Debug 点：

词典是不是因为重复值被覆盖

是不是有重复值

是不是有孤立点

输入是一组还是多组

有没有考虑 0，不止一位数，负数，空

注意节点序号是从 0 开始还是从 1 开始

用到优先队列的时候，一定要把 weight 放在第一位

行列有没有弄反

索引是不是从 0 开始！！！！尤其在树，图的问题中，一定注意

语法点：

浅拷贝和深拷贝，浅拷贝是全部相互影响，深拷贝互不影响

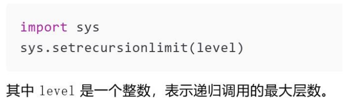
import copy

ss=copy.copy(ii)

dd=copy.deepcopy(ii)

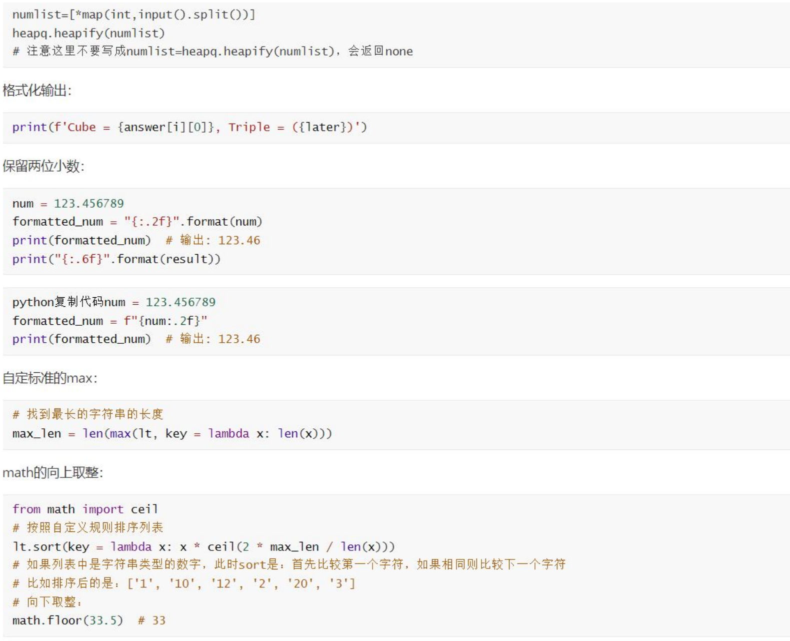
.第一行加 # pylint: skip-fle 可以忽略检查

递归深度调整：



```
import sys
sys.setrecursionlimit(level)
```

其中 level 是一个整数，表示递归调用的最大层数。



```
numlist=[*map(int,input().split())]
heapq.heapify(numlist)
# 注意这里不要写成numlist=heapq.heapify(numlist)，会返回none
```

格式化输出:

```
print(f'Cube = {answer[i][0]}, Triple = ({later})')
```

保留两位小数:

```
num = 123.456789
formatted_num = "{:.2f}".format(num)
print(formatted_num)  # 输出: 123.46
print("{:.6f}".format(result))
```

```
python复制代码num = 123.456789
formatted_num = f"{num:.2f}"
print(formatted_num)  # 输出: 123.46
```

自定标准的max:

```
# 找到最长的字符串的长度
max_len = len(max(lt, key = lambda x: len(x)))
```

math的向上取整:

```
from math import ceil
# 按照自定义规则排序列表
lt.sort(key = lambda x: x * ceil(2 * max_len / len(x)))
# 如果列表中是字符串类型的数字，此时sort是：首先比较第一个字符，如果相同则比较下一个字符
# 比如排序后的是：['1', '10', '12', '2', '20', '3']
# 向下取整：
math.floor(33.5)  # 33
```

if char.isalpha():   # 如果是字母

‘123’isdigit() #是不是数字

islower(),isupper()

自定义标准排序：

candies=sorted(candies,key=lambda x:x[0]/x[1],reverse=True)

辗转相除算最大公约数：

```
def gcd(a, b):
  while b:
    a, b = b, a % b
  return a
```

自定义可迭代的 deque：

```
1.from collections import deque
class MyDeque:
  def **init**(self):
    self.deque = deque()
```

```
def __iter__(self):
    return iter(self.deque)
```

不定参数接受：

op, *args = map(int, input().split())

args 此时是一个列表

`iterable.count(value)`

`str.find(sub) #未找到抛出-1`

`list.index(x) #未找到抛出 ValueError`

replace() 替换字符串中的指定字符， eval() 函数计算表达式的值

enumerate 快速获取索引和值： for index, value in enumerate(list, start)

冒泡排序应用：

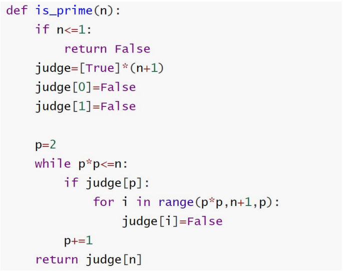
n 个数组合，求最大的数，不超过多少位

```
m=int(input())# 最大位数
n=int(input())# 正整数数量
l=input().split()

# 冒泡排序
for i in range(n):
    for j in range(n-1-i):
        if l[j]+l[j+1]>l[j+1]+l[j]:
            l[j],l[j+1]=l[j+1],l[j]
# 从小到大排序
weight=[len(i) for i in l]

dp=[['']*(m+1) for _ in range(n+1)]
#dp[i][j]表示在前i个数中选，不超过j位
for k in range(m+1):
    dp[0][k]=''
for q in range(n+1):
    dp[q][0]=''
for i in range(1,n+1):
    for j in range(1,m+1):
        if weight[i-1]>j:#肯定不能选,防止超出索引
            dp[i][j]=dp[i-1][j]
        else:
            dp[i][j]=str(max(f(dp[i-1][j]),int(l[i-1]+dp[i-1][j-weight[i-1]])))

print(dp[n][m])
```

归并排序：

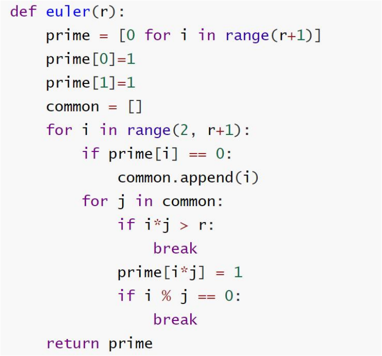
```
res=0
def merge_count(lis):
    if len(lis)<=1:
        return lis
    global res

    mid=len(lis)//2

    left = merge_count(lis[:mid])  # Dividing the array elements
    right = merge_count(lis[mid:])  # Into 2 halves

    merged = []
    while left and right:
        if left[0] <= right[0]:
            merged.append(left.pop(0))
        else:
            merged.append(right.pop(0))
            res+=len(left)
    merged.extend(right if right else left)
    return merged
```

埃氏筛法：



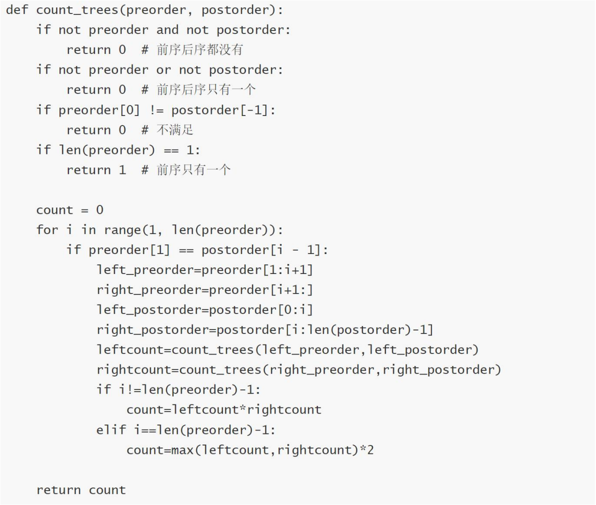
```
def is_prime(n):
    if n<=1:
        return False
    judge=[True]*(n+1)
    judge[0]=False
    judge[1]=False

    p=2
    while p*p<=n:
        if judge[p]:
            for i in range(p*p,n+1,p):
                judge[i]=False
        p+=1
    return judge[n]
```

