1、根据二叉树前序中序序列建树

描述

假设二叉树的节点里包含一个大写字母，每个节点的字母都不同。

给定二叉树的前序遍历序列和中序遍历序列(长度均不超过 26)，请输出该二叉树的后序遍历序列

输入

多组数据

每组数据 2 行，第一行是前序遍历序列，第二行是中序遍历序列

输出

对每组序列建树，输出该树的后序遍历序列

```
def postorder(preorder,inorder):
    if not preorder or not inorder:
        return []
    root=preorder[0]    #前序遍历第一个是根
    k=inorder.index(root)    #中序遍历找到根，前面是左子树，后面是右子树
    left_inorder=inorder[:k]
    right_inorder=inorder[k+1:]
    left_preorder=preorder[1:len(left_inorder)+1]
    right_preorder=preorder[len(left_inorder)+1:]
    #递归
    left_postorder=postorder(left_preorder,left_inorder)
    right_postorder=postorder(right_preorder,right_inorder)
    return left_postorder+right_postorder+[root]
while True:
    try:
        L=list(input())
        S=list(input())
        print(''.join(postorder(L,S)))
    except EOFError:
        break
```

2、括号嵌套二叉树

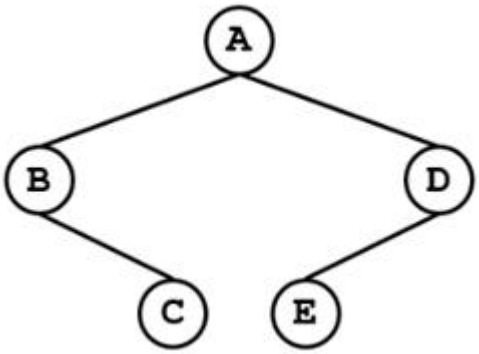
描述

可以用括号嵌套的方式来表示一棵二叉树。方法如下：

- '*'表示空的二叉树
- 如果一棵二叉树只有一个结点，则该树就用一个非'*'字符表示，代表其根结点。
- 如果一棵二叉左右子树都非空，则用“树根(左子树,右子树)”的形式表示。树根是一个非'*'字符,左右子树之间用逗号隔开，没有空格。左右子树都用括号嵌套法表示。如果左子树非空而右子树为空，则用“树根(左子树)”形式表示；如果左子树为空而右子树非空，则用“树根(*,右子树)”形式表示。

给出一棵树的括号嵌套表示形式，请输出其前序遍历序列、中序遍历序列、后序遍历序列。

例如，"A(B(*,C),D(E))"表示的二叉树如图所示



输入

第一行是整数 n 表示有 n 棵二叉树(n<100)

接下来有 n 行，每行是 1 棵二叉树的括号嵌套表示形式

输出

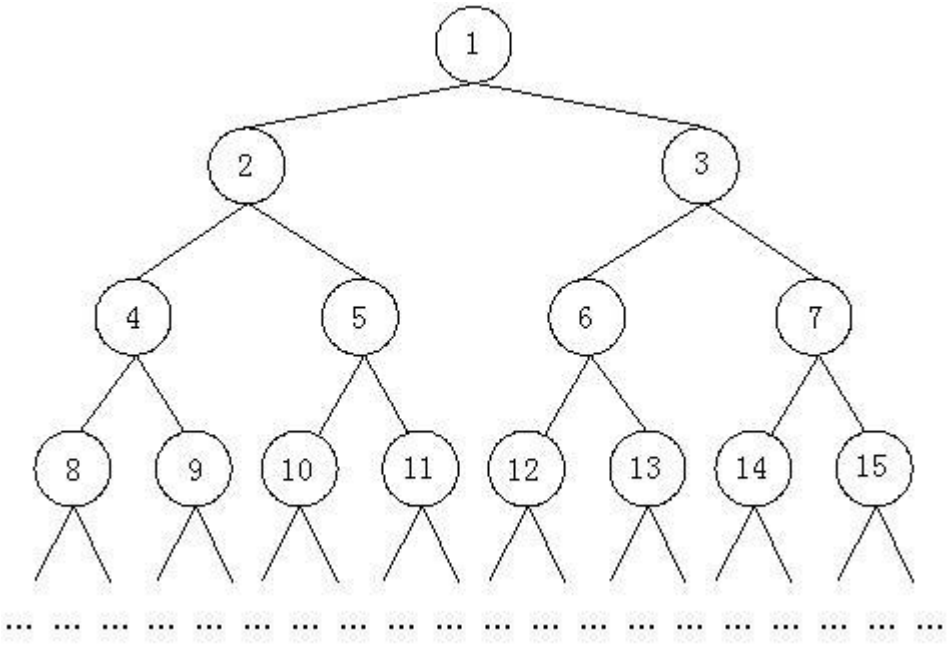
对每棵二叉树，输出其前序遍历序列和中序遍历序列

```
n=int(input())
P=[]for i in range (n):
    L=str(input())
    #现根据括号形式构建出二叉树

    def build_tree(s):
        if s=='*':
            return None
        if '(' not in s:
            return [s,None,None]
        root=s[0]
        left=''
        right=''
        del_root=s[2:-1]
        k=0
        a=0
        for i,char in enumerate(del_root):
            if char=='(':
                k=k+1
            elif char==')':
                k=k-1
            elif char==',' and k==0:
                a=i
                break
            #找中间点了
        if a==0:
            left=del_root
        else:
```

```
                left=del_root[:a]
                right=del_root[a+1:]
            return [root,build_tree(left),build_tree(right)]

        K=build_tree(L)
        S=K[:]

        def preorder(tree):
            if not tree:
                return ''
            return tree[0]+preorder(tree[1])+preorder(tree[2])

        pre=preorder(K)
        def inorder(tree):
            if not tree:
                return ''
            return inorder(tree[1])+tree[0]+inorder(tree[2])

        ino=inorder(K)
        P.append(pre)
        P.append(ino)
for i in P:
        print(i)
```

3、二叉树

描述



如上图所示，由正整数 1，2，3……组成了一颗二叉树。我们已知这个二叉树的最后一个结点是 n。现在的问题是，结点 m 所在的子树中一共包括多少个结点。

比如，n = 12，m = 3 那么上图中的结点 13，14，15 以及后面的结点都是不存在的，结点 m 所在子树中包括的结点有 3，6，7，12，因此结点 m 的所在子树中共有 4 个结点。

输入

输入数据包括多行，每行给出一组测试数据，包括两个整数 m，n (1 <= m <= n <= 1000000000)。最后一组测试数据中包括两个 0，表示输入的结束，这组数据不用处理。

输出

对于每一组测试数据，输出一行，该行包含一个整数，给出结点 m 所在子树中包括的结点的数目。

```
def count(m,n):
    left,right=m,m
    count=0
    while left<=n:
        current_count=min(right,n)-left+1
        count=count+current_count
        left=left*2
        right=right*2+1
    return count
while True:
    m,n=map(int,input().split())
    if m==0 and n==0:
        break
    print(count(m,n))
```

