘/’:

stack.append(b/a)

return stack[0]

**·合法出栈序列：**

def is\_possible\_out\_stack(orig,test):

stack=[]

index=0

if len(orig)!=len(test):

return False

for char in test:

if char not in orig:

return False

while not stack or stack[-1]!=char:

if index==len(orig):

return False

stack.append(orig[index])

index+=1

stack.pop()

return True

**·出栈序列统计(卡特兰数)：**

from collections import comb

n=int(input())

ans=int(comb(2\*n,n)/(n+1))

**·单调栈(monotone stack)：**

1.寻找**左侧**第一个**比当前元素大**的元素(或其索引)：

def monotone\_increasing\_stack(nums):

ans=[None]\*len(nums)

stack=[]

for i in range(len(nums)):

while stack and nums[stack[-1]]<=nums[i]:

stack.pop()

if stack:

ans[i]=nums[stack[-1]]或stack[-1]

stack.append(i)

2.寻找**左侧**第一个**比当前元素小**的元素(或其索引)：

def monotone\_decreasing\_stack(nums):

ans=[None]\*len(nums)

stack=[]

for i in range(len(nums)):

while stack and nums[stack[-1]]>=nums[i]:

stack.pop()

if stack:

ans[i]=nums[stack[-1]]或stack[-1]

stack.append(i)

3.寻找**右侧**第一个**比当前元素大**的元素(或其索引)：

def monotone\_increasing\_stack(nums):

ans=[None]\*len(nums)

stack=[]

for i in range(len(nums)-1,-1,-1):

while stack and nums[stack[-1]]<=nums[i]:

stack.pop()

if stack:

ans[i]=nums[stack[-1]]或stack[-1]

stack.append(i)

4.寻找**右侧**第一个**比当前元素小**的元素(或其索引)：

def monotone\_decreasing\_stack(nums):

ans=[None]\*len(nums)

stack=[]

for i in range(len(nums)-1,-1,-1):

while stack and nums[stack[-1]]>=nums[i]:

stack.pop()

if stack:

ans[i]=nums[stack[-1]]或stack[-1]

stack.append(i)

·**二叉树的遍历：**

1.先序遍历：访问根结点→先序遍历左子树→先序遍历右子树

def preorder\_travelsal(root):

if not root:

return

res=[]

res.append(root.val)

res.append(preorder\_traversal(root.left))

res.append(preorder\_traversal(root.right))

return res

2.中序遍历：中序遍历左子树→访问根结点→中序遍历右子树

def inorder\_travelsal(root):

if not root:

Return

res=[]

res.append(inorder\_traversal(root.left))

res.append(root.val)

res.append(inorder\_traversal(root.right))

return res

3.后序遍历：后序遍历左子树→后序遍历右子树→访问根结点

def postorder\_travelsal(root):

if not root:

return

res=[]

res.append(postorder\_traversal(root.left))

res.append(postorder\_traversal(root.right))

res.append(root.val)

return res

4.层次遍历(也称为广度优先遍历)：按照从上到下的层次顺序，从左到右的结点顺序进行遍历

def level\_traversal(root):

result=[]

if not root:

return result

queue=deque([root])

while queue:

node=queue.popleft()

result.append(node.value)

if node.left:

queue.append(node.left)

if node.right:

queue.append(node.right)

return result

5.根据前中遍历序列得后序遍历序列：

def postorder(preorder,inorder):

if not preorder or not inorder:

return []

root=preorder[0]

rootindex=inorder.index(root)

leftinorder=inorder[:rootindex]

rightinorder=inorder[rootindex+1:]

leftpreorder=preorder[1:len(leftinorder)+1]

rightpreorder=preorder[len(leftinorder)+1:]

tree=[]

tree.extend(postorder(leftpreorder,leftinorder))

tree.extend(postorder(rightpreorder,rightinorder))

tree.append(root)

return tree

6.根据中后遍历序列得前序遍历序列：

def preorder(inorder,postorder):

if not inorder or not postorder:

return []

root=postorder[-1]

rootindex=inorder.index(root)

leftinorder=inorder[:rootindex]

rightinorder=inorder[rootindex+1:]

leftpostorder=postorder[:len(leftinorder)]

rightpostorder=postorder[len(leftinorder):-1]

tree=[root]

tree.extend(preorder(leftinorder,leftpostorder)) tree.extend(preoreder(rightinorder,rightpostorder)

return tree

**·二叉搜索/查找树：**

class Treenode:

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

self.val=val

self.left=None

self.right=None

def insert(root,key): #插入

if not root:

return Treenode(key)

if root.val>key:

root.left=insert(root.left,key)

if root.val<key:

root.right=insert(root.right,key)

return root

root=None

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

for key in keys:

root=insert(root,key)

def search(node,parent,key): #搜索

if not node:

return node,parent

if node.val==key:

return node,parent

elif node.val>key:

return search(node.left,node,key)

else:

return search(node.right,node,key)

def delete(root,key): #删除

node,parent=search(root,root,key)

if node:

if node.left is None:

if node==parent.left:

parent.left=node.right

else:

parent.right=node.right

del node

elif node.right is None:

if node==parent.left:

parent.left=node.left

else:

parent.right=node.left

del node

else: #左右子树均不为空

pre=node.right

if pre.left is None:

node.val=pre.val

node.right=pre.right

del pre

else:

next=preleft

while next.left is not None:

pre=next

next=next.left

node.data=next.data

pre.left=next.right

del next

**·平衡二叉树：**

class Treenode:

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

self.val=val

self.left=None

self.right=None

self.height=1

class AVLtree:

def get\_height(self,node): #获取树的高度

if not node:

return 0

return node.height

def get\_balance\_factor(self,node): #获取平衡因子

if not node:

return 0

return self.get\_height(node.left)- self.get\_ hei- ght(node.right)

def right\_rotate(self,lost\_balance\_node): #右旋操作

new\_node=lost\_balance\_node.left right\_subtree=new\_node.right

new\_node.right=lost\_balance\_node

lost\_balance\_node.left=right\_subtree

lost\_balance\_node.height=1+max(self.get\_height(lost\_balance\_node.left),self.get\_height(lost\_balance\_node.right))

new\_node.height=1+max(self.get\_height(new\_node.left),self.get\_height(new\_node.right))

return new\_node

def left\_rotate(self,lost\_balance\_node): #左旋操作

new\_node=lost\_balance\_node.right

left\_subtree=new\_node.left

new\_node.left=lost\_balance\_node

lost\_balance\_node.right=left\_subtree

lost\_balance\_node.height=1+max(self.get\_height(lost\_balance\_node.left),self.get\_height(lost\_balance\_node.right))

new\_node.height=1+max(self.get\_height(new\_node.left),self.get\_height(new\_node.right))

return new\_node

def insert(self,node,key): #插入AVL树

if not node:

return Treenode(key)

if node.val>key:

node.left=self.insert(node.left,key)

if node.val<key:

node.right=self.insert(node.right,key)

node.height=1+max(self.get\_height(node.left),self.get\_height(node.right))

balance\_factor=self.get\_balance\_factor(node)

if balance\_factor>1 and key<node.left.val: #LL型失衡，右旋一次

return self.right\_rotate(node)

if balance\_factor<-1 and key>node.right.val: #RR型失衡，左旋一次

return self.left\_rotate(node)

if balance\_factor>1 and key>node.left.val: #LR型失衡，失衡节点的左子树左旋一次，然后整个树右旋一次

node.left=self.left\_rotate(node.left)

return self.right\_rotate(node)

if balance\_factor<-1 and key<node.right.val: #RL型失衡，失衡节点右子树右旋一次，然后整个树左旋一次

node.right=self.right\_rotate(node.right)

return self.left\_rotate(node)

return node

·**Huffman编码树：**

class Node(object): #节点类

def \_\_init\_\_(self,name=None,value=None):

self.\_name=name

self.\_value=value

self.\_left=None

self.\_right=None

class HuffmanTree(object): #哈夫曼树类

def \_\_init\_\_(self,char\_weights):

self.a=[Node(part[0],part[1]) for part in char\_ weights] #根据输入的字符及其频数生成叶子节点

while len(self.a)!=1:

self.a.sort(key=lambda node:node.\_value,rev erse=True)

c=Node(value=(self.a[-1].\_value+self.a[-2].\_value))

c.\_left=self.a.pop(-1)

c.\_right=self.a.pop(-1)

self.a.append(c)

self.root=self.a[0]

self.b=range(10) #self.b用于保存每个叶子节点的Haffuman编码,range的值只需要不小于树的深度就行

def pre(self,tree,length):#用递归的思想生成编码

node=tree

if (not node):

return

elif node.\_name:

return node.\_name,self.b[:length]

self.b[length]=0

self.pre(node.\_left,length+1)

self.b[length]=1

self.pre(node.\_right,length+1)

def get\_code(self): #生成哈夫曼编码

self.pre(self.root,0)

**·字典树：**

class Node:

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

self.children={} #当前节点的子节点字典

class Trie:

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

self.root=Node() #Trie的根节点为空

def insert(self,phone\_number): #添加新子树

node=self.root

for num in phone\_number:

if num not in node.children: #当前数字不在子节点字典中，则添加

node.children[num]=Node()

node=node.children[num]

def startwith(self,prefix):

node=self.root

for num in prefix:

if num not in node.children: #当前数字没有在子节点字典中出现，则不是前缀

return False

node=node.children[num]

return True

**·并查集：**

def find(x):

if parents[x]!=x:

parents[x]=find(parents[x])

return parents[x]

def union(x,y):

parents[find(y)]=find(x)

·**bfs：**

from collections import deque

def bfs(graph,start\_node):

queue=deque([start\_node])

visited=set()

visited.add(start\_node)

while queue:

current\_node=queue.popleft()

for neighbor in graph[current\_node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

**·dfs/回溯：**

def dfs(row,k):

if k==0:

    return 1

  if row==n:

    return 0

  count=0

  for col in range(n):

    if board[row][col]=='#' and not col\_occupied[col]:

      col\_occupied[col]=True

      count+=dfs(row+1,k-1)

      col\_occupied[col]=False

  count+=dfs(row+1,k)

  return count

col\_occupied=[False]\*n

print(dfs(0,k))

**·dijkstra:**

1.使用vis集合

def dijkstra(start,end):

heap=[(0,start,[start])]

vis=set()

  while heap:

(cost,u,path)=heappop(heap)

if u in vis: continue

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]))

2.使用dist数组

import heapq

def dijkstra(graph,start):

distances={node:float('inf')for node in graph}

  distances[start]=0

  priority\_queue=[(0,start)]

  while priority\_queue:

current\_distance,current\_node=heapq.heappop(priority\_queue)

if current\_distance>distances[current\_node]:

      continue

    for neighbor,weight in graph[current\_node].items():

      distance=current\_distance+weight

      if distance<distances[neighbor]:

        distances[neighbor]=distance

        heapq.heappush(priority\_queue,(distance,neighbor))

  return distances

**·最小生成树：**

1.Prim算法

while len(visited)<len(Graph):

min\_weight=float('inf')

min\_edge=None

for node in visited:

for edge in Graph[node]:

if edge not in visited:

if Graph[node][edge]<min\_weight:

min\_weight=Graph[node][edge]

min\_edge=edge

if min\_edge:

total\_weight+=min\_weight

visited.add(min\_edge)

2.kruskal算法(要用到并查集)：

X = dict()

R = dict()

def make\_set(point):

X[point] = point

R[point] = 0

def find(point):

if X[point] != point:

X[point] = find(X[point])

return X[point]

def merge(point1, point2):

r1 = find(point1)

r2 = find(point2)

if r1 != r2:

if R[r1] > R[r2]:

X[r2] = r1

else:

X[r1] = r2

if R[r1] == R[r2]:

R[r2] += 1

def kruskal(vertices,edges):

for vertice in vertices:

make\_set(vertice)

minu\_tree = []

edges.sort() # 按照权重从小到大排序

for edge in edges:

weight, vertice1, vertice2 = edge

if find(vertice1) != find(vertice2):

merge(vertice1, vertice2)

minu\_tree.append(edge)

return minu\_tree

**·判断无向图是否联通、是否有环：**

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

graph={i:[] for i in range(n)}

for j in range(m):

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

graph[a].append(b)

graph[b].append(a)

def is\_connected(graph,n):

visited=set()

stack=[0]

while stack:

node=stack.pop()

if node not in visited:

visited.add(node)

stack.extend(graph[node])

return len(visited)==n

def has\_loop(graph):

visited=set()

stack=[(0,-1)]

while stack:

node,parent=stack.pop()

if node in visited:

return True

visited.add(node)

for neighbor in graph[node]:

if neighbor!=parent:

stack.append((neighbor,node))

return False

**·判断有向图是否成环(拓扑排序)：**

def has\_loop(graph):

queue=deque([])

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

if in\_degrees[i]==0:

queue.append(i)

while queue:

node=queue.popleft()

visited.add(node)

for neighbor in graph[node]:

in\_degrees[neighbor]-=1

if in\_degrees[neighbor]==0:

queue.append(neighbor)

return N!=len(visited)

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

graph=defaultdict(list)

in\_degrees=[0]\*(N+1)

visited=set()

for \_ in range(M):

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

graph[x].append(y)

in\_degrees[y]+=1

if has\_loop(graph):

print('Yes')

else:

print('No')