Course 6: 二叉树基础（3）

五、二叉搜索树

（一）、定义

二叉搜索树（BST,Binary Search Tree）依赖于这样的一个性质：小于父结点的键都在左子树中，大于父结点的键都在右子树中。我们称这个性质为二叉搜索性。利用二叉树结构，二叉搜索树可以提供高效的搜索。

二叉搜索树的左子树和右子树也分别是二叉搜索树。

（二）二叉搜索树的遍历

1. 例1：基于二叉搜索树的前序遍历，求其后序遍历

输入

第一行一个正整数n（n<=2000）表示这棵二叉搜索树的结点个数；第二行n个正整数，表示这棵二叉搜索树的前序遍历；保证第二行的n个正整数中，1~n的每个值刚好出现一次

输出

一行n个正整数，表示这棵二叉搜索树的后序遍历

思路：最简单的处理方式是采取递归处理。对于任何给定树的前序遍历，基于二叉搜索树的性质，我们可以确定左、右子树的前序遍历，从而进行分割、递归的基本情况为子树为空。最后，采用左子树+右子树+结点的方式组织，即可得到后序遍历序列。

```
def post_order(pre_order):
    if not pre_order:
        return []
    root = pre_order[0]
    left_subtree = [x for x in pre_order if x < root]
    right_subtree = [x for x in pre_order if x > root]
    return post_order(left_subtree) + post_order(right_subtree) + [root]

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

2.例2：二叉搜索树的层次遍历

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。

思路：这里最重要的是构建二叉搜索树的过程。代码提供了一种解决方案：对于给定的序列，依次将数字放入二叉搜索树的相应位置。对于任一数字和任一数字所在的任一结点，若当前结点不为空，则在数字小于（大于）当前结点值的情况下，使得当前结点的左（右）结点为函数下次递归调用的返回值，对于下次递归调用，参数即为子结点位置和该数字；直到碰到空结点，为了将该数字放到这里，函数直接返回以该数字作为value的结点，完成递归。从而首次调用时，返回的即为根结点，且保证完成对二叉搜索树的构建。然后层次遍历即可。

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

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

def level_order_traversal(root):
    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

numbers = list(map(int, input().strip().split()))
numbers = list(dict.fromkeys(numbers))  # remove duplicates
root = None
for number in numbers:
    root = insert(root, number)
traversal = level_order_traversal(root)
print(' '.join(map(str, traversal)))
```

（三）二叉搜索树实现快排

回顾**快速排序**：快速排序是一种基于分治法的排序算法，它通过选择一个元素作为基准pivot，将数组分割为两个子数组，其中一个子数组的元素都小于基准，另一个子数组的元素都大于基准，然后，对两个字数组递归地应用相同的排序过程，直到排序完成。

可以使用二叉搜索树来实现快速排序的过程，具体步骤如下：

1. 选择数组中的一个元素作为基准pivot

2. 创建一个空的二叉搜索树
3. 将数组中的其他元素逐个插入二叉搜索树中
4. 按照二叉搜索树的**中序遍历**（左子树+根结点+右子树）即可得到排序结果

这种算法的时间复杂度是O（nlogn），n为数组的长度。具体而言，每次插入操作都需要O（logn）的时间复杂度，总共进行n-1次插入操作。

需要注意的是，二叉搜索树的性能取决于树的平衡性。如果二叉树变得不平衡，性能可能会下降到O（n^2）的时间复杂度。因此，在实际应用中，为了确保性能，通常会使用平衡二叉树来实现快排。

代码实现：

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

def insert(root, val):
    if root is None:
        return TreeNode(val)
    if val < root.val:
        root.left = insert(root.left, val)
    else:
        root.right = insert(root.right, val)
    return root

def inorder_traversal(root, result):
    if root:
        inorder_traversal(root.left, result)
        result.append(root.val)
        inorder_traversal(root.right, result)