4、二叉搜索树的层次遍历

描述

二叉搜索树在动态查表中有特别的用处，一个无序序列可以通过构造一棵二叉搜索树变成一个有序序列，

构造树的过程即为对无序序列进行排序的过程。每次插入的新的结点都是二叉搜索树上新的叶子结点，在进行

插入操作时，不必移动其它结点，只需改动某个结点的指针，由空变为非空即可。

这里，我们想探究二叉树的建立和层次输出。

输入

只有一行，包含若干个数字，中间用空格隔开。（数字可能会有重复，对于重复的数字，只计入一个）

输出

输出一行，对输入数字建立二叉搜索树后进行按层次周游的结果。

```
from collections import dequeclass TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, val):
        if not self.root:
            self.root = TreeNode(val)
        else:
            self._insert(self.root, val)

    def _insert(self, node, val):
        if val < node.val:
            if node.left is None:
                node.left = TreeNode(val)
            else:
                self._insert(node.left, val)
        else:
            if node.right is None:
                node.right = TreeNode(val)
            else:
                self._insert(node.right, val)

    def search(self, val):
        return self._search(self.root, val)

    def _search(self, node, val):
        if node is None or node.val == val:
            return node
        if val < node.val:
            return self._search(node.left, val)
        return self._search(node.right, val)

    def level_order(self):
        if not self.root:
            return []
        result=[]
        queue=deque([self.root])
        while queue:
            #持续遍历该层的子节点，向下得到下一层的节点
```

```
                node=queue.popleft()
                result.append(node.val)
                if node.left:
                        queue.append(node.left)
                if node.right:
                        queue.append(node.right)
        return result     #BFS 方法，把函数写在后面也一样

L=list(map(int,input().split( )))
tree=BinarySearchTree()for i in L:
    if not tree.search(i):
        tree.insert(i)
L=tree.level_order()for i in range(len(L)-1):
    print(L[i],end=' ')print(L[-1])
```

5、实现堆结构

描述

定义一个数组，初始化为空。在数组上执行两种操作：

1、增添 1 个元素，把 1 个新的元素放入数组。

2、输出并删除数组中最小的数。

使用堆结构实现上述功能的高效算法。

输入

第一行输入一个整数 t，代表测试数据的组数。

对于每组测试数据，第一行输入一个整数 n，代表操作的次数。

每次操作首先输入一个整数 type。

当 type=1，增添操作，接着输入一个整数 u，代表要插入的元素。

当 type=2，输出删除操作，输出并删除数组中最小的元素。

1<=n<=100000。

输出

每次删除操作输出被删除的数字。

```
class BinHeap:
    def __init__(self):
        self.heapList=[0]
        self.currentSize=0

    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
```

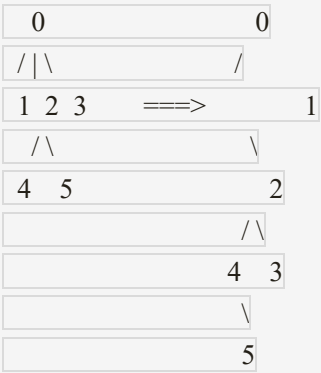
```
    def insert(self,k):
        self.heapList.append(k)
        self.currentSize=self.currentSize+1
        self.percUp(self.currentSize)

    def percDown(self,i):
        while i*2<=self.currentSize:
            mc=self.minChild(i)
            if self.heapList[i]>self.heapList[mc]:
                tmp=self.heapList[i]
                self.heapList[i]=self.heapList[mc]
                self.heapList[mc]=tmp
            i=mc

    def minChild(self,i):
        if i*2+1>self.currentSize:
            return i*2
        else:
            if self.heapList[i*2]<self.heapList[i*2+1]:
                return i*2
            else:
                return 2*i+1

    def delMin(self):
        retval=self.heapList[1]
        self.heapList[1]=self.heapList[self.currentSize]
        self.currentSize=self.currentSize-1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def buildHeap(self,alist):
        i=len(alist)//2
        self.currentSize=len(alist)
        self.heapList=[0]+alist[:]
        print(len(self.heapList),i)
        while i>0:
            print(self.heapList,i)
            self.percDown(i)
            i=i-1
            print(self.heapList,i)
L=[]
t=int(input())for i in range(t):
```

```
    n=int(input())
    heap=BinHeap()
    for i in range(n):
        a=list(map(int,input().split()))
        if a[0]==1:
            heap.insert(a[1])
        if a[0]==2:
            c=heap.delMin()
            L.append(c)for i in L:
    print(i)
```

6、树的转换

描述

我们都知道用“左儿子右兄弟”的方法可以将一棵一般的树转换为二叉树，如：



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。

输入

输入是一个由“u”和“d”组成的字符串，表示一棵树的深度优先搜索信息。比如，dudduduudu可以用来表示上文中的左树，因为搜索过程为：0 Down to 1 Up to 0 Down to 2 Down to 4 Up to 2 Down to 5 Up to 2 Up to 0 Down to 3 Up to 0。

你可以认为每棵树的结点数至少为 2，并且不超过 10000。

输出

按如下格式输出转换前和转换后树的高度：

h1 => h2

其中，h1 是转换前树的高度，h2 是转换后树的高度。

```
class TreeNode:
    def __init__(self):
        self.children=[]
        self.left=None
        self.right=None
def build_tree1(s):
    L=[]
    root=TreeNode()
```

```
    L.append(root)
    for i in s:
        if i=='d':
            new_node=TreeNode()
            L[-1].children.append(new_node)
            L.append(new_node)
        else:
            L.pop()
    return root
def build_tree2(root):
    if not root:
        return None
    tree=TreeNode()

    if root.children:
        tree.left=build_tree2(root.children[0])
        #left 是第一个兄弟节点
        curr=tree.left
        for i in root.children[1:]:
            curr.right=build_tree2(i)
            curr=curr.right
            #用 right 串联所有兄弟节点
    return tree
def height(node):
    if not node:
        return 0
    return 1+max(height(node.left),height(node.right))

s=str(input())
h1=0
L=[]for i in range (len(s)):
    if s[i]=='d':
        h1=h1+1
    else:
        h1=h1-1
    L.append(h1)
h1=max(L)

tree1=build_tree1(s)
tree2=build_tree2(tree1)
h2=height(tree2)-1print(f'{h1} => {h2}')
```

7、Huffman 编码树

描述

构造一个具有 n 个外部节点的扩充二叉树，每个外部节点 Ki 有一个 Wi 对应，作为该外部节点的权。使得这个扩充二叉树的叶节点带权外部路径长度总和最小：

Min(W1 * L1 + W2 * L2 + W3 * L3 + … + Wn * Ln)