欧式筛法：



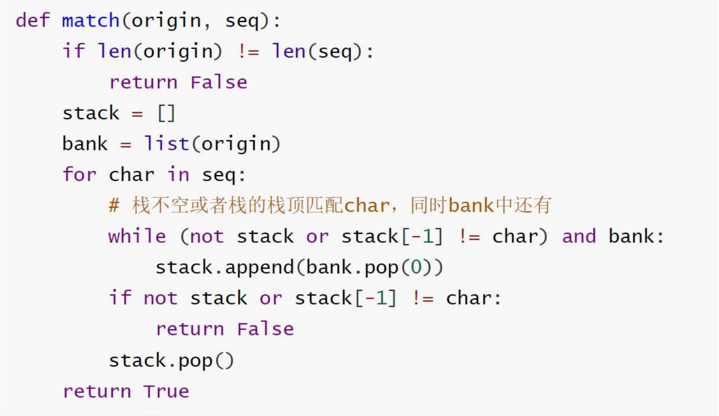
```
def euler(r):
    prime = [0 for i in range(r+1)]
    prime[0]=1
    prime[1]=1
    common = []
    for i in range(2, r+1):
        if prime[i] == 0:
            common.append(i)
        for j in common:
            if i*j > r:
                break
            prime[i*j] = 1
            if i % j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

知道树的前序和后序遍历，求有几种可能：



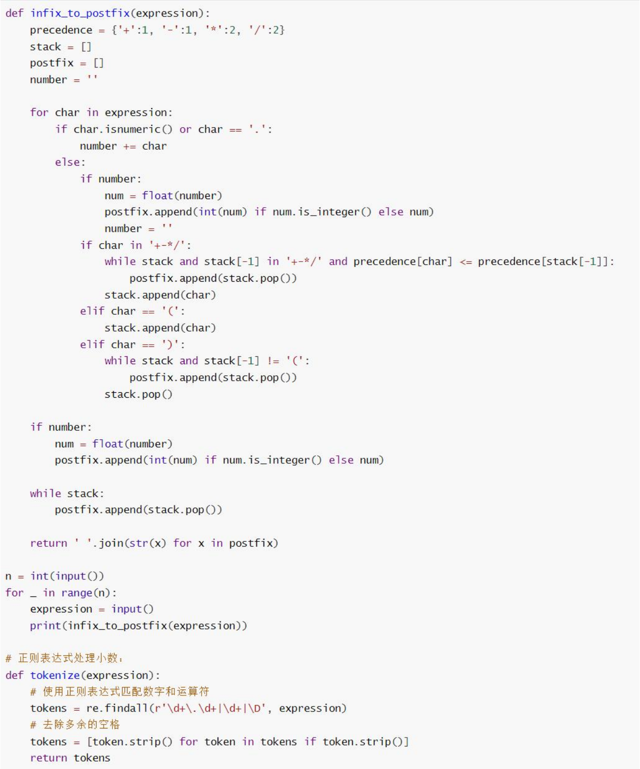
```
def count_trees(preorder, postorder):
    if not preorder and not postorder:
        return 0  # 前序后序都没有
    if not preorder or not postorder:
        return 0  # 前序后序只有一个
    if preorder[0] != postorder[-1]:
        return 0  # 不满足
    if len(preorder) == 1:
        return 1  # 前序只有一个

    count = 0
    for i in range(1, len(preorder)):
        if preorder[1] == postorder[i - 1]:
            left_preorder=preorder[1:i+1]
            right_preorder=preorder[i+1:]
            left_postorder=postorder[0:i]
            right_postorder=postorder[i:len(postorder)-1]
            leftcount=count_trees(left_preorder,left_postorder)
            rightcount=count_trees(right_preorder,right_postorder)
            if i!=len(preorder)-1:
                count=leftcount*rightcount
            elif i==len(preorder)-1:
                count=max(leftcount,rightcount)*2

    return count
```

合法出栈序列：



```
def match(origin, seq):
    if len(origin) != len(seq):
        return False
    stack = []
    bank = list(origin)
    for char in seq:
        # 栈不空或者栈的栈顶匹配char，同时bank中还有
        while (not stack or stack[-1] != char) and bank:
            stack.append(bank.pop(0))
        if not stack or stack[-1] != char:
            return False
        stack.pop()
    return True
```

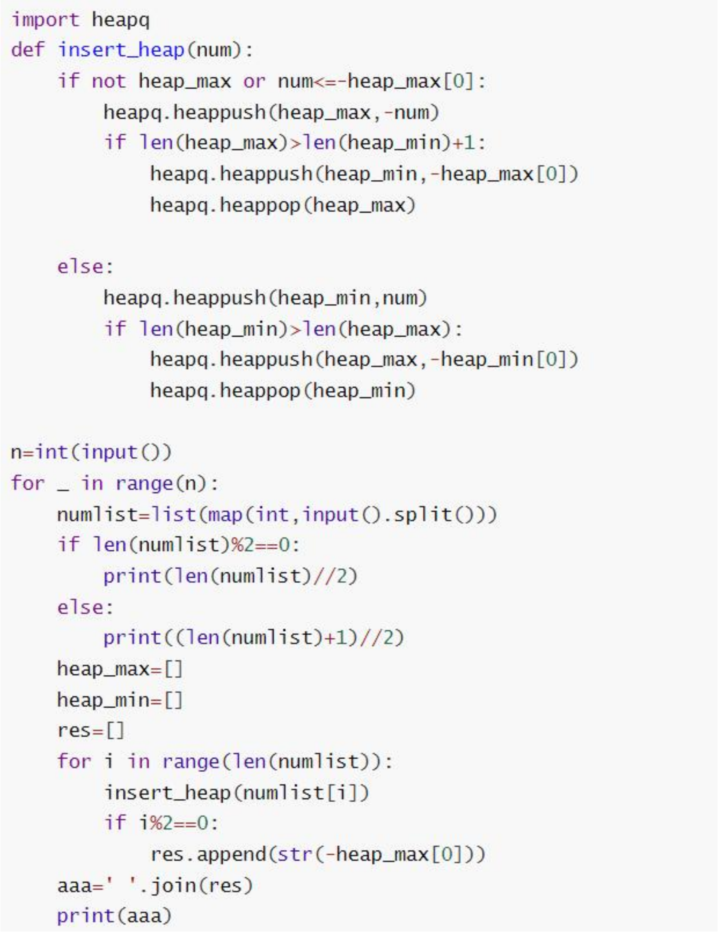
中置表达式转后置表达式（如何处理小数）



```python
def infix_to_postfix(expression):
    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))

# 正则表达式处理小数：
def tokenize(expression):
    # 使用正则表达式匹配数字和运算符
    tokens = re.findall(r'\d+\.\d+|\d+|\D', expression)
    # 去除多余的空格
    tokens = [token.strip() for token in tokens if token.strip()]
    return tokens
```

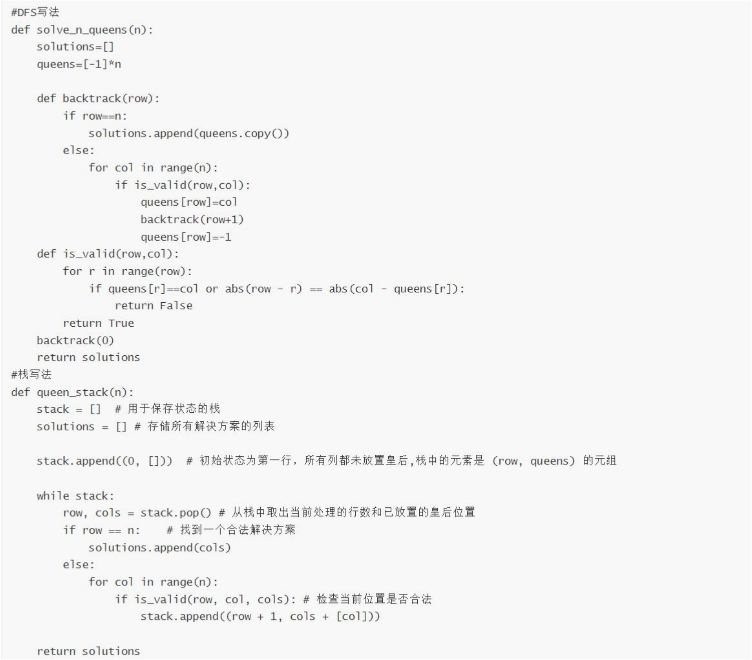
动态中位数：

两个堆，一个模拟左子树，一个模拟右子树：



```python
import heapq
def insert_heap(num):
    if not heap_max or num<=-heap_max[0]:
        heapq.heappush(heap_max,-num)
        if len(heap_max)>len(heap_min)+1:
            heapq.heappush(heap_min,-heap_max[0])
            heapq.heappop(heap_max)

    else:
        heapq.heappush(heap_min,num)
        if len(heap_min)>len(heap_max):
            heapq.heappush(heap_max,-heap_min[0])
            heapq.heappop(heap_min)

n=int(input())
for _ in range(n):
    numlist=list(map(int,input().split()))
    if len(numlist)%2==0:
        print(len(numlist)//2)
    else:
        print((len(numlist)+1)//2)
    heap_max=[]
    heap_min=[]
    res=[]
    for i in range(len(numlist)):
        insert_heap(numlist[i])
        if i%2==0:
            res.append(str(-heap_max[0]))
    aaa=' '.join(res)
    print(aaa)
```

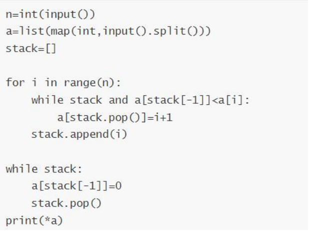
八皇后：