def quicksort(nums):
    if not nums:
        return []
    root = TreeNode(nums[0])
    for num in nums[1:]:
        insert(root, num) #函数可以选择不进行赋值（返回结果）
    result = []
    inorder_traversal(root, result)
    return result
```

六、平衡二叉搜索树AVL

（一）平衡二叉搜索树的定义

当二叉搜索树不平衡时，get和put等操作的性能可能降到O(n)。**平衡二叉搜索树**（AVL，Adelson-Velsky and Landis）是一种特殊的二叉搜索树，它能自动维持平衡，AVL即取自这两位科学家的姓氏的首字母缩写。

AVL平衡树通过在每个结点上维护一个平衡因子（balance factor）来实现平衡。平衡因子是指结点的左子树高度与右子树高度之差。通过不断调整树的结构，AVL树能够保持树的平衡，使得最坏情况下的查找、插入和删除操作的时间复杂度保持在O（logn）

AVL的自平衡性，来源于在每次插入或删除结点时，会通过旋转操作来调整树的结构，使得平衡因子在特定的范围内，通常是-1，0，1。

平衡因子：balanceFactor = height(leftSubTree) - height(rightSubtree)

根据上述定义，如果平衡因子大于0，我们称之为左倾；如果平衡因子小于0，我们称之为右倾；如果平衡因子等于0，那么树就是完全平衡的。

平衡树：平衡因子为-1、0和1的树。一旦某个结点的平衡因子超出这个范围，我们就需要通过一个过程让树恢复平衡。

（二）AVL树的性能

1.n层AVL树至少有几个结点

AVL树的最小结点对应着AVL树最不平衡的左倾/右倾情况

设最小结点数为an，考虑根结点，最不平衡的情况在于一颗子树为n-1的最坏情况，一颗为n-2的最坏情况，故而有递推公式：

an = an-1 + an-2 + 1，其中a0 = 0，a1 = 1

代码实现：

```
def avl_min_nodes(n,memo):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    elif memo[n] != 0:
        return memo[n]
    else:
        memo[n] = avl_min_nodes(n-1,memo) + avl_min_nodes(n-2,memo)+1
        return memo[n]
n = int(input())
memo = [0]*(n+1)
min_nodes = avl_min_nodes(n,memo)
print(min_nodes)
#或可采用@lru_cache(maxsize = None)方法
```

2.n个结点的AVL树最多有多少层?

思路：依据n层AVL树至少有几个结点，该问题本质上是一个对偶问题。只需找到结点树n不小于ak的最大的k即可