Wi:每个节点的权值。

Li:根节点到第 i 个外部叶子节点的距离。

编程计算最小外部路径长度总和。

输入

第一行输入一个整数 t，代表测试数据的组数。

对于每组测试数据，第一行输入一个整数 n，外部节点的个数。第二行输入 n 个整数，代表各个外部节点的权值。

2<=N<=100

输出

输出最小外部路径长度总和。

```
n=int(input())
L=[]for i in range (n):
    t=int(input())
    Lt=list(map(int,input().split()))
    val=0
    while len(Lt)>1:
        a=min(Lt)
        b=Lt.pop(Lt.index(a))
        c=min(Lt)
        d=Lt.pop(Lt.index(c))
        Lt.append(b+d)
        val=val+b+d
    L.append(val)for i in L:
    print(i)
```

8、宗教信仰

描述

世界上有许多宗教，你感兴趣的是你学校里的同学信仰多少种宗教。

你的学校有 n 名学生（0 < n <= 50000），你不太可能询问每个人的宗教信仰，因为他们不太愿意透露。但是当你同时找到 2 名学生，他们却愿意告诉你他们是否信仰同一宗教，你可以通过很多这样的询问估算学校里的宗教数目的上限。你可以认为每名学生只会信仰最多一种宗教。

输入

输入包括多组数据。

每组数据的第一行包括 n 和 m，0 <= m <= n(n-1)/2，其后 m 行每行包括两个数字 i 和 j，表示学生 i 和学生 j 信仰同一宗教，学生被标号为 1 至 n。输入以一行 n = m = 0 作为结束。

输出

对于每组数据，先输出它的编号（从 1 开始），接着输出学生信仰的不同宗教的数目上限。

```
def GetRoot(a):
    if parent[a]!=a:
        parent[a]=GetRoot(parent[a])
    return parent[a]
def Merge(a,b):
    p1=GetRoot(a)
    p2=GetRoot(b)
    if p1==p2:
        return
    parent[p2]=p1

c=1while True:
    n,m=list(map(int,input().split( )))
    if n==0 and m==0:
        break
    parent=[i for i in range (n+1)]
    for i in range (m):
        a,b=map(int,input().split( ))
        Merge(a,b)

    s=set()
    for i in range (1,n+1):
        s.add(GetRoot(i))

    print(f'Case {c}: {len(s)}')
    c=c+1
```

9、兔子与樱花

描述

很久很久之前，森林里住着一群兔子。有一天，兔子们希望去赏樱花，但当他们到了上野公园门口却忘记了带地图。现在兔子们想求助于你来帮他们找到公园里的最短路。

输入

输入分为三个部分。

第一个部分有 P+1 行（P<30），第一行为一个整数 P，之后的 P 行表示上野公园的地点，字符串长度不超过 20。

第二个部分有 Q+1 行（Q<50），第一行为一个整数 Q，之后的 Q 行每行分别为两个字符串

与一个整数，表示这两点有直线的道路，并显示二者之间的矩离（单位为米）。
第三个部分有 R+1 行（R<20），第一行为一个整数 R，之后的 R 行每行为两个字符串，表示需要求的路线。

输出

输出有 R 行，分别表示每个路线最短的走法。其中两个点之间，用->(矩离)->相隔。

```
from collections import defaultdictimport heapq

d=defaultdict(list)
list_1=[]
P=int(input())for i in range (P):
    x=str(input())
    list_1.append(x)

Q=int(input())for i in range (Q):
    u,v,w=map(str,input().split( ))
    w=int(w)
    d[u].append((v,w))
    d[v].append((u,w))
def bfs(x,y):
    pre={i:None for i in list_1}
    dis={i:float('inf') for i in list_1}
    h=[(0,x)]
    dis[x]=0

    while h:
        cur_dis,cur_x=heapq.heappop(h)
        if cur_x==y:
            break
        for v,w in d[cur_x]:
            if dis[cur_x]+w<dis[v]:
                dis[v]=dis[cur_x]+w
                heapq.heappush(h,(dis[v],v))
                pre[v]=cur_x

    path=[]
    current=y

    while current:
        path.insert(0,current)
        current=pre[current]
    return path
def jieguo(path):
    if len(path)==1:
```

```
            return
        else:
            n=len(path)
            for i in range (n-1):
                x=path[i]
                y=path[i+1]
                for u,v in d[x]:
                    if u==y:
                        dis=v
                        break
                print(f'{x}->({dis})->',end='')

R=int(input())for i in range (R):
    x,y=map(str,input().split( ))
    path=bfs(x,y)
    jieguo(path)
    print(path[-1])
```

10、社交网络 minus

描述

随着社交平台的兴起，人们之间的沟通变得越来越密切。通过 Facebook 的分享功能，只要你是对方的好友，你就可以转发对方的状态，并且你的名字将出现在“转发链”上。经过若干次转发以后，很可能 A 分享了一条好友 C 的状态，而 C 的这条状态实际上是分享 B 的，但 A 与 B 可能并不是好友，即 A 通过 C 间接分享了 B 的状态。

给定你 N 个人之间的好友关系，好友关系一定是双向的。只要两个人是好友，他们就可以互相转发对方的状态，无论这条状态是他自己的，还是他转发了其他人的。现在请你统计，对于每两个人，他们是否有可能间接转发对方的状态。

输入

第一行 1 个整数 N（1<=N<=20）。

接下来 N 行每行 N 个整数，表示一个 N*N 的 01 矩阵，若矩阵的第 i 行第 j 列是 1，表示这两个人是好友，0 则表示不是好友。

保证矩阵的主对角线上都是 1，并且矩阵关于主对角线对称。

输出

一个 N*N 的 01 矩阵，若矩阵的第 i 行第 j 列是 1，表示这两个人可能间接转发对方的状态，0 则表示不可能。

```
def getroot(a):
    if parent[a]!=a:
        parent[a]=getroot(parent[a])
    return parent[a]def merge(a,b):
    p1=getroot(a)
    p2=getroot(b)
    if p1==p2:
```

```
        return
    total[p1]=total[p1]+total[p2]
    parent[p2]=p1def query(a,b):
    return getroot(a)==getroot(b)

N=int(input())
L=[]
parent=list(range(N))
total=[1]*Nfor i in range (N):
    S=list(input())
    L.append(S)for i in range (N):
    for j in range (N):
        if L[i][j]=='1':
            merge(i,j)
K=[['0' for i in range (N)]for j in range (N)]for i in range (N):
    for j in range (N):
        if query(i,j):
            K[i][j]='1'for i in K:
    print(''.join(i))
```

11、棋盘问题

描述

在一个给定形状的棋盘（形状可能是不规则的）上面摆放棋子，棋子没有区别。要求摆放时任意的两个棋子不能放在棋盘中的同一行或者同一列，请编程求解对于给定形状和大小的棋盘，摆放 k 个棋子的所有可行的摆放方案 C。