```
#DFS写法
def solve_n_queens(n):
    solutions=[]
    queens=[-1]*n

    def backtrack(row):
        if row==n:
            solutions.append(queens.copy())
        else:
            for col in range(n):
                if is_valid(row,col):
                    queens[row]=col
                    backtrack(row+1)
                    queens[row]=-1
    def is_valid(row,col):
        for r in range(row):
            if queens[r]==col or abs(row - r) == abs(col - queens[r]):
                return False
        return True
    backtrack(0)
    return solutions
#栈写法
def queen_stack(n):
    stack = []  # 用于保存状态的栈
    solutions = [] # 存储所有解决方案的列表

    stack.append((0, []))  # 初始状态为第一行，所有列都未放置皇后,栈中的元素是 (row, queens) 的元组

    while stack:
        row, cols = stack.pop() # 从栈中取出当前处理的行数和已放置的皇后位置
        if row == n:    # 找到一个合法解决方案
            solutions.append(cols)
        else:
            for col in range(n):
                if is_valid(row, col, cols): # 检查当前位置是否合法
                    stack.append((row + 1, cols + [col]))

    return solutions
```

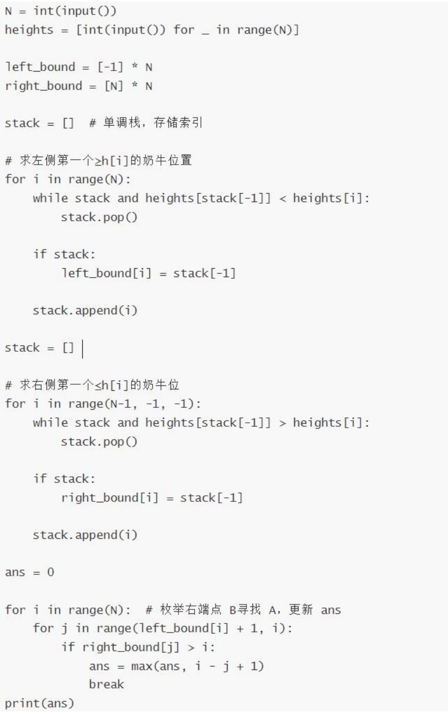
单调栈：

给出项数为 n 的整数数列 a1...an。定义函数 f(i) 代表数列中第 i 个元素之后第一个大于 ai 的元素的**下标**， 。若不存在，则 f(i)=0。试求出 f(1...n)



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

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

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

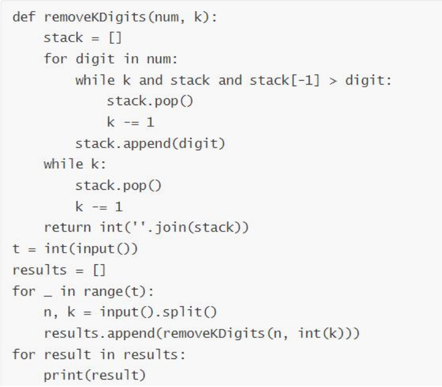
奶牛排队：



```
N = int(input())
heights = [int(input()) for _ in range(N)]

left_bound = [-1] * N
right_bound = [N] * N

stack = []  # 单调栈，存储索引

# 求左侧第一个≥h[i]的奶牛位置
for i in range(N):
    while stack and heights[stack[-1]] < heights[i]:
        stack.pop()

    if stack:
        left_bound[i] = stack[-1]

    stack.append(i)

stack = []

# 求右侧第一个≤h[i]的奶牛位
for i in range(N-1, -1, -1):
    while stack and heights[stack[-1]] > heights[i]:
        stack.pop()

    if stack:
        right_bound[i] = stack[-1]

    stack.append(i)

ans = 0

for i in range(N):  # 枚举右端点 B寻找 A，更新 ans
    for j in range(left_bound[i] + 1, i):
        if right_bound[j] > i:
            ans = max(ans, i - j + 1)
            break
print(ans)
```

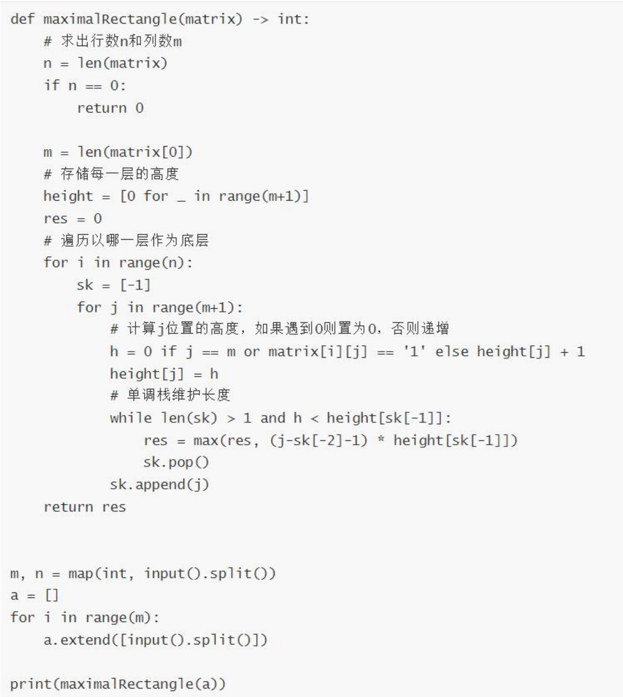
最小新整数：

给定一个十进制正整数 n(0 < n < 1000000000)，每个数位上数字均不为 0。n 的位数为 m。 现在从 m 位中删除 k 位(0<k < m)， 求生成的新整数最小为多少？ 例如: n = 9128456, k = 2, 则生成的新整数最小为 12456



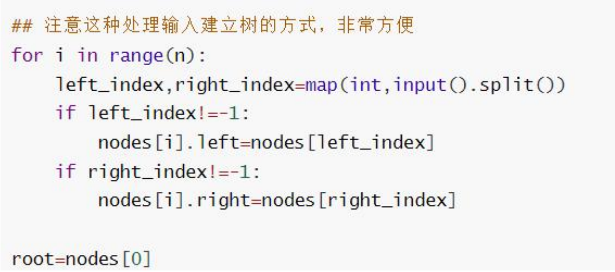
```
def removeKDigits(num, k):
    stack = []
    for digit in num:
        while k and stack and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)
    while k:
        stack.pop()
        k -= 1
    return int(''.join(stack))
t = int(input())
results = []
for _ in range(t):
    n, k = input().split()
    results.append(removeKDigits(n, int(k)))
for result in results:
    print(result)
```

护林员盖房子：在一片保护林中，护林员想要盖一座房子来居住，但他不能砍伐任何树木。现在请你帮他计算：保护林中所能用来盖房子的矩形空地的最大面积。子矩阵边长可以为 1，也就是说： 0 0 0 0 0 依然是一个可以盖房子的子矩阵。



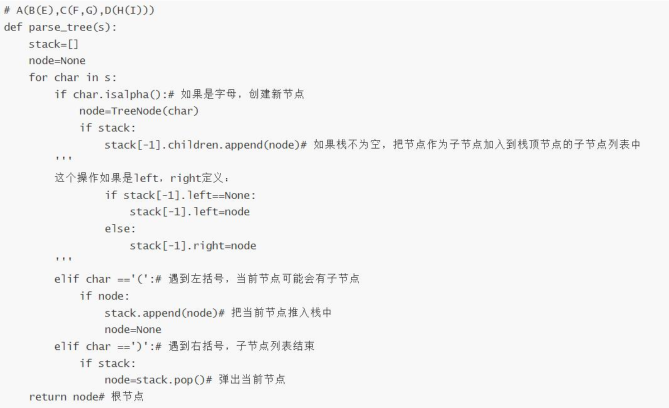
```
def maximalRectangle(matrix) -> int:
    # 求出行数n和列数m
    n = len(matrix)
    if n == 0:
        return 0

    m = len(matrix[0])
    # 存储每一层的高度
    height = [0 for _ in range(m+1)]
    res = 0
    # 遍历以哪一层作为底层
    for i in range(n):
        sk = [-1]
        for j in range(m+1):
            # 计算j位置的高度，如果遇到0则置为0，否则递增
            h = 0 if j == m or matrix[i][j] == '1' else height[j] + 1
            height[j] = h
            # 单调栈维护长度
            while len(sk) > 1 and h < height[sk[-1]]:
                res = max(res, (j-sk[-2]-1) * height[sk[-1]])
                sk.pop()
            sk.append(j)
    return res


m, n = map(int, input().split())
a = []
for i in range(m):
    a.extend([input().split()])

print(maximalRectangle(a))
```

树



```
## 注意这种处理输入建立树的方式，非常方便
for i in range(n):
    left_index,right_index=map(int,input().split())
    if left_index!=-1:
        nodes[i].left=nodes[left_index]
    if right_index!=-1:
        nodes[i].right=nodes[right_index]