```
from functools import lru_cache
@lru_cache(maxsize=None)
def min_nodes(h):
    if h == 0:return 0
    if h == 1:return 1
    return min_nodes(h-1)+min_nodes(h-2)+1
def max_height(n):
    h = 0
    while min_nodes(h)<=n:
        h += 1
    return h-1
n = int(input())
print(max_height(n))
```

（三）AVL树的实现

1.基本原理

考虑往AVL中插入一个键。所有新键都必须以叶子结点插入，插入的过程同一般的二叉搜索树没有区别。但是插入新结点后，我们必须更新父结点的平衡因子。若新的叶子结点是左子结点，父结点的平衡因子+1；如果新的叶子结点是右子结点，则父结点的平衡因子-1。插入一个键后，只有在从根结点到该插入结点的路径上的结点才可能发生平衡因子的变化，因此只需要对这条路径上的失衡结点进行调整。

可以证明，只要把最靠近插入结点的失衡结点调整到正常，路径上的所有结点都会平衡。

高效的再平衡需要再树上作一次或多次旋转。

（1）基本分类

我们首先考虑最靠近插入结点的失衡结点A可能存在的情况：

显然，A的平衡因子只可能是2或者-2。

当A的平衡因子是2时，左子树高度比右子树大2，这意味着我们向左子树插入了一个叶子结点。设A的左子结点为B，则B平衡因子不可能为0（这是因为若平衡因子为0，则B的左子树与右子树插入后高度相等，说明在插入前，至少有一个子树达到了该高度，那么此时A就不可能在插入前保持平衡）。

进一步，若B的平衡因子是1，这说明B的左子树高度比右子树高度高1，且结点插入在B的左子树位置帮助B的左子树产生了高度跃迁，我们称这种树型为**LL型**；

若B的平衡因子是-1，这说明使得高度产生跃迁的是非平衡结点的左子树的右子树，称这种树型为**LR型**

同理可得，使得高度产生跃迁的是非平衡节点的右子树的左子树，称这种树型为**RL型**；若是非平衡结点的右子树的右子树，则为**RR型**。

（2）对LL型的处理：右旋

将左子结点提升为非平衡树的根结点；

将原非平衡树的根结点下沉为新根结点（原左子结点）的右结点；

若新根结点（原左子结点）存在右子结点，将该子结点作为新右子结点（原根结点）的左子结点

（3）对RR型的处理：左旋

将右子结点提升为非平衡树的根结点

将原非平衡树的根节点下沉为新根结点（原右子结点）的左子结点

若新根结点（原右子结点）存在左子结点，将该子结点作为新左子结点（原根结点）的右子结点

（4）对LR型的处理：先左旋再右旋

对非平衡子树根结点的左子结点进行一次左旋：即以左子树为处理树，将其右子结点作为根结点，原根结点下沉至左子结点，在原右子结点（新根结点）有左子结点的情况下，将其连接到原根结点（新左子结点）的右子结点处

左旋后，该非平衡树成为**LL型**（注意此时有可能非平衡树的左子结点此时也可能是非平衡的），再进行右旋变化即可（这里右旋以最初的非平衡树作为根结点）

（5）对RL型的处理：先右旋再左旋

对非平衡子树根结点的右子结点进行一次右旋：即以右子树为处理树，将其左子结点作为根结点，原根结点下沉至右子结点，在原左子结点（新根结点）有右子结点的情况下，将其连接到原根结点（新右子结点）的左子结点处

右旋后，该平衡树成为**RR型**（注意此时有可能非平衡树的右子结点也可能是非平衡的），再进行左旋变化即可（这里左旋以最初的非平衡树作为根结点）

2.时间复杂度

通过维持树的平衡，可以保证查找的时间复杂度为O（log_2n）

插入的时间：新结点作为叶子结点插入，更新所有父结点的平衡因子最多需要log2(n)次操作（每层一次）；如果树发生失衡，恢复平衡最多需要旋转两次，每次旋转的时间复杂度为O（1）。因此，插入操作的时间复杂度为O（log_2n）

注：进行左右旋后，可以保证非平衡树的根结点高度同在插入前的高度保持一致。

3.平衡二叉树AVL的建立

将n个互不相同的正整数先后插入到一棵空的AVL树中，求最后生成的AVL树的先序序列

输入

第一行一个整数n，表示AVL树的结点个数；

第二行n个以空格分开的整数，表示插入序列。

输出

输出n个整数，表示先序遍历序列，中间用空格隔开，行末不允许有多余的空格。

案例代码：

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1

class AVL:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = Node(value)
        else:
            self.root = self._insert(value, self.root)

    def _insert(self, value, node):
        if not node:
            return Node(value)
        elif value < node.value:
            node.left = self._insert(value, node.left)
        else:
            node.right = self._insert(value, node.right)

        node.height = 1 + max(self._get_height(node.left), self._get_height(node.right))

        balance = self._get_balance(node)

        if balance > 1:
            if value < node.left.value: # 树形是 LL
                return self._rotate_right(node)
            else:       # 树形是 LR
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)

        if balance < -1:
            if value > node.right.value:        # 树形是 RR
                return self._rotate_left(node)
            else:       # 树形是 RL
                node.right = self._rotate_right(node.right)
                return self._rotate_left(node)

        return node

    def _get_height(self, node):
        if not node:
            return 0
        return node.height

    def _get_balance(self, node):
```

```
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    def _rotate_left(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        x.height = 1 + max(self._get_height(x.left), self._get_height(x.right))
        return x

    def preorder(self):
        return self._preorder(self.root)

    def _preorder(self, node):
        if not node:
            return []
        return [node.value] + self._preorder(node.left) + self._preorder(node.right)

n = int(input().strip())
sequence = list(map(int, input().strip().split()))

avl = AVL()
for value in sequence:
    avl.insert(value)

print(' '.join(map(str, avl.preorder())))
```

思路：对于结点的定义，增加高度属性，初始高度设置为1（高度的定义本质是我们原来定义的深度）

对于任何一个value，插入时调入avl.insert(value)函数：如果avl没有根结点，将value设置为根结点；否则将根结点赋值为_insert(value,self.root)函数

avl.insert(value,node)函数表示该值已经索引到node这一结点。如果这一结点为空，直接进行赋值；否则value与node.value进行比较，确定下一步索引left还是right。若是left，则该步将node.left重新赋值，调用本身_insert(value,node.left)，right部分同理。需要注意到的是，这样的调用说明我们每次插入时，是从叶结点向根结点逐层递归的，只有当底部结点完成组装、旋转，我们才会将稳定的结构调用给这一结点。

在插入好后（至少在该结点底部的结点完成了安装），考虑这一结点是否符合balance。为此，先通过_get_height函数确定左右结点的高度，进而得到该结点的高度。再用_get_balance函数确定左右子树高度之差。若balance在正负一之间，直接return node，表明完成node这一个结点的插入调用，进而返回上一层的处

理。若balance有问题，则需要对这一个balance做处理。根据balance为2或者-2，以及插入value同子结点value大小的比较，确定树型。依据树型分别进行左旋&右旋操作。