输入

输入含有多组测试数据。

每组数据的第一行是两个正整数，n k，用一个空格隔开，表示了将在一个 n*n 的矩阵内描述棋盘，以及摆放棋子的数目。 n <= 8 , k <= n

当为-1 -1 时表示输入结束。

随后的 n 行描述了棋盘的形状：每行有 n 个字符，其中 # 表示棋盘区域， . 表示空白区域（数据保证不出现多余的空白行或者空白列）。

输出

对于每一组数据，给出一行输出，输出摆放的方案数目 C （数据保证 C<2^31）。

```
while True:
    n,k=map(int,input().split( ))
    if n==-1 and k==-1:
        break
    L=[]
    for i in range (n):
        S=list(input())
        L.append(S)
```

```
    used=[False]*n
    def dfs(col,count):  #第 k 行，一共放了 count 个棋子
        global result
        if count+(n-col)<k:
            return
        if count==k:
            result=result+1
            return
        if col>=n:
            return

        for i in range(n):  #该行放棋子
            if L[col][i]=='#' and used[i]==False:
                used[i]=True
                dfs(col+1,count+1)
                used[i]=False
        dfs(col+1,count)  #该行不放棋子
    result=0
    dfs(0,0)
    print(result)
```

12、斗地主大师

描述

斗地主大师今天有 P 个欢乐豆，他夜观天象，算出了一个幸运数字 Q，如果他能有恰好 Q 个欢乐豆，就可以轻松完成程设大作业了。

斗地主大师显然是斗地主大师，可以在斗地主的时候轻松操控游戏的输赢。

1.他可以轻松赢一把，让自己的欢乐豆变成原来的 Y 倍

2.他也可以故意输一把，损失 X 个欢乐豆(注意欢乐豆显然不能变成负数，所以如果手里没有 X 个豆就不能用这个能力)

而斗地主大师还有一种怪癖，扑克除去大小王只有 52 张，所以他一天之内最多只会打 52 把斗地主。

斗地主大师希望你能告诉他，为了把 P 个欢乐豆变成 Q 个，他至少要打多少把斗地主？

输入

第一行 4 个正整数 P,Q,X,Y

0< P,X,Q <= 2^31, 1< Y <= 225

输出

输出一个数表示斗地主大师至少要用多少次能力

如果打了 52 次斗地主也不能把 P 个欢乐豆变成 Q 个，请输出一行 "Failed"

```
from collections import deque

P,Q,X,Y=map(int,input().split( ))
```

```
def bfs(P,Q,X,Y):
    if P==Q:
        return 0

    q=deque()
    visited=set()
    q.append((0,P))
    visited.add(P)

    while q:
        step,current_value=q.popleft()
        if step>=52:
            return 'Failed'
            break

        Q_1=Y*current_value
        if Q_1==Q:
            return step+1
            break
        if Q_1 not in visited and Q_1 <= 2**31:
            q.append((step+1,Q_1))
            visited.add(Q_1)

        Q_2=current_value-X
        if Q_2>=0:
            if Q_2==Q:
                return step+1
                break
            if Q_2 not in visited:
                q.append((step+1,Q_2))
                visited.add(Q_2)

a=bfs(P,Q,X,Y)print(a)
```

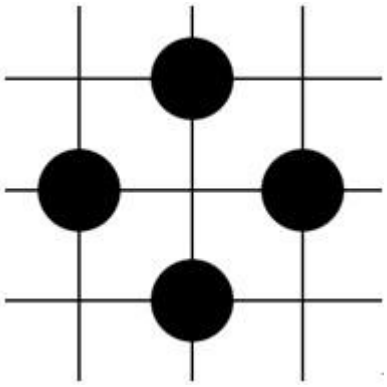
13、围棋

描述

围棋的棋盘上有 19*19 条线交织成的 361 个交点，黑棋和白棋可以下在交点上。我们称这些交点为“目”。

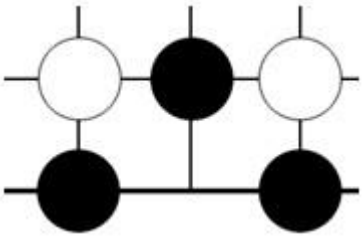
一个目的上下左右四个方向，称之为“气”，如果一个目的四个方向都被某一种颜色的棋子占据，那么即使这个目上并没有棋子，仍然认为这个目被该颜色棋子占据。

如下图中，四个黑棋中心的交点，由于被黑棋包围，因此我们认为这个目属于黑棋，



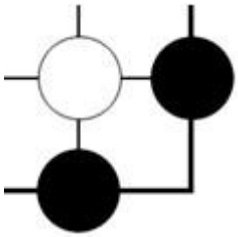
黑棋拥有 4+1=5 目

在棋盘的边框地区，只要占据目的三个方向，就可以拥有这个目。



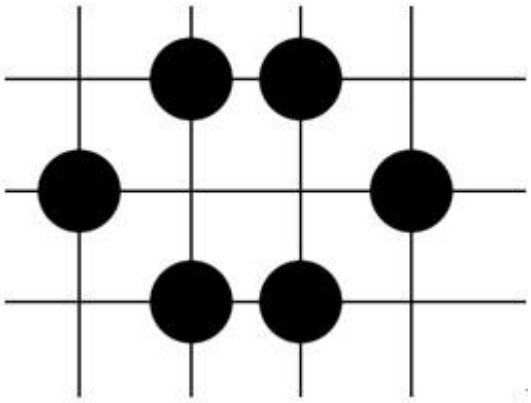
黑棋拥有 3+1=4 目

同理在棋盘的四个角上，只要占据目的两个气即可。



黑棋拥有 2+1=3 目

推而广之，当有多个目互相连通的时候，如果能用一种颜色把所有交点的气都包裹住，那么就拥有所有目。



黑棋拥有 6+2 = 8 目

请编写一个程序，计算棋盘上黑棋和白棋的目数。
输入数据中保证所有的目，不是被黑棋包裹，就是被白棋包裹。不用考虑某些棋子按照围棋规则实际上是死的，以及互相吃（打劫），双活等情况。

输入

第一行，只有一个整数 N(1<=N<=100)，代表棋盘的尺寸是 N * N
第 2~n+1 行，每行 n 个字符，代表棋盘上的棋子颜色。

“.”代表一个没有棋子的目
“B”代表黑棋
“W”代表白棋

输出

只有一行，包含用空格分隔的两个数字，第一个数是黑棋的目数，第二个数是白棋的目数。

```
from collections import deque
N=int(input())
L=[]for i in range (N):
    S=list(input())
    L.append(S)
black=0
white=0
visited=[[False]*N for i in range (N)]
def bfs(i,j):
    q=deque()
    q.append((i,j))
    visited[i][j]=True
    area=[(i,j)] #记录区域
    nbr=set()

    while q:
        x,y=q.popleft()
```