root=nodes[0]
```

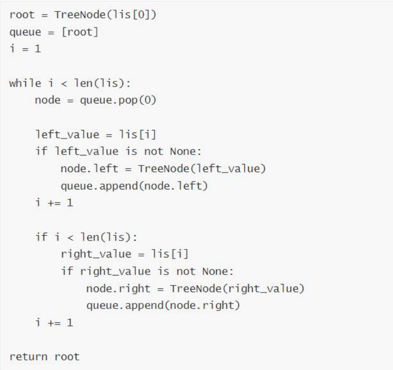
括号嵌套表达式生成树：



```
# A(B(E),C(F,G),D(H(I)))
def parse_tree(s):
    stack=[]
    node=None
    for char in s:
        if char.isalpha():# 如果是字母，创建新节点
            node=TreeNode(char)
            if stack:
                stack[-1].children.append(node)# 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中
        '''
        这个操作如果是left，right定义：
                if stack[-1].left==None:
                    stack[-1].left=node
                else:
                    stack[-1].right=node
        '''
        elif char =='(':# 遇到左括号，当前节点可能会有子节点
            if node:
                stack.append(node)# 把当前节点推入栈中
                node=None
        elif char ==')':# 遇到右括号，子节点列表结束
            if stack:
                node=stack.pop()# 弹出当前节点
    return node# 根节点
```

层次遍历树生成 node 树：

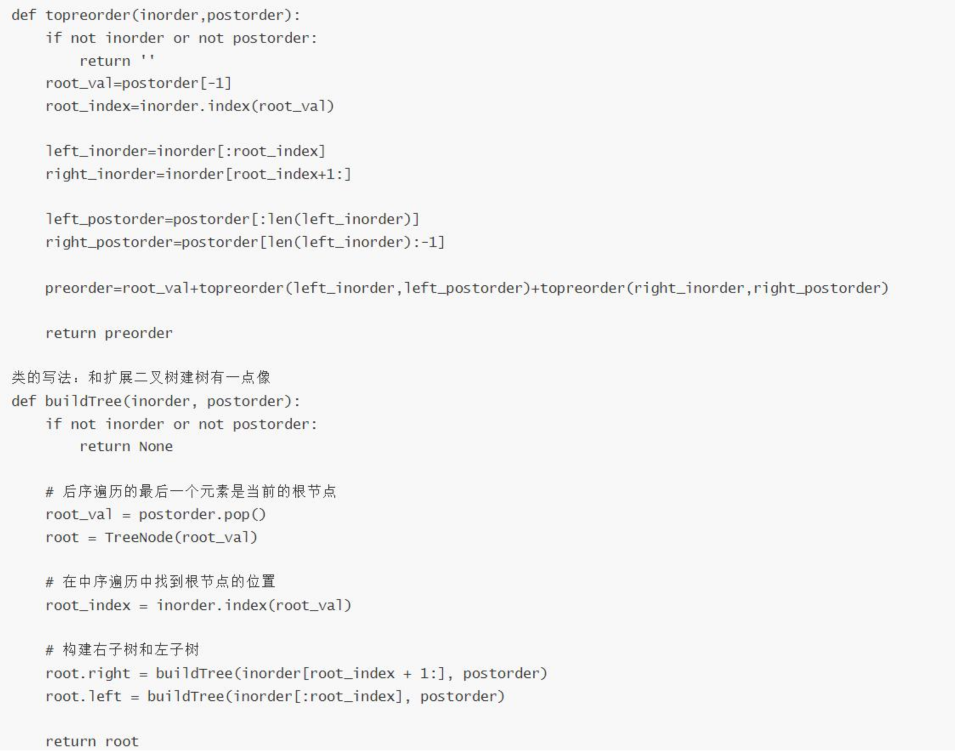
```
def build_tree_list2node(lis):
    if not lis:
        return None
```



```
root = TreeNode(lis[0])
queue = [root]
i = 1

while i < len(lis):
    node = queue.pop(0)

    left_value = lis[i]
    if left_value is not None:
        node.left = TreeNode(left_value)
        queue.append(node.left)
    i += 1

    if i < len(lis):
        right_value = lis[i]
        if right_value is not None:
            node.right = TreeNode(right_value)
            queue.append(node.right)
    i += 1

return root
```

一般扩展二叉树，都可以考虑用递归

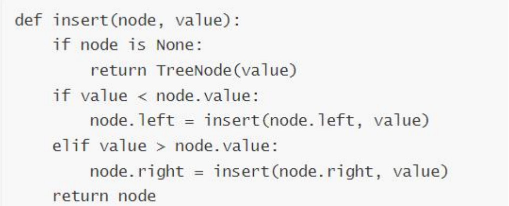
二叉树中序后序，建树：



```
def topreorder(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]

    preorder=root_val+topreorder(left_inorder,left_postorder)+topreorder(right_inorder,right_postorder)

    return preorder

类的写法：和扩展二叉树建树有一点像
def buildTree(inorder, postorder):
    if not inorder or not postorder:
        return None

    # 后序遍历的最后一个元素是当前的根节点
    root_val = postorder.pop()
    root = TreeNode(root_val)

    # 在中序遍历中找到根节点的位置
    root_index = inorder.index(root_val)

    # 构建右子树和左子树
    root.right = buildTree(inorder[root_index + 1:], postorder)
    root.left = buildTree(inorder[:root_index], postorder)

    return root
```

二叉搜索树的中序遍历，就是 sort 排序

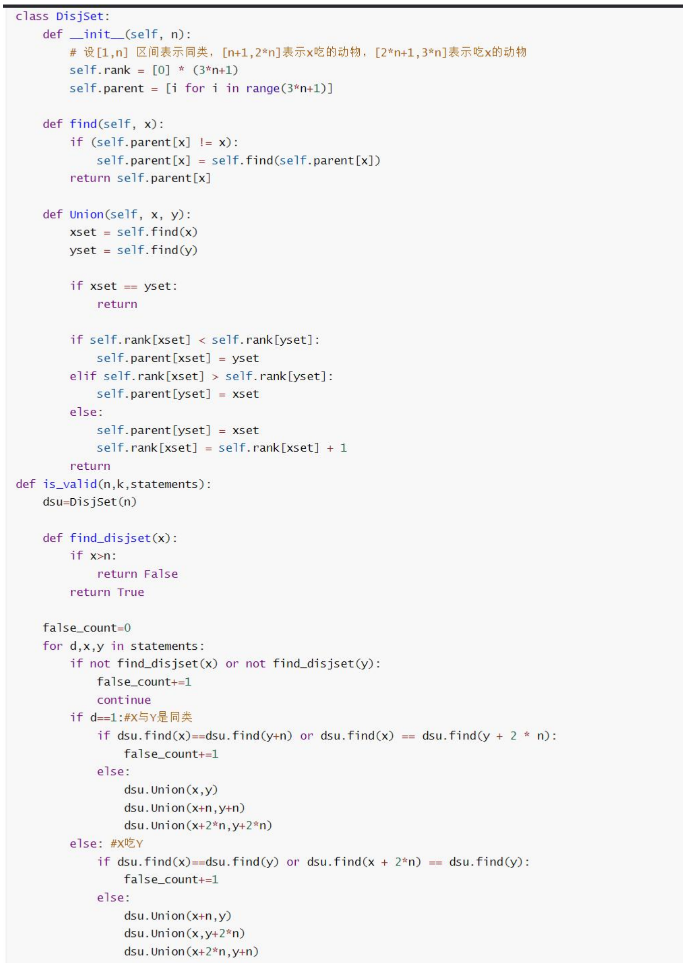
给一行数，建立二叉搜索树：



```
def insert(node, value):
    if node is None:
        return TreeNode(value)
    if value < node.value:
        node.left = insert(node.left, value)
    elif value > node.value:
        node.right = insert(node.right, value)
    return node
```

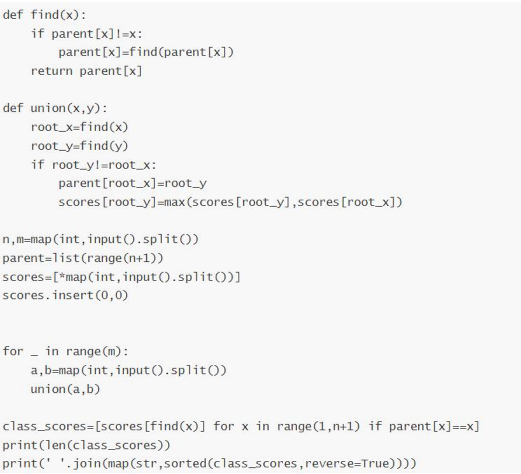
并查集：

食物链：



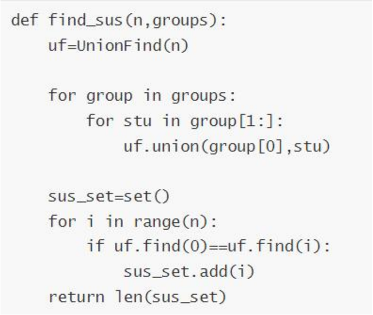
```
class DisjSet:
    def __init__(self, n):
        # 设[1,n] 区间表示同类，[n+1,2*n]表示x吃的动物，[2*n+1,3*n]表示吃x的动物
        self.rank = [0] * (3*n+1)
        self.parent = [i for i in range(3*n+1)]