对于LL，对node进行右旋；对RR，对node进行左旋；对LR，对node.left进行左旋，再对node进行右旋；对RL，对node.right进行右旋，再对node进行左旋。

为此，考虑左旋与右旋的函数构建：

对于左旋函数，记node为z，右子结点为y，应该让z.right = y.left, y.lef t= z,然后更新z和y的高度，并返回根结点y；

对于右旋函数，记node为z，左子结点为y，应该让z.left = y.right, y.right = z,然后更新z和y的高度，并返回根结点y。

最后，解析完毕后，进行前序遍历输出。为此，先得到AVL的root，然后调用preorder函数，输出前序遍历即可。

（四）映射实现总结

用来实现映射这一抽象数据类型的多种数据结构，包括有序列表、散列表、二叉搜索树以及AVL树。表6-1总结了每个数据结构的性能。

operation	Sorted List	Hash Table	Binary Search Tree	AVL Tree
put	O(n)	O(1)	O(n)	O(log_2{n})
get	O(log_2{n})	O(1)	O(n)	O(log_2{n})
in	O(log_2{n})	O(1)	O(n)	O(log_2{n})
del	O(n)	O(1)	O(n)	O(log_2{n})

七、并查集（Disjoint Set）

（一）基本定义

在并查集中，每个元素都属于一个集合，并且这些集合之间是互不相交的。为了高效地实现并查集操作，通常会使用树形结构来表示集合之间的关系。每个集合用一个树来表示，其中树的根结点是集合的代表元素。

因此，从根本上说，并查集是存储不重叠或不相交的子集的数据结构。并查集支持以下操作：

（1）将新集合添加到并查集中

（2）使用**并操作**（union operation）将并查集合并为单个并查集

（3）使用**查找操作**（find operation）查找并查集的代表元素

（4）检查两个集合是否不相交

（二）代码实现

1.基本实现

```
# Finds the representative of the set
# that i is an element of

def find(parent,i):

        # If i is the parent of itself
        if (parent[i] == i):

                # Then i is the representative of
                # this set
                return i
        else:

                # Else if i is not the parent of
                # itself, then i is not the
                # representative of his set. So we
                # recursively call Find on its parent
                return find(parent[i])

# Unites the set that includes i
# and the set that includes j

def union(parent, i, j):
        # Find the representatives
        # (or the root nodes) for the set
        # that includes i
        irep = find(parent, i)

        # And do the same for the set
        # that includes j
        jrep = find(parent, j)

        # Make the parent of i's representative
        # be j's representative effectively
        # moving all of i's set into j's set)

        parent[irep] = jrep
```

注：时间复杂度为O（n）

（三）优化

1.路径压缩 Path Compression

通过对find函数进行优化，降低查找代表元素的时间复杂度。其具体实现方式在于，每次查找时，都通过一个简单的缓存机制使得路径上的全部元素的represent都指向根结点

```
# Finds the representative of the set that i
# is an element of.
```

```
def find(Parent,i):

        # If i is the parent of itself
        if Parent[i] == i:

                # Then i is the representative
                return i
        else:

                # Recursively find the representative.
                result = find(Parent[i])

                # We cache the result by moving i’s node
                # directly under the representative of this
                # set
                Parent[i] = result

                # And then we return the result
                return result
```

2.按秩合并 Union by Rank

按秩合并指的是在每次合并操作时，都把**秩**较小的树根结点（代表元）指向**秩**较大的树根结点

这里的秩，一般指的是树的深度，即该并查集经过合并的次数，初始化为1。

通过rank列表，我们可以记录每棵树的大小：若该元素为代表元素，则其索引的rank列表即为其秩的大小

```
class DisjSet:
        def __init__(self, n):
                # Constructor to create and initialize sets of n items
                self.rank = [1] * n
                self.parent = [i for i in range(n)]


        # Finds set of given item x
        def find(self, x):

                # Finds the representative of the set that x is an element of
                if (self.parent[x] != x):

                        # if x is not the parent of itself
                        # Then x is not the representative of its set
                        self.parent[x] = self.find(self.parent[x])

                        # so we recursively call Find on its parent
                        # and move i's node directly under the
                        # representative of this set

                return self.parent[x]
```

```
        # Do union of two sets represented by x and y.
        def Union(self, x, y):

                # Find current sets of x and y
                xset = self.find(x)
                yset = self.find(y)

                # If they are already in same set
                if xset == yset:
                        return

                # Put smaller ranked item under
                # bigger ranked item if ranks are different
                if self.rank[xset] < self.rank[yset]:
                        self.parent[xset] = yset

                elif self.rank[xset] > self.rank[yset]:
                        self.parent[yset] = xset

                # If ranks are same, then move y under x (doesn't matter
    # which one goes where) and increment rank of x's tree
                else:
                        self.parent[yset] = xset
                        self.rank[xset] = self.rank[xset] + 1

# Driver code
obj = DisjSet(5)
obj.Union(0, 2)
obj.Union(4, 2)
obj.Union(3, 1)
if obj.find(4) == obj.find(0):
        print('Yes')
else:
        print('No')
```