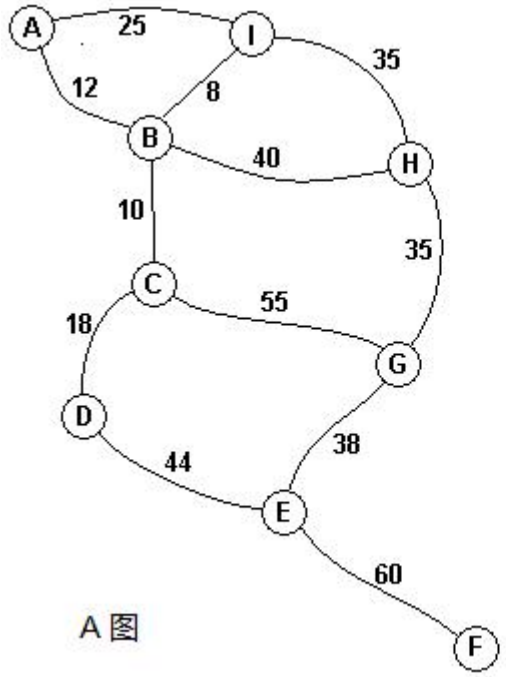
```
            if 0<=x-1<N and 0<=y<N:
                if not visited[x-1][y]:
                    if L[x-1][y]=='.':
                        visited[x-1][y]=True
                        q.append((x-1,y))
                        area.append((x-1,y))
                    else:
                        nbr.add(L[x-1][y])
            if 0<=x+1<N and 0<=y<N:
                if not visited[x+1][y]:
                    if L[x+1][y]=='.':
                        visited[x+1][y]=True
                        q.append((x+1,y))
                        area.append((x+1,y))
                    else:
                        nbr.add(L[x+1][y])
            if 0<=x<N and 0<=y-1<N:
                if not visited[x][y-1]:
                    if L[x][y-1]=='.':
                        visited[x][y-1]=True
                        q.append((x,y-1))
                        area.append((x,y-1))
                    else:
                        nbr.add(L[x][y-1])
            if 0<=x<N and 0<=y+1<N:
                if not visited[x][y+1]:
                    if L[x][y+1]=='.':
                        visited[x][y+1]=True
                        q.append((x,y+1))
                        area.append((x,y+1))
                    else:
                        nbr.add(L[x][y+1])
            else:
                continue
        if nbr=={'B'}:
            return len(area),0
        else:
            return 0,len(area)

for i in range (N):
    for j in range (N):
        if L[i][j]=='B':
            black=black+1
        elif L[i][j]=='W':
```

```
                white=white+1
            else:
                if visited[i][j]==False:
                    b,w=bfs(i,j)
                    black=black+b
                    white=white+w
print(f'{black} {white}')
```

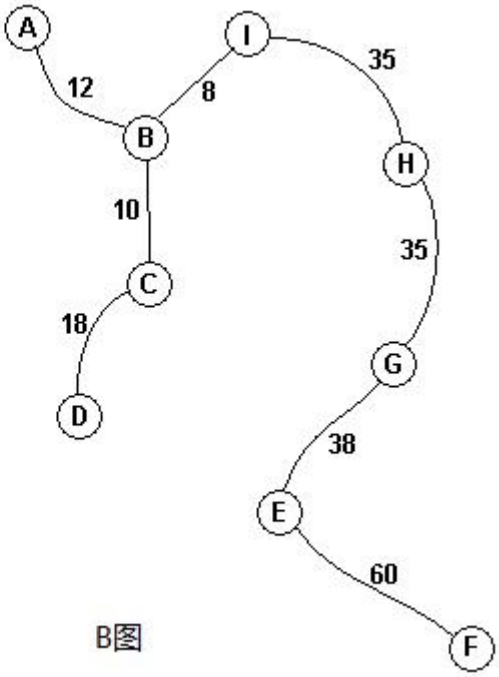
14、兔子与星空

描述

很久很久以前，森林里住着一群兔子。兔子们无聊的时候就喜欢研究星座。如图所示，天空中已经有了 n 颗星星，其中有些星星有边相连。兔子们希望删除掉一些边，然后使得保留下的边仍能是 n 颗星星连通。他们希望计算，保留的边的权值之和最小是多少？



A 图



B图

输入

第一行只包含一个表示星星个数的数 n，n 不大于 26，并且这 n 个星星是由大写字母表里的前 n 个字母表示。接下来的 n-1 行是由字母表的前 n-1 个字母开头。最后一个星星表示的字母不用输入。对于每一行，以每个星星表示的字母开头，然后后面跟着一个数字，表示有多少条边可以从这个星星到后面字母表中的星星。如果 k 是大于 0，表示该行后面会表示 k 条边的 k 个数据。每条边的数据是由表示连接到另一端星星的字母和该边的权值组成。权值是正整数的并且小于 100。该行的所有数据字段分隔单一空白。该星星网络将始终连接所有的星星。该星星网络将永远不会超过 75 条边。没有任何一个星星会有超过 15 条的边连接到其他星星（之前或之后的字母）。在下面的示例输入，数据是与上面的图相一致的。

输出

输出是一个整数，表示最小的权值和

```
def getroot(a):
    if parent[a]==a:
        return a
    parent[a]=getroot(parent[a])
    return parent[a]def merge(a,b):
    p1=getroot(a)
    p2=getroot(b)
    if p1==p2:
        return
    parent[p2]=p1def union(a,b):
    return getroot(a)==getroot(b)

n=int(input())
edges=[]
for i in range(n-1):
    L=list(input().split( ))
    k=int(L[1])
    for i in range(k):
        a,b=L[2+2*i],int(L[3+2*i])
        edges.append((b,ord(L[0])-ord('A'),ord(a)-ord('A')))

edges.sort()
parent=[i for i in range(n)]
weight=0
count=0
for w,u,v in edges:
    if not union(u,v):
        merge(u,v)
        weight=weight+w
        count=count+1
        if count==n-1:
            breakprint(weight)
```

15、拓扑排序

描述

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。

输入

若干行整数，第一行有 2 个数，分别为顶点数 v 和弧数 a，接下来有 a 行，每一行有 2 个数，分别是该条弧所关联的两个顶点编号。

v<=100, a<=500

输出

若干个空格隔开的顶点构成的序列(用小写字母)。

```
from collections import defaultdict,deque import heapq

v,a=map(int,input().split( ))
d=defaultdict(list)
degree=[0]*(v+1)
for i in range (a):
    x,y=map(int,input().split( ))
    d[x].append(y)
    degree[y]=degree[y]+1

heap=[]
for i in range (1,v+1):
    if degree[i]==0:
        heapq.heappush(heap,i)

result=[]
while heap:
    u=heapq.heappop(heap)
    result.append(f'v{u}')
    for nbr in d[u]:
        degree[nbr]=degree[nbr]-1
        if degree[nbr]==0:
            heapq.heappush(heap,nbr)
print(' '.join(result))
```

16、踩方格问题

有一个方格矩阵，矩阵边界在无穷远处。我们做如下假设：

a.每走一步时，只能从当前方格移动一格，走到某个相邻的方格上；

b.走过的格子立即塌陷无法再走第二次；

c.只能向北、东、西三个方向走；

请问：如果允许在方格矩阵上走 n 步（n<=20），共有多少种不同的方案。2 种走法只要有一步不一样，即被认为是不同的方案。

```
visited=[[0 for i in range (50)] for i in range(30)]
def ways(i,j,n):
    if n==0:
        return 1
    visited[i][j]=1
    nums=0
    if not visited[i][j-1]:
        nums=nums+ways(i,j-1,n-1)
    if not visited[i][j+1]:
        nums=nums+ways(i,j+1,n-1)
    if not visited[i+1][j]:
```

```
            nums=nums+ways(i+1,j,n-1)
        visited[i][j]=0
        return nums
```