    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):
        xset = self.find(x)
        yset = self.find(y)

        if xset == yset:
            return

        if self.rank[xset] < self.rank[yset]:
            self.parent[xset] = yset
        elif self.rank[xset] > self.rank[yset]:
            self.parent[yset] = xset
        else:
            self.parent[yset] = xset
            self.rank[xset] = self.rank[xset] + 1
        return
def is_valid(n,k,statements):
    dsu=DisjSet(n)

    def find_disjset(x):
        if x>n:
            return False
        return True

    false_count=0
    for d,x,y in statements:
        if not find_disjset(x) or not find_disjset(y):
            false_count+=1
            continue
        if d==1:#X与Y是同类
            if dsu.find(x)==dsu.find(y+n) or dsu.find(x) == dsu.find(y + 2 * n):
                false_count+=1
            else:
                dsu.Union(x,y)
                dsu.Union(x+n,y+n)
                dsu.Union(x+2*n,y+2*n)
        else: #X吃Y
            if dsu.find(x)==dsu.find(y) or dsu.find(x + 2*n) == dsu.find(y):
                false_count+=1
            else:
                dsu.Union(x+n,y)
                dsu.Union(x,y+2*n)
                dsu.Union(x+2*n,y+n)
```

班级最高分：



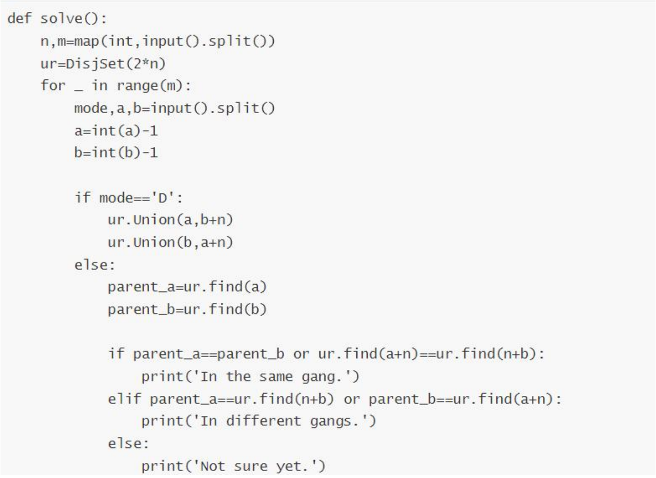
```
def find(x):
    if parent[x]!=x:
        parent[x]=find(parent[x])
    return parent[x]

def union(x,y):
    root_x=find(x)
    root_y=find(y)
    if root_y!=root_x:
        parent[root_x]=root_y
        scores[root_y]=max(scores[root_y],scores[root_x])

n,m=map(int,input().split())
parent=list(range(n+1))
scores=[*map(int,input().split())]
scores.insert(0,0)


for _ in range(m):
    a,b=map(int,input().split())
    union(a,b)

class_scores=[scores[find(x)] for x in range(1,n+1) if parent[x]==x]
print(len(class_scores))
print(' '.join(map(str,sorted(class_scores,reverse=True))))
```

找 suspect：



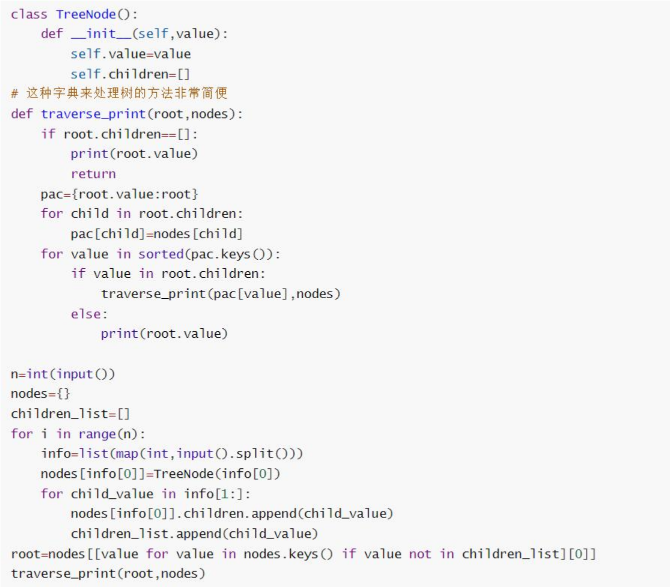
```
def find_sus(n,groups):
    uf=UnionFind(n)

    for group in groups:
        for stu in group[1:]:
            uf.union(group[0],stu)

    sus_set=set()
    for i in range(n):
        if uf.find(0)==uf.find(i):
            sus_set.add(i)
    return len(sus_set)
```

发现他，抓住他：



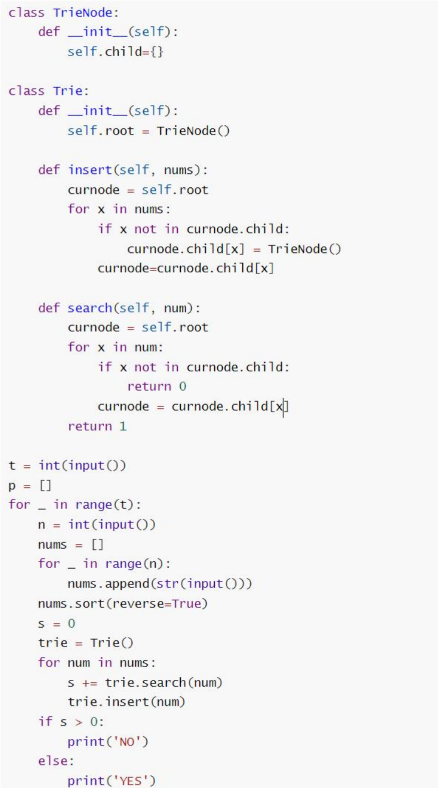
```
def solve():
    n,m=map(int,input().split())
    ur=DisjSet(2*n)
    for _ in range(m):
        mode,a,b=input().split()
        a=int(a)-1
        b=int(b)-1

        if mode=='D':
            ur.Union(a,b+n)
            ur.Union(b,a+n)
        else:
            parent_a=ur.find(a)
            parent_b=ur.find(b)

            if parent_a==parent_b or ur.find(a+n)==ur.find(n+b):
                print('In the same gang.')
            elif parent_a==ur.find(n+b) or parent_b==ur.find(a+n):
                print('In different gangs.')
            else:
                print('Not sure yet.')
```

遍历树，输出是按照当前节点和子节点从小到大的顺序：



```
class TreeNode():
    def __init__(self,value):
        self.value=value
        self.children=[]
# 这种字典来处理树的方法非常简便
def traverse_print(root,nodes):
    if root.children==[]:
        print(root.value)
        return
    pac={root.value:root}
    for child in root.children:
        pac[child]=nodes[child]
    for value in sorted(pac.keys()):
        if value in root.children:
            traverse_print(pac[value],nodes)
        else:
            print(root.value)

n=int(input())
nodes={}
children_list=[]
for i in range(n):
    info=list(map(int,input().split()))
    nodes[info[0]]=TreeNode(info[0])
    for child_value in info[1:]:
        nodes[info[0]].children.append(child_value)
        children_list.append(child_value)
root=nodes[[value for value in nodes.keys() if value not in children_list][0]]
traverse_print(root,nodes)
```

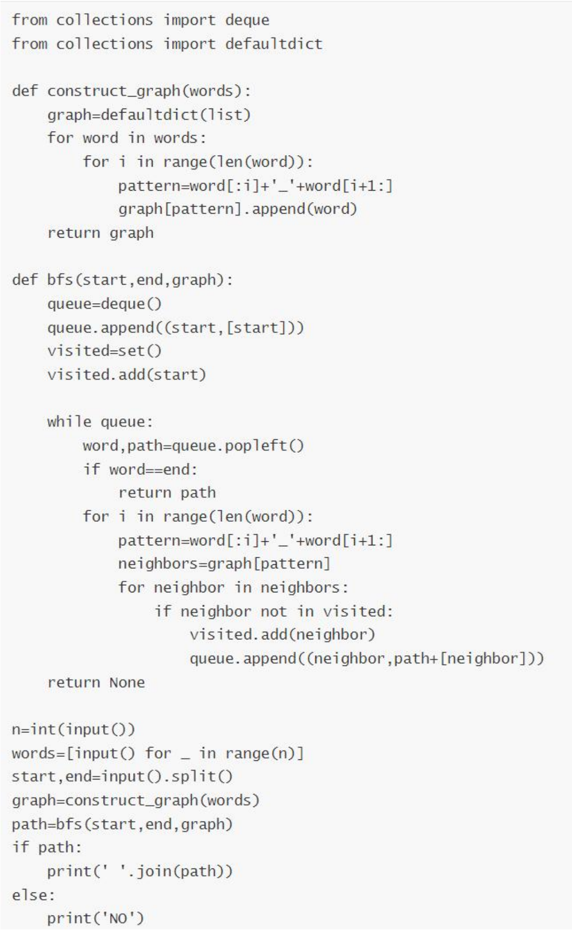
字典树：电话号码查找：