3.按大小合并 Union by Size

这里的大小指的是并查集（树）的结点个数。在每次合并操作时，都把集合结点个数较少的树根结点指向集合结点个数较大的树根结点

具体操作上，采用数组size记录每个根结点对应的集合结点个数

```
class UnionFind:
        def __init__(self, n):
                self.Parent = list(range(n))
                self.Size = [1] * n

        # Function to find the representative (or the root node) for the set that includes i
        def find(self, i):
                if self.Parent[i] != i:
                        # Path compression: Make the parent of i the root of the set
                        self.Parent[i] = self.find(self.Parent[i])
```

```
                return self.Parent[i]

        # Unites the set that includes i and the set that includes j by size
        def unionBySize(self, i, j):
                # Find the representatives (or the root nodes) for the set that includes i
                irep = self.find(i)

                # And do the same for the set that includes j
                jrep = self.find(j)

                # Elements are in the same set, no need to unite anything.
                if irep == jrep:
                        return

                # Get the size of i's tree
                isize = self.Size[irep]

                # Get the size of j's tree
                jsize = self.Size[jrep]

                # If i's size is less than j's size
                if isize < jsize:
                        # Then move i under j
                        self.Parent[irep] = jrep

                        # Increment j's size by i's size
                        self.Size[jrep] += self.Size[irep]
                # Else if j's size is less than i's size
                else:
                        # Then move j under i
                        self.Parent[jrep] = irep

                        # Increment i's size by j's size
                        self.Size[irep] += self.Size[jrep]

# Example usage
n = 5
unionFind = UnionFind(n)

# Perform union operations
unionFind.unionBySize(0, 1)
unionFind.unionBySize(2, 3)
unionFind.unionBySize(0, 4)
```

4.优化算法的时间复杂度问题

对于path_compression而言，平均的时间复杂度为O(logn)

对于创造n个单一的集合来说，时间复杂度无可避免为O(n)

对于单一的合并技术（without compression）而言，时间复杂度为O（logn），对于两种技术共同存在的情况下，时间复杂度可以接近于常数形态

（四）例题

1.晴问9.6.1 学校的班级个数

现有一个学校，学校中有若干个班级，每个班级中有若干个学生，每个学生只会存在于一个班级中。如果学生 A 和学生 B 处于一个班级，学生 B 和学生 C 处于一个班级，那么我们称学生 A 和学生 C 也处于一个班级。

现已知学校中共 n 个学生（编号为从 1 到 n ），并给出 m 组学生关系（指定两个学生处于一个班级），问总共有多少个班级。

输入

第一行两个整数n,m，分别表示学生个数、学生关系个数；

接下来 m 行，每行两个整数 a 和 b，表示编号为 a 的学生和编号为 b 的学生处于一个班级。

输出

输出一个整数，表示班级个数。

```
def find(x):
    if parent[x] != x: # 如果不是根结点，继续循环
        parent[x] = find(parent[x])
    return parent[x]

def union(x, y):
    parent[find(x)] = find(y)

n, m = map(int, input().split())
parent = list(range(n + 1))     # parent[i] == i，则说明元素i是该集合的根结点

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

classes = set(find(x) for x in range(1, n + 1))
print(len(classes))
```

Note: find和union都是比较标准的写法，其中union没有进行优化。最后输出组别个数时，采用了set集合表示代表元；也可以采用列表表达式，加入条件parent[x] == x

2.晴问9.6.2 学校的班级个数（2）

现有一个学校，学校中有若干个班级，每个班级中有若干个学生，每个学生只会存在于一个班级中。如果学生 A 和学生 B 处于一个班级，学生 B 和学生 C 处于一个班级，那么我们称学生 A 和学生 C 也处于一个班级。