17、Roads

N 个城市，编号 1 到 N。城市间有 R 条单向道路。每条道路连接两个城市，有长度和过路费两个属性。Bob 只有 K 块钱，他想从城市 1 走到城市 N。问最短共需要走多长的路。如果到不了 N，输出-1。

2<=N<=100

0<=K<=10000

1<=R<=10000

每条路的长度 L, 1 <= L <= 100

每条路的过路费 T , 0 <= T <= 100

动态规划+最短路径算法

```
import heapq

N, R, K = map(int, input().split())

# 建图：邻接表，d[u] = [(v, len, toll), ...]
graph = [[] for _ in range(N + 1)]
for _ in range(R):
    u, v, length, toll = map(int, input().split())
    graph[u].append((v, length, toll))

# dist[u][k]: 到达城市 u，使用 k 元以内过路费的最短路径长度
INF = float('inf')
dist = [[INF] * (K + 1) for _ in range(N + 1)]
dist[1][0] = 0

# 优先队列：(总距离, 当前城市, 当前总花费)
heap = [(0, 1, 0)]

while heap:
    cur_dist, u, cost = heapq.heappop(heap)

    if dist[u][cost] < cur_dist:
        continue  # 已被更优路径更新过了

    for v, l, t in graph[u]:
        if cost + t <= K and dist[v][cost + t] > cur_dist + l:
            dist[v][cost + t] = cur_dist + l
            heapq.heappush(heap, (dist[v][cost + t], v, cost + t))
```

```
# 答案是 dist[N][0~K]中最小的值
ans = min(dist[N])
print(ans if ans != INF else -1)
```

18、蛋糕问题

要制作一个体积为 Nπ的 M 层生日蛋糕，每层都是一个圆柱体。

设从下往上数第 i(1 <= i <= M)层蛋糕是半径为 Ri, 高度为 Hi 的圆柱。当 i<M 时，要求 Ri>Ri+1 且 Hi>Hi+1。

由于要在蛋糕上抹奶油，为尽可能节约经费，我们希望蛋糕外表面（最下一层的下底面除外）的面积 Q 最小。

令 Q = Sπ

请编程对给出的 N 和 M，找出蛋糕的制作方案（适当的 Ri 和 Hi 的值），使 S 最小。

（除 Q 外，以上所有数据皆为正整数）

多约束 DFS+剪枝

```
import math

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

INF=float('inf')
min_surf=INF

# 剪枝预处理：最少体积和面积（用于提前剪掉不可能解）
minV = [0] * (M + 1)  # 最少体积
minS = [0] * (M + 1)  # 最少面积（忽略底面）
for i in range(1, M + 1):
    minV[i] = minV[i - 1] + i * i * i
    minS[i] = minS[i - 1] + 2 * i * i

def dfs(layer, vol, surf, max_r, max_h):
    global min_surf

    # 剪枝 1：体积超了
    if vol > N:
        return

    # 剪枝 2：当前体积+最小剩余体积 > N，不可能正好填满
    if vol + minV[layer] > N:
        return

    # 剪枝 3：当前面积+最小剩余面积 ≥ 已知最优，不值得继续
    if surf + minS[layer] >= min_surf:
        return
```

```
    # 递归边界
    if layer == 0:
        if vol == N:
            min_surf = min(min_surf, surf)
        return

    # 枚举本层半径和高度
    for r in range(min(max_r - 1, int(math.sqrt(N - vol))) , layer - 1, -1):
        for h in range(min(max_h - 1, N - vol // (r * r)), layer - 1, -1):
            v = r * r * h
            s = 2 * r * h
            top = 0
            if layer == M:
                top = r * r   # 最顶层加上表面
            dfs(layer - 1, vol + v, surf + s + top, r, h)

# 从 M 层开始
dfs(M, 0, 0, int(math.sqrt(N)) + 1, N + 1)

print(min_surf if min_surf != INF else -1)
```

19、迷宫问题

定义一个矩阵：
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
它表示一个迷宫，其中的 1 表示墙壁，0 表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的最短路线。

```
from collections import deque

# 定义迷宫
maze = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0],
]

n, m = len(maze), len(maze[0])
```

```
visited = [[False]*m for _ in range(n)]
pre = [[None]*m for _ in range(n)]   # 记录前驱

# 四个方向：下、上、右、左
dirs = [(1,0), (-1,0), (0,1), (0,-1)]

def bfs():
    queue = deque()
    queue.append((0, 0))
    visited[0][0] = True

    while queue:
        x, y = queue.popleft()
        if (x, y) == (n-1, m-1):
            break   # 到终点了

        for dx, dy in dirs:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < m and not visited[nx][ny] and maze[nx][ny] == 0:
                visited[nx][ny] = True
                pre[nx][ny] = (x, y)
                queue.append((nx, ny))

# 回溯路径
def print_path():
    path = []
    x, y = n - 1, m - 1
    while (x, y) != (0, 0):
        path.append((x, y))
        x, y = pre[x][y]
    path.append((0, 0))
    path.reverse()
    for p in path:
        print(p)

# 主函数
bfs()
if visited[n-1][m-1]:
    print("最短路径如下：")
    print_path()
else:
    print("无路可走")
```

20、鸣人和佐助

已知一张地图（以二维矩阵的形式表示）以及佐助和鸣人的位置。地图上的每个位置都可以走到，只不过有些位置上有大蛇丸的手下(#)，需要先打败大蛇丸的手下才能到这些位置。

鸣人有一定数量的查克拉，每一个单位的查克拉可以打败一个大蛇丸的手下。假设鸣人可以往上下左右四个方向移动，每移动一个距离需要花费 1 个单位时间，打败大蛇丸的手下不需要时间。如果鸣人查克拉消耗完了，则只可以走到没有大蛇丸手下的位置，不可以再移动到有大蛇丸手下的位置。