```python
class TrieNode:
    def __init__(self):
        self.child={}

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, nums):
        curnode = self.root
        for x in nums:
            if x not in curnode.child:
                curnode.child[x] = TrieNode()
            curnode=curnode.child[x]

    def search(self, num):
        curnode = self.root
        for x in num:
            if x not in curnode.child:
                return 0
            curnode = curnode.child[x]
        return 1

t = int(input())
p = []
for _ in range(t):
    n = int(input())
    nums = []
    for _ in range(n):
        nums.append(str(input()))
    nums.sort(reverse=True)
    s = 0
    trie = Trie()
    for num in nums:
        s += trie.search(num)
        trie.insert(num)
    if s > 0:
        print('NO')
    else:
        print('YES')
```

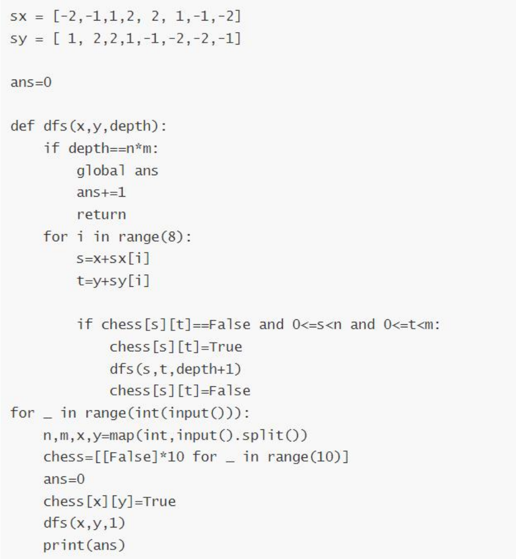
图

词梯：BFS



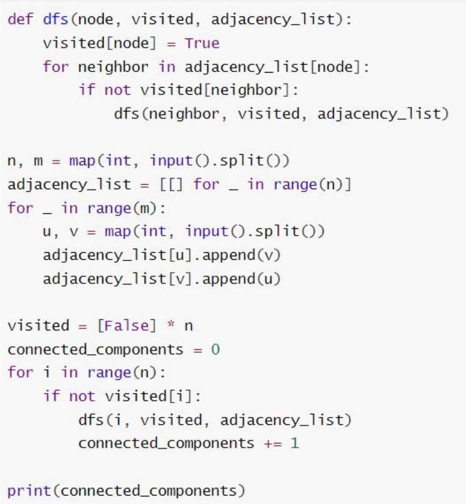
```python
from collections import deque
from collections import defaultdict

def construct_graph(words):
    graph=defaultdict(list)
    for word in words:
        for i in range(len(word)):
            pattern=word[:i]+'_'+word[i+1:]
            graph[pattern].append(word)
    return graph

def bfs(start,end,graph):
    queue=deque()
    queue.append((start,[start]))
    visited=set()
    visited.add(start)

    while queue:
        word,path=queue.popleft()
        if word==end:
            return path
        for i in range(len(word)):
            pattern=word[:i]+'_'+word[i+1:]
            neighbors=graph[pattern]
            for neighbor in neighbors:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor,path+[neighbor]))
    return None

n=int(input())
words=[input() for _ in range(n)]
start,end=input().split()
graph=construct_graph(words)
path=bfs(start,end,graph)
if path:
    print(' '.join(path))
else:
    print('NO')
```

马走日：DFS



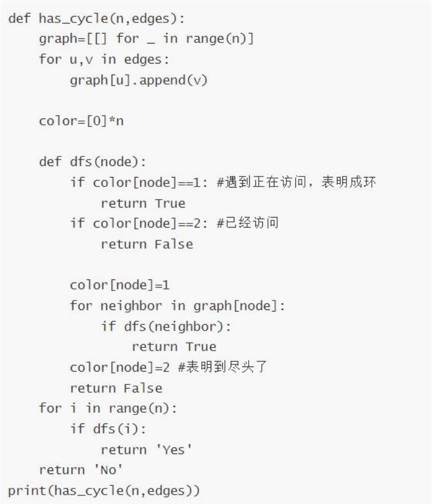
```
sx = [-2,-1,1,2, 2, 1,-1,-2]
sy = [ 1, 2,2,1,-1,-2,-2,-1]

ans=0

def dfs(x,y,depth):
    if depth==n*m:
        global ans
        ans+=1
        return
    for i in range(8):
        s=x+sx[i]
        t=y+sy[i]

        if chess[s][t]==False and 0<=s<n and 0<=t<m:
            chess[s][t]=True
            dfs(s,t,depth+1)
            chess[s][t]=False
for _ in range(int(input())):
    n,m,x,y=map(int,input().split())
    chess=[[False]*10 for _ in range(10)]
    ans=0
    chess[x][y]=True
    dfs(x,y,1)
    print(ans)
```

无向图联通快个数：



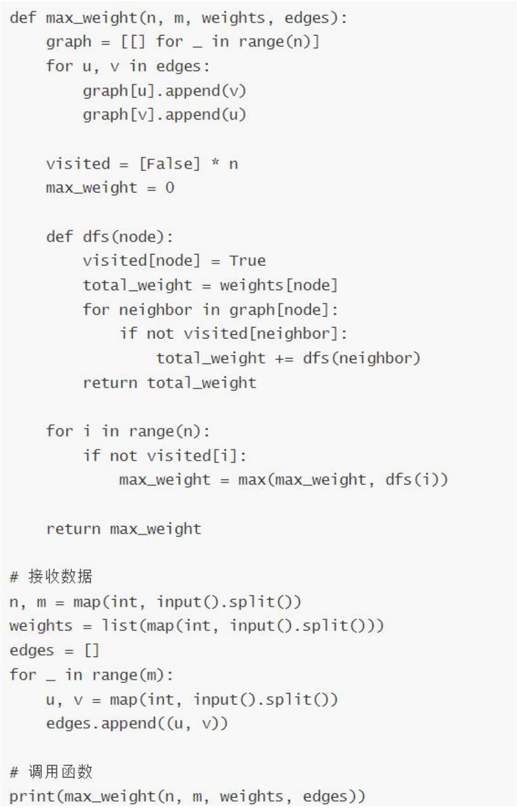
```
def dfs(node, visited, adjacency_list):
    visited[node] = True
    for neighbor in adjacency_list[node]:
        if not visited[neighbor]:
            dfs(neighbor, visited, adjacency_list)

n, m = map(int, input().split())
adjacency_list = [[] for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adjacency_list[u].append(v)
    adjacency_list[v].append(u)

visited = [False] * n
connected_components = 0
for i in range(n):
    if not visited[i]:
        dfs(i, visited, adjacency_list)
        connected_components += 1

print(connected_components)
```

有向图判环：



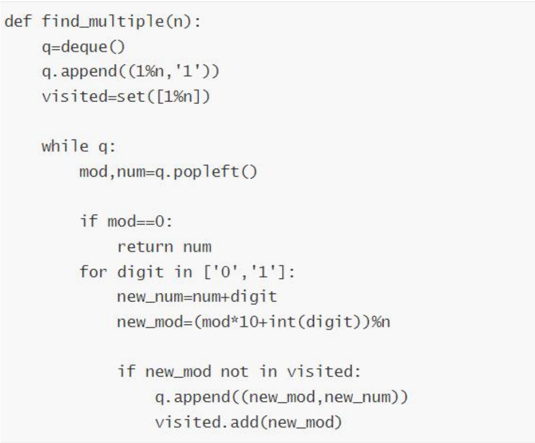
```
def has_cycle(n,edges):
    graph=[[] for _ in range(n)]
    for u,v in edges:
        graph[u].append(v)

    color=[0]*n

    def dfs(node):
        if color[node]==1: #遇到正在访问，表明成环
            return True
        if color[node]==2: #已经访问
            return False

        color[node]=1
        for neighbor in graph[node]:
            if dfs(neighbor):
                return True
        color[node]=2 #表明到尽头了
        return False
    for i in range(n):
        if dfs(i):
            return 'Yes'
    return 'No'
print(has_cycle(n,edges))
```

无向图最大权值联通块：



```
def max_weight(n, m, weights, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = [False] * n
    max_weight = 0

    def dfs(node):
        visited[node] = True
        total_weight = weights[node]
        for neighbor in graph[node]:
            if not visited[neighbor]:
                total_weight += dfs(neighbor)
        return total_weight

    for i in range(n):
        if not visited[i]:
            max_weight = max(max_weight, dfs(i))

    return max_weight

# 接收数据
n, m = map(int, input().split())
weights = list(map(int, input().split()))
edges = []
for _ in range(m):
    u, v = map(int, input().split())
    edges.append((u, v))