现已知学校中共 n 个学生（编号为从 1 到 n ），并给出 m 组学生关系（指定两个学生处于一个班级），问总共有多少个班级，并按降序给出每个班级的人数。

输入

第一行两个整数n、m，分别表示学生个数、学生关系个数；

接下来 m 行，每行两个整数 a 和 b，表示编号为 a 的学生和编号为 b 的学生处于一个班级。

输出

第一行输出一个整数，表示班级个数；

第二行若干个整数，按降序给出每个班级的人数。整数之间用空格隔开，行末不允许有多余的空格。

```
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_x != root_y:
        parent[root_x] = root_y
        size[root_y] += size[root_x]

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

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

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

Note:这里的妙处在于size的应用，不仅降低了时间复杂度，还帮助统计了每个并查集的元素个数。最后采用列表表达式完成人数输出

3.晴问9.6.3 是否相同班级

现有一个学校，学校中有若干个班级，每个班级中有若干个学生，每个学生只会存在于一个班级中。如果学生 A 和学生 B 处于一个班级，学生 B 和学生 C 处于一个班级，那么我们称学生 A 和学生 C 也处于一个班级。

现已知学校中共 n 个学生（编号为从 1 到 n ），并给出 m 组学生关系（指定两个学生处于一个班级）。然后给出 k 个查询，每个查询询问两个学生是否在同一个班级。

输入

第一行两个整数n、m，分别表示学生个数、学生关系个数；

接下来 m 行，每行两个整数 a 和 b，表示编号为 a 的学生和编号为 b 的学生处于一个班级。

然后一个整数k，表示查询个数；

接下来 k 行，每行两个整数 a 和 b，表示询问编号为 a 的学生和编号为 b 的学生是否在同一个班级。

输出

每个查询输出一行，如果在同一个班级，那么输出 Yes ，否则输出 No 。

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

def union(x, y):
    parent[find(x)] = find(y)

n, m = map(int, input().split())
parent = list(range(n + 1))

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

k = int(input())
for _ in range(k):
    a, b = map(int, input().split())
    if find(a) == find(b):
        print('Yes')
    else:
        print('No')
```

Note:主程序直接调用find函数进行判断即可

4.晴问9.6.4 是否相同班级

现有一个迷宫，迷宫中有 n 个房间（编号为从 1 到 n ），房间与房间之间可能连通。如果房间 A 和房间 B 连通，房间 B 和房间 C 连通，那么我们称房间 A 和房间 C 也连通。给定 m 组连通关系（指定两个房间连通），问迷宫中的所有房间是否连通。

输入

第一行两个整数n、m，分别表示房间个数、连通关系个数；

接下来行，每行两个整数 a 和 b，表示编号为 a 的房间和编号为 b 的房间是连通的。

输出

如果所有房间连通，那么输出 Yes ，否则输出 No 。

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

def union(x, y):
    parent[find(x)] = find(y)

n, m = map(int, input().split())
parent = list(range(n + 1))

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

sets = set(find(x) for x in range(1, n + 1))
if len(sets) == 1:
    print('Yes')
else:
    print('No')
```

Note:主程序运用集合，判断是否代表元集合长度为1

5.晴问9.6.5 班级最高分

现有一个学校，学校中有若干个班级，每个班级中有若干个学生，每个学生只会存在于一个班级中。如果学生 A 和学生 B 处于一个班级，学生 B 和学生 C 处于一个班级，那么我们称学生 A 和学生 C 也处于一个班级。

现已知学校中共 n 个学生（编号为从 1 到 n ），每个学生有一个考试分数，再给出 m 组学生关系（指定两个学生处于一个班级），问总共有多少个班级，并按降序给出每个班级的最高考试分数。

输入

第一行两个整数 n、m，分别表示学生个数、学生关系个数；

第二行为用空格隔开的 n 个整数，表示个学生的考试分数；

接下来 m 行，每行两个整数 a 和 b，表示编号为 a 的学生和编号为 b 的学生处于一个班级。

输出

第一行输出一个整数，表示班级个数；

第二行若干个整数，按降序给出每个班级的最高考试分数。整数之间用空格隔开，行末不允许有多余的空格。

```
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_x != root_y:
        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 = list(map(int, input().split()))
scores.insert(0, 0)  # to make the scores 1-indexed

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

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

Note:建立score列表，在每次进行union时，将代表元的score修改为两个代表元score的最大值。主程序采用列表表达式的条件判断得到代表元分数列表，降序输出即可，而班级个数可以用列表长度得到。

6.OJ 01182 食物链

动物王国中有三类动物A,B,C，这三类动物的食物链构成了有趣的环形。A吃B， B吃C，C吃A。

现有N个动物，以1－N编号。每个动物都是A,B,C中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这N个动物所构成的食物链关系进行描述：