佐助在此期间不移动，大蛇丸的手下也不移动。请问，鸣人要追上佐助最少需要花费多少时间？

```
from collections import deque

# 示例地图输入（你也可以改为读输入）
maze = [
    ['N', '.', '#', '.', '.'],
    ['#', '#', '.', '#', '.'],
    ['.', '.', '.', '.', '.'],
    ['.', '#', '#', '#', '.'],
    ['.', '.', '.', '#', 'S']
]
chakra = 3   # 查克拉总量

n, m = len(maze), len(maze[0])

# 找鸣人和佐助的位置
for i in range(n):
    for j in range(m):
        if maze[i][j] == 'N':
            start = (i, j)
        if maze[i][j] == 'S':
            end = (i, j)

# visited[x][y][c] 表示到 (x,y) 且剩余查克拉为 c 是否访问过
visited = [[[False] * (chakra + 1) for _ in range(m)] for _ in range(n)]

# 四个方向
dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# BFS
def bfs():
    q = deque()
    sx, sy = start
    q.append((sx, sy, chakra, 0))   # (x, y, 剩余查克拉, 当前时间)
    visited[sx][sy][chakra] = True
```

```
    while q:
        x, y, c, t = q.popleft()
        if (x, y) == end:
            return t

        for dx, dy in dirs:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < m:
                if maze[nx][ny] == '#':
                    if c > 0 and not visited[nx][ny][c - 1]:
                        visited[nx][ny][c - 1] = True
                        q.append((nx, ny, c - 1, t + 1))
                else:
                    if not visited[nx][ny][c]:
                        visited[nx][ny][c] = True
                        q.append((nx, ny, c, t + 1))
    return -1   # 到不了佐助

print("鸣人最少需要时间:", bfs())
```

问题变形：
鸣人要从迷宫中的起点 r 走到终点 a, 去营救困在那里的佐助。迷宫中各个字符代表道路（@）、墙壁（#）、和守卫（x）。
能向上下左右四个方向走。不能走到墙壁。每走一步需要花费 1 分钟
行走过程中一旦遇到守卫，必须杀死守卫才能继续前进。杀死一个守卫需要花费额外的 1 分钟，求到达目的地最少用时

```
from heapq import heappush, heappop

maze = [
    ['@', '@', '#', 'x', '@'],
    ['#', '@', '#', '@', '@'],
    ['r', '@', 'x', '#', 'a'],
    ['#', '#', '@', '@', '@'],
]

n, m = len(maze), len(maze[0])

# 找起点和终点
for i in range(n):
    for j in range(m):
        if maze[i][j] == 'r':
            start = (i, j)
```

```
            if maze[i][j] == 'a':
                end = (i, j)

# 四方向
dirs = [(-1,0), (1,0), (0,-1), (0,1)]

# Dijkstra 的基本框架（BFS+优先队列）
def bfs():
    dist = [[float('inf')]*m for _ in range(n)]
    sx, sy = start
    dist[sx][sy] = 0
    heap = [(0, sx, sy)]   # (时间花费, x, y)

    while heap:
        time, x, y = heappop(heap)
        if (x, y) == end:
            return time
        for dx, dy in dirs:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < m and maze[nx][ny] != '#':
                extra = 2 if maze[nx][ny] == 'x' else 1
                if time + extra < dist[nx][ny]:
                    dist[nx][ny] = time + extra
                    heappush(heap, (time + extra, nx, ny))
    return -1   # 到不了终点

print("鸣人最少用时：", bfs(), "分钟")
```

问题变形：
要从迷宫中的起点 r 走到终点 a, 迷宫中各个字符代表道路 （@）、墙壁（#）、和守卫（x），放有钥匙的道路 (1-9，表示有 9 种钥匙）
行走过程中一旦遇到守卫，必须杀死守卫才能继续前进。杀死一个守卫需要花费额外 1 分钟。最多 5 个守卫。
走到终点时，必须要每种钥匙至少有一把才算完成任务。 钥匙不全，也可以经过终点。
想拿第 k 种钥匙，必须手里已经有第 k-1 种钥匙。拿不了 钥匙，也可以经过放钥匙的地方。
求完成任务最少用时

```
from collections import deque

maze = [
    ['r', '@', '1', '@', '2'],
    ['#', 'x', '#', '@', '3'],
    ['@', '@', '@', 'x', '4'],
    ['5', '#', '6', '@', '7'],
```

```
        ['x', 'x', '@', '8', 'a']
]

n, m = len(maze), len(maze[0])

# 找起点
for i in range(n):
        for j in range(m):
                if maze[i][j] == 'r':
                        start = (i, j)

# 方向
dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# visited[x][y][key_level][guard_count]
visited = [[[[False] * 6 for _ in range(10)] for _ in range(m)] for _ in range(n)]

def bfs():
        q = deque()
        sx, sy = start
        q.append((sx, sy, 0, 0, 0))    # (x, y, key_level, guard_count, time)
        visited[sx][sy][0][0] = True

        while q:
                x, y, key, guard, time = q.popleft()

                # 成功条件：到达终点并拥有 1~9 所有钥匙
                if maze[x][y] == 'a' and key == 9:
                        return time

                for dx, dy in dirs:
                        nx, ny = x + dx, y + dy
                        if not (0 <= nx < n and 0 <= ny < m):
                                continue
                        cell = maze[nx][ny]
                        nkey, nguard = key, guard

                        if cell == '#':
                                continue
                        if cell == 'x':
                                if nguard >= 5:
                                        continue
                                nguard += 1
                        elif cell.isdigit():
```

```
                    k = int(cell)
                    if k == key + 1:  # 合法拿到下一个钥匙
                        nkey = k
                    elif k > key + 1:
                        # 不能提前拿更高级钥匙，但可以经过
                        pass

                if not visited[nx][ny][nkey][nguard]:
                    visited[nx][ny][nkey][nguard] = True
                    hext = time + 1 + (1 if cell == 'x' else 0)
                    q.append((nx, ny, nkey, nguard, hext))

    return -1

result = bfs()
if result == -1:
    print("无法完成任务")
else:
    print(f"鸣人完成任务的最短时间是：{result} 分钟")
```

21、Network

某地区共有 n 座村庄，每座村庄的坐标用一对整数(x, y)表示，现在要在村庄之间建立通讯网络。

通讯工具有两种，分别是需要铺设的普通线路和无线通讯的卫星设备。

只能给 k 个村庄配备卫星设备，拥有卫星设备的村庄互相间直接通讯。

铺设了线路的村庄之间也可以通讯。但是由于技术原因，两个村庄之间线路

长度最多不能超过 d, 否则就会由于信号衰减导致通讯不可靠。要想增大 d 值，则会导致要投入更多的设备（成本）

已知所有村庄的坐标(x,y)，卫星设备的数量 k。

问：如何分配卫星设备，才能使各个村庄之间能直接或间接的通讯，并且 d 的值最小？求出 d 的最小值。

数据规模：0 <= k <= n<= 500

```
import math

class UnionFind:
    def __init__(self, n):
        self.fa = list(range(n))

    def find(self, x):
        if self.fa[x] != x:
            self.fa[x] = self.find(self.fa[x])
        return self.fa[x]
```

```
    def union(self, x, y):
        fx, fy = self.find(x), self.find(y)
        if fx == fy:
            return False
        self.fa[fx] = fy
        return True

def min_d_with_satellites(points, k):
    n = len(points)
    edges = []

    # 所有边和权重（距离平方避免浮点误差）
    for i in range(n):
        for j in range(i+1, n):
            x1, y1 = points[i]
            x2, y2 = points[j]
            dist = math.hypot(x1 - x2, y1 - y2)
            edges.append((dist, i, j))

    edges.sort()
    uf = UnionFind(n)
    mst_edges = []

    for dist, u, v in edges:
        if uf.union(u, v):
            mst_edges.append(dist)

    # 删除最大的 k-1 条边（分成 k 个块）
    mst_edges.sort(reverse=True)
    if len(mst_edges) < k - 1:
        return 0.0  # 至多能分成 len(mst_edges)+1 块
    if k == 0:
        return max(mst_edges)
    return mst_edges[k - 1]

# 示例调用
n, k = 6, 2
villages = [(0, 0), (0, 1), (1, 0), (5, 5), (5, 6), (6, 5)]
print(f"最小可行的 d 值为：{min_d_with_satellites(villages, k):.2f}")
```