# 调用函数
print(max_weight(n, m, weights, edges))
```

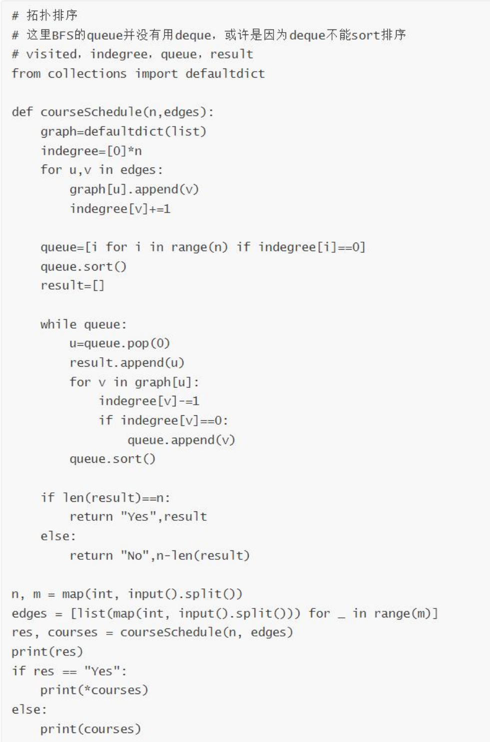
找 0-1 组成的倍数：



```
def find_multiple(n):
    q=deque()
    q.append((1%n,'1'))
    visited=set([1%n])

    while q:
        mod,num=q.popleft()

        if mod==0:
            return num
        for digit in ['0','1']:
            new_num=num+digit
            new_mod=(mod*10+int(digit))%n

            if new_mod not in visited:
                q.append((new_mod,new_num))
                visited.add(new_mod)
```

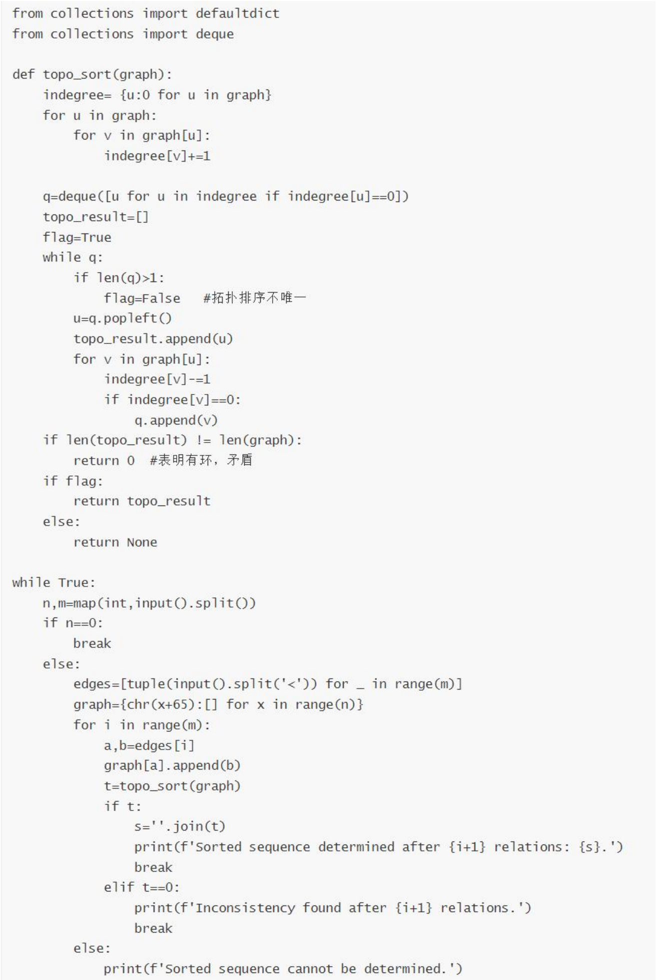
拓扑排序：

先导课程：



```python
# 拓扑排序
# 这里BFS的queue并没有用deque，或许是因为deque不能sort排序
# visited，indegree，queue，result
from collections import defaultdict

def courseSchedule(n,edges):
    graph=defaultdict(list)
    indegree=[0]*n
    for u,v in edges:
        graph[u].append(v)
        indegree[v]+=1

    queue=[i for i in range(n) if indegree[i]==0]
    queue.sort()
    result=[]

    while queue:
        u=queue.pop(0)
        result.append(u)
        for v in graph[u]:
            indegree[v]-=1
            if indegree[v]==0:
                queue.append(v)
        queue.sort()

    if len(result)==n:
        return "Yes",result
    else:
        return "No",n-len(result)

n, m = map(int, input().split())
edges = [list(map(int, input().split())) for _ in range(m)]
res, courses = courseSchedule(n, edges)
print(res)
if res == "Yes":
    print(*courses)
else:
    print(courses)
```

Sorting it all out：想清楚是中途判断的任务还是结尾判断



```python
from collections import defaultdict
from collections import deque

def topo_sort(graph):
    indegree= {u:0 for u in graph}
    for u in graph:
        for v in graph[u]:
            indegree[v]+=1

    q=deque([u for u in indegree if indegree[u]==0])
    topo_result=[]
    flag=True
    while q:
        if len(q)>1:
            flag=False   #拓扑排序不唯一
        u=q.popleft()
        topo_result.append(u)
        for v in graph[u]:
            indegree[v]-=1
            if indegree[v]==0:
                q.append(v)
    if len(topo_result) != len(graph):
        return 0  #表明有环，矛盾
    if flag:
        return topo_result
    else:
        return None

while True:
    n,m=map(int,input().split())
    if n==0:
        break
    else:
        edges=[tuple(input().split('<')) for _ in range(m)]
        graph={chr(x+65):[] for x in range(n)}
        for i in range(m):
            a,b=edges[i]
            graph[a].append(b)
            t=topo_sort(graph)
            if t:
                s=''.join(t)
                print(f'Sorted sequence determined after {i+1} relations: {s}.')
                break
            elif t==0:
                print(f'Inconsistency found after {i+1} relations.')
                break
        else:
            print(f'Sorted sequence cannot be determined.')
```

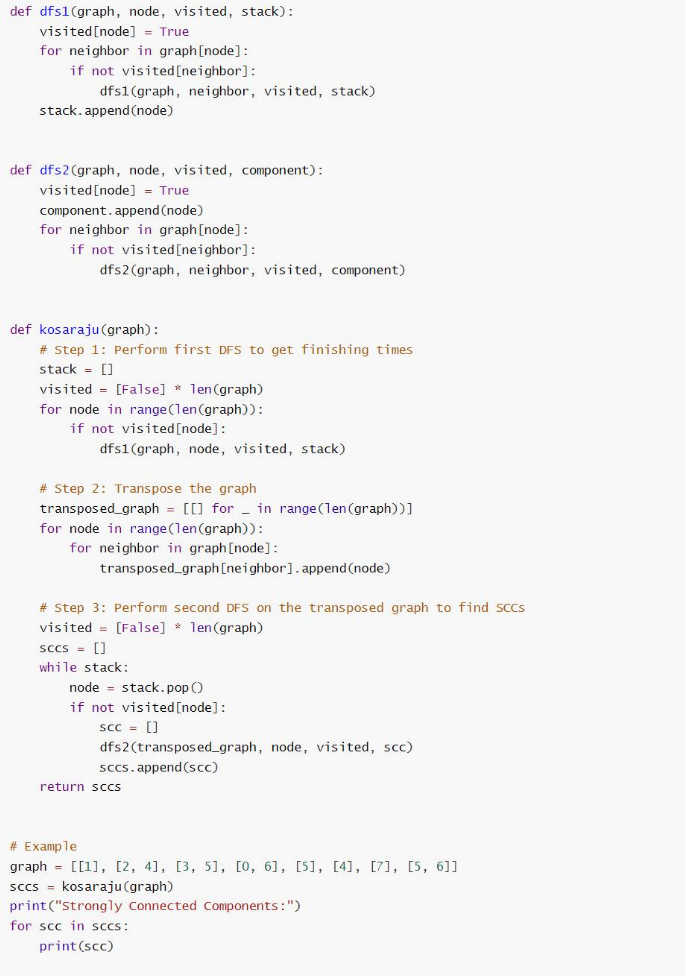
强连通单元：

Kosaraju 算法的核心思想就是两次深度优先搜索（DFS）

1. **第一次 DFS**：在第一次 DFS 中，我们对图进行标准的深度优先搜索，但是在此过程中，我们记录下顶点完成搜索的顺序。这一步的目的是为了找出每个顶点的完成时间（即结束时间）。

2. **反向图**：接下来，我们对原图取反，即将所有的边方向反转，得到反向图。

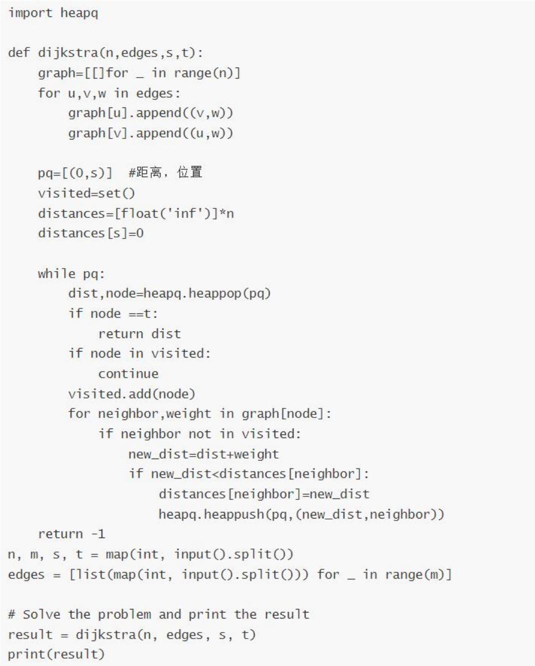
3. **第二次 DFS**：在第二次 DFS 中，我们按照第一步中记录的顶点完成时间的逆序，对反向图进行 DFS。这样，我们将找出反向图中的强连通分量。