第一种说法是"1 X Y"，表示X和Y是同类。

第二种说法是"2 X Y"，表示X吃Y。

此人对N个动物，用上述两种说法，一句接一句地说出K句话，这K句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

1） 当前的话与前面的某些真的话冲突，就是假话；

2） 当前的话中X或Y比N大，就是假话；

3） 当前的话表示X吃X，就是假话。

你的任务是根据给定的N（1 <= N <= 50,000）和K句话（0 <= K <= 100,000），输出假话的总数。

输入

第一行是两个整数N和K，以一个空格分隔。

以下K行每行是三个正整数 D，X，Y，两数之间用一个空格隔开，其中D表示说法的种类。

若D=1，则表示X和Y是同类。

若D=2，则表示X吃Y。

输出

只有一个整数，表示假话的数目。

代码思路：

```
jointSet:
_init__(self, n):
设[1,n] 区间表示同类（第i个元素表示与i同类的代表元）
[n+1,2*n]表示x吃的动物（第i个元素表示与i吃同类的代表元）
[2*n+1,3*n]表示吃x的动物（第i个元素表示吃i的代表元）
对于真话的解读：两者同类等价于--吃两者同类，两者吃同类，两者同类
x吃y等价于--x与吃y同类，x吃与y同类，y吃与吃x同类
此刻，对于任何x与y，站在x的角度，若存在联系，则其必然有：
x与y，x与y+n，x与y+2n一者有联系（代表元相同）
对于假话的判断：对于x与y，只看x同y、y+n和y+2n的关系是否违背
x=y为错当且仅当我们已知x与y吃(y+n)或吃y(y+2n)的关系
x吃y为错当且仅当已知y与x同类或y与吃x（x+2n）同类
elf.parent = [i for i in range(3 * n + 1)] # 每个动物有三种可能的类型，用 3 * n 来表示每种类型的并查集
elf.rank = [0] * (3 * n + 1)

ind(self, u):
f self.parent[u] != u:
   self.parent[u] = self.find(self.parent[u])
eturn self.parent[u]

nion(self, u, v):
u, pv = self.find(u), self.find(v)
f pu == pv:
   return False
f self.rank[pu] > self.rank[pv]:
   self.parent[pv] = pu
lif self.rank[pu] < self.rank[pv]:
   self.parent[pu] = pv
lse:
   self.parent[pv] = pu
   self.rank[pu] += 1
eturn True


lid(n,statements):
 DisjointSet(n)

ind_disjoint_set(x):
f x > n:
   return False
eturn True
```

```
_count = 0
, x, y in statements:
f not find_disjoint_set(x) or not find_disjoint_set(y):
   false_count += 1
   continue
f d == 1:  # X and Y are of the same type
   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)
lse:  # X eats Y
   if dsu.find(x) == dsu.find(y) or dsu.find(x + 2*n) == dsu.find(y):
       false_count += 1
   else: #[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
       dsu.union(x + n, y) #x吃的动物与y同类
       dsu.union(x, y + 2 * n) #吃y的动物与x同类
       dsu.union(x + 2 * n, y + n) #吃x的动物与y吃的动物同类

n false_count


__ == "__main__":
= map(int, input().split())
ments = []
 in range(K):
, X, Y = map(int, input().split())
tatements.append((D, X, Y))
t = is_valid(N, statements)
(result)
```

八、线段树（segment Tree）

1.基本定义

线段树是一种二叉搜索树，数据结构用列表储存。使用线段树可以让我们既能快速改变元素的值，又能以较快速度求出array中一个区间的元素求和

具体而言，对于一个待处理的、长度为n的array，构建一个长度为2n的列表。array中索引数为i的元素，在列表中索引位置为i+n。而对于列表中索引位置为0-n-1的元素，其值生成方式为：

```
tree[i] = tree[2*i] + tree[2*i+1]
```