22、Fence Repair

一块长木板，要切割成长度为 L1,L2...Ln 的 n 块板子。每切一刀的费用，等于被切的那块板子的长度。求最少费用。

思路：

考虑等价的切割的逆过程，即用n块板子去粘接成最终的长板子。每粘接一次的费用等于粘成的木板长度。

将粘接过程中产生的每个木板，包括最终长木板，都看作一个结点。则粘接的过程可以描述成一棵树的建立过程。将n1,n2粘接成R，就相当于建一棵以R为根，n1,n2为子结点的二叉树。最终长板就是最终二叉树的树根。

建树完成后，设n_i到根的路径长度为L_i，则其参加了次L_i粘接,贡献了费用$L_i \times W_i$。要使总费用最小就是 WPL = $\sum_{i=1}^{n} W_i \times Li$ 最小，即最优二叉树问题。

```
import heapq

def min_cut_cost(lengths):
    if not lengths:
        return 0
    heapq.heapify(lengths)
    total_cost = 0
    while len(lengths) > 1:
        a = heapq.heappop(lengths)
        b = heapq.heappop(lengths)
        cost = a + b
        total_cost += cost
        heapq.heappush(lengths, cost)
    return total_cost

# 示例：
lengths = [4, 3, 2, 6]
print(f"最小切割总费用为: {min_cut_cost(lengths)}")
```

23、动态中位数

描述

工厂中的流水线上依次运送来一系列零件，每个零件具有一个重量 w_i 。

你希望知道，在每个时刻，当前已经运送来的零件的重量的中位数是多少。

输入

第一行包含一个正整数 n，表示测试数据的组数，1 <= n <= 10

下面包含 n 行，每行代表一个测试用例。

一行中包含 2k+1 个正整数（0 <= k <= 10000），依次表示运送来的零件的重量，重量不超过 1e10。

输出

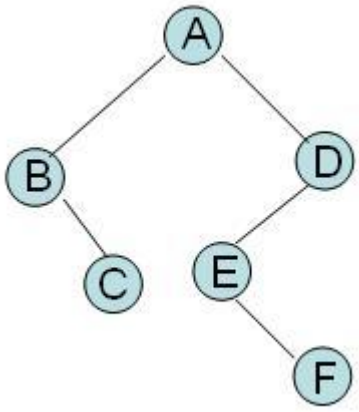
对于每行输入，依次输出前 2i+1 个零件的重量的中位数。（0 <= i <= k）

考虑用两个堆来维护：

```
import heapq

def median_stream(weights):
    low = []   # 最大堆（存较小一半，使用负数）
    high = []   # 最小堆（存较大一半）
    result = []

    for i, w in enumerate(weights):
        if not low or w <= -low[0]:
            heapq.heappush(low, -w)
        else:
            heapq.heappush(high, w)

        # 保证大小平衡：low 可以比 high 多一个
        if len(low) > len(high) + 1:
            heapq.heappush(high, -heapq.heappop(low))
        elif len(high) > len(low):
            heapq.heappush(low, -heapq.heappop(high))

        # 只在奇数个数时记录中位数
        if (i + 1) % 2 == 1:
            result.append(-low[0])

    return result
```

24、文本二叉树

描述



如上图，一棵每个节点都是一个字母，且字母互不相同的二叉树，可以用以下若干行文本表示:

```
A
-B
--*
--C
-D
--E
---*
---F
```

在这若干行文本中：
1) 每个字母代表一个节点。该字母在文本中是第几行，就称该节点的行号是几。根在第 1 行
2) 每个字母左边的'-'字符的个数代表该结点在树中的层次（树根位于第 0 层）
3) 若某第 i 层的非根节点在文本中位于第 n 行，则其父节点必然是第 i-1 层的节点中，行号小于 n,且行号与 n 的差最小的那个
4) 若某文本中位于第 n 行的节点(层次是 i) 有两个子节点，则第 n+1 行就是其左子节点，右子节点是 n+1 行以下第一个层次为 i+1 的节点
5) 若某第 i 层的节点在文本中位于第 n 行，且其没有左子节点而有右子节点，那么它的下一行就是 i+1 个'-' 字符再加上一个 '*'

给出一棵树的文本表示法，要求输出该数的前序、后序、中序遍历结果
输入
第一行是树的数目 n

接下来是 n 棵树，每棵树以'0'结尾。'0'不是树的一部分
每棵树不超过 100 个节点
输出
对每棵树，分三行先后输出其前序、后序、中序遍历结果
两棵树之间以空行分隔

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def parse_tree(lines):
    stack = {}    # depth -> latest node at this depth
    nodes = []

    i = 0
    while i < len(lines):
```

```
            line = lines[i]
            depth = line.count('-')
            value = line[depth:]

            if value == '*':
                # '*' 表示空左子节点，占位，无需处理，只用于跳过左子树
                i += 1
                continue

            node = TreeNode(value)
            nodes.append((i, depth, node))

            if depth > 0:
                # 找父节点：向上找层级为 depth-1 的节点中，行号最小的那个
                for j in range(i-1, -1, -1):
                    j_depth = lines[j].count('-')
                    j_value = lines[j][j_depth:]
                    if j_value != '*' and j_depth == depth - 1:
                        parent = stack[j_depth]
                        # 左子树优先填充
                        if not parent.left:
                            parent.left = node
                        else:
                            parent.right = node
                        break

            stack[depth] = node
            i += 1

        # 树的根是 depth==0 的那个
        for _, d, node in nodes:
            if d == 0:
                return node

def preorder(root):
    return '' if not root else root.val + preorder(root.left) + preorder(root.right)

def inorder(root):
    return '' if not root else inorder(root.left) + root.val + inorder(root.right)

def postorder(root):
    return '' if not root else postorder(root.left) + postorder(root.right) + root.val

def main():
```

```
n = int(input())
results = []

lines = []
tree_count = 0
while tree_count < n:
    line = input().strip()
    if line == '0':
        root = parse_tree(lines)
        results.append((preorder(root), postorder(root), inorder(root)))
        lines = []
        tree_count += 1
    else:
        lines.append(line)

for i, (pre, post, ino) in enumerate(results):
    print(pre)
    print(post)
    print(ino)
    if i < len(results) - 1:
        print()
```