```python
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)


def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)


def kosaraju(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs


# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)
```

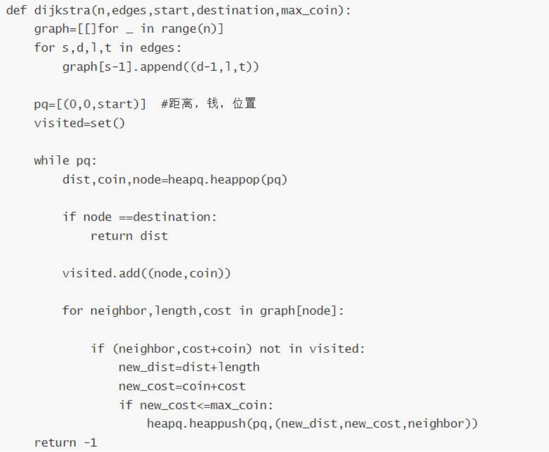
Dijkstra 用到了优先队列，注意一定要把 weight 放在 vertex 前面

最小生成树：MSTs **如果是稠密图(边多)，则用 prim 算法;如果是稀疏图(边少)，则用 kruskal 算法**。



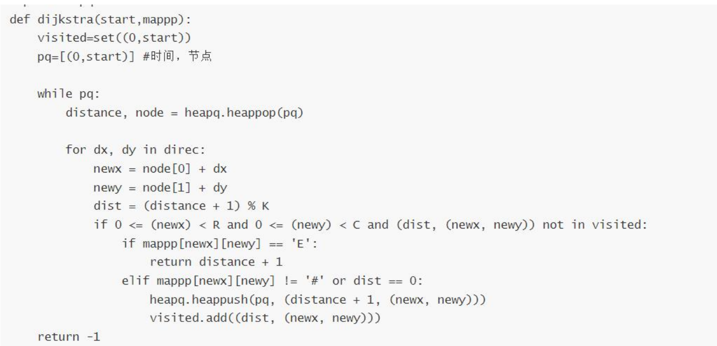
```
import heapq

def dijkstra(n,edges,s,t):
    graph=[[]for _ in range(n)]
    for u,v,w in edges:
        graph[u].append((v,w))
        graph[v].append((u,w))

    pq=[(0,s)]  #距离，位置
    visited=set()
    distances=[float('inf')]*n
    distances[s]=0

    while pq:
        dist,node=heapq.heappop(pq)
        if node ==t:
            return dist
        if node in visited:
            continue
        visited.add(node)
        for neighbor,weight in graph[node]:
            if neighbor not in visited:
                new_dist=dist+weight
                if new_dist<distances[neighbor]:
                    distances[neighbor]=new_dist
                    heapq.heappush(pq,(new_dist,neighbor))
    return -1
n, m, s, t = map(int, input().split())
edges = [list(map(int, input().split())) for _ in range(m)]

# Solve the problem and print the result
result = dijkstra(n, edges, s, t)
print(result)
```

不能超过多少钱的最短距离：



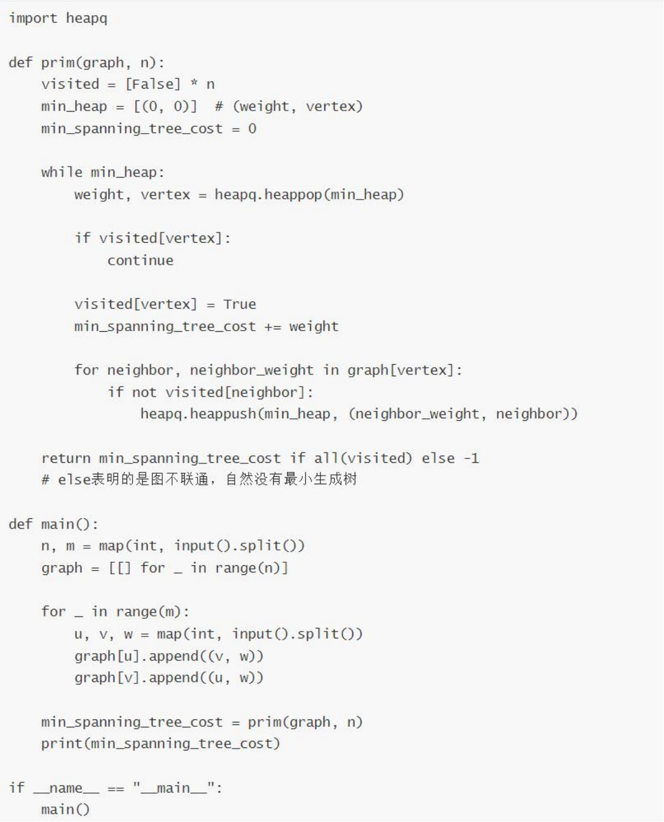
```
def dijkstra(n,edges,start,destination,max_coin):
    graph=[[]for _ in range(n)]
    for s,d,l,t in edges:
        graph[s-1].append((d-1,l,t))

    pq=[(0,0,start)]  #距离，钱，位置
    visited=set()

    while pq:
        dist,coin,node=heapq.heappop(pq)

        if node ==destination:
            return dist

        visited.add((node,coin))

        for neighbor,length,cost in graph[node]:

            if (neighbor,cost+coin) not in visited:
                new_dist=dist+length
                new_cost=coin+cost
                if new_cost<=max_coin:
                    heapq.heappush(pq,(new_dist,new_cost,neighbor))
    return -1
```

变换迷宫：



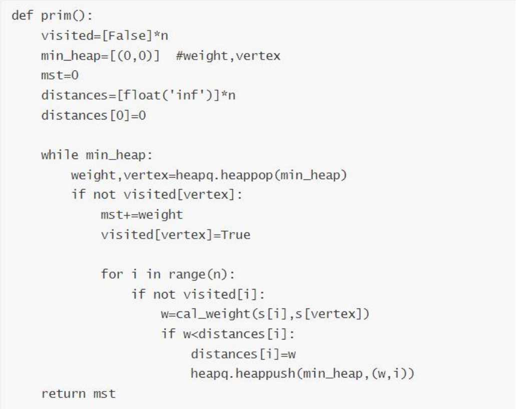
```
def dijkstra(start,mappp):
    visited=set((0,start))
    pq=[(0,start)] #时间，节点

    while pq:
        distance, node = heapq.heappop(pq)

        for dx, dy in direc:
            newx = node[0] + dx
            newy = node[1] + dy
            dist = (distance + 1) % K
            if 0 <= (newx) < R and 0 <= (newy) < C and (dist, (newx, newy)) not in visited:
                if mappp[newx][newy] == 'E':
                    return distance + 1
                elif mappp[newx][newy] != '#' or dist == 0:
                    heapq.heappush(pq, (distance + 1, (newx, newy)))
                    visited.add((dist, (newx, newy)))
    return -1
```

最小生成树：MSTs **如果是稠密图(边多)，则用 prim 算法;如果是稀疏图(边少)，则用 kruskal 算法**。

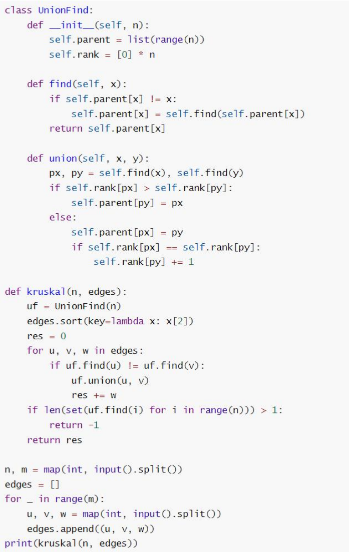


```
import heapq

def prim(graph, n):
    visited = [False] * n
    min_heap = [(0, 0)]  # (weight, vertex)
    min_spanning_tree_cost = 0

    while min_heap:
        weight, vertex = heapq.heappop(min_heap)

        if visited[vertex]:
            continue

        visited[vertex] = True
        min_spanning_tree_cost += weight

        for neighbor, neighbor_weight in graph[vertex]:
            if not visited[neighbor]:
                heapq.heappush(min_heap, (neighbor_weight, neighbor))

    return min_spanning_tree_cost if all(visited) else -1
    # else表明的是图不联通，自然没有最小生成树

def main():
    n, m = map(int, input().split())
    graph = [[] for _ in range(n)]

    for _ in range(m):
        u, v, w = map(int, input().split())
        graph[u].append((v, w))
        graph[v].append((u, w))

    min_spanning_tree_cost = prim(graph, n)
    print(min_spanning_tree_cost)

if __name__ == "__main__":
    main()
```

卡车：



```
def prim():
    visited=[False]*n
    min_heap=[(0,0)]  #weight,vertex
    mst=0
    distances=[float('inf')]*n
    distances[0]=0

    while min_heap:
        weight,vertex=heapq.heappop(min_heap)
        if not visited[vertex]:
            mst+=weight
            visited[vertex]=True

            for i in range(n):
                if not visited[i]:
                    w=cal_weight(s[i],s[vertex])
                    if w<distances[i]:
                        distances[i]=w
                        heapq.heappush(min_heap,(w,i))
    return mst
```



```
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 kruskal(n, edges):
    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

n, m = map(int, input().split())
edges = []
for _ in range(m):
    u, v, w = map(int, input().split())
    edges.append((u, v, w))
print(kruskal(n, edges))
```