2.代码实现

```
# limit for array size
N = 100000

# Max size of tree
```

```
tree = [0] * (2 * N)

# function to build the tree
def build(arr) :

        # insert leaf nodes in tree
        for i in range(n) :
                tree[n + i] = arr[i]
        for i in range(n - 1, 0, -1):
        tree[i] = tree[i<<1] + tree[i<<1+1]


def updateTreeNode(p, value) :
        tree[p + n] = value
        p = p + n

        # move upward and update parents
        i = p

        while i > 1 :
# >>表示整除2，后面的数值表示整除几次，<<同理表示乘2
#^表示异或运算符，分别比较两个数的二进制的每一位：若该位的值不相同，该位结果为1，否则该位结果为0
#这里^用以判断与i在一棵树下的另一个节点的坐标
                tree[i >> 1] = tree[i] + tree[i ^ 1]
                i >>= 1

# function to get sum on interval [l, r) #表示索引位置
def query(l, r) :

        res = 0

        # loop to find the sum in the range
        l += n
        r += n

        while l < r :
        # 这里&表示位与运算符，当两个操作数都为1时，该位为1，否则为0
    # 用在这里，以确认l/r是否为奇数
    #这是因为只有奇数的情况下，我们才需要进行操作，这是因为偶数是左子结点，往上走一定可以被容纳到
                if l& 1 :
                        res += tree[l]
                        l += 1

                if r& 1 :
                        r -= 1
#注意操作顺序
                        res += tree[r]
                l >>= 1
                r >>= 1

        return res

if __name__ == "__main__" :

        a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

```
n = len(a)

build(a)

# print the sum in range(1,2) index-based
print(query(1, 3))

# modify element at 2nd index
updateTreeNode(2, 1)

# print the sum in range(1,2) index-based
print(query(1, 3))
```

3.时间复杂度分析

建树过程：O(n)

更新元素过程：O(logn)

得到区间和过程：O(logn)

4.例题解析：1364A-XXXXX

（1）问题描述

Ehab loves number theory, but for some reason he hates the number x. Given an array a, find the length of its longest subarray such that the sum of its elements **isn't** divisible by x, or determine that such subarray doesn't exist.

An array a is a subarray of an array b if a can be obtained from b by deletion of several (possibly, zero or all) elements from the beginning and several (possibly, zero or all) elements from the end.

Input

The first line contains an integer t (1≤t≤5) — the number of test cases you need to solve. The description of the test cases follows.

The first line of each test case contains 2 integers n and x (1≤n≤10^5^, 1≤x≤10^4^) — the number of elements in the array a and the number that Ehab hates.

The second line contains n space-separated integers $$a_1$, a_2, ……, a_n (0≤a_i≤10^4)$ — the elements of the array a.

Output

For each testcase, print the length of the longest subarray whose sum isn't divisible by x. If there's no such subarray, print –1.

（2）代码实现

```
t = int(input())
ans = []
for _ in range(t):
    n, x = map(int, input().split())
    a = [int(i) for i in input().split()]
    # Max size of tree
    tree = [0] * (2 * n)
    def build(arr) :
        # insert leaf nodes in tree
        for i in range(n) :
            tree[n + i] = arr[i]

        # build the tree by calculating parents
        for i in range(n - 1, 0, -1) :
            tree[i] = tree[i << 1] + tree[i << 1 | 1]

    # function to update a tree node
    def updateTreeNode(p, value) :

        # set value at position p
        tree[p + n] = value
        p = p + n

        # move upward and update parents
        i = p

        while i > 1 :

            tree[i >> 1] = tree[i] + tree[i ^ 1]
            i >>= 1

    # function to get sum on interval [l, r)
    def query(l, r) :

        res = 0

        # loop to find the sum in the range
        l += n
        r += n

        while l < r :

            if (l & 1) :
                res += tree[l]
                l += 1

            if (r & 1) :
                r -= 1
                res += tree[r]

            l >>= 1
            r >>= 1

        return res
```

```
build([i%x for i in a]);

left = 0
right = n - 1
if right == 0:
    if a[0] % x !=0:
        print(1)
    else:
        print(-1)
    continue

leftmax = 0
rightmax = 0
while left != right:
    total = query(left, right+1)
    if total % x != 0:
        leftmax = right - left + 1
        break
    else:
        left += 1

left = 0
right = n - 1
while left != right:
    total = query(left, right+1)
    if total % x != 0:
        rightmax = right - left + 1
        break
    else:
        right -= 1

if leftmax == 0 and rightmax == 0:
    print(-1)
else:
    print(max(leftmax, rightmax))
```

九、前缀树（Trie Tree）

1.基本定义

Trie一词源于retrieval（检索）。Trie数据结构遵循的属性是，如果两个字符串有一个共同的前缀，那么它们在Trie中将具有相同的祖先。Trie可以用于按照字母顺序对字符串集合进行排序，以及搜索Trie是否存在具有给定前缀的字符串

Trie数据结构的一些重要属性：

1. 每个Trie都有一个根节点
2. Trie的每个节点表示一个字符串，每条边表示一个字符
3. 每个节点都由散列映射或指针数组组成，每个索引表示一个字符和一个标志，用于指示是否有任何字符串在当前节点结束
4. Trie数据结构可以包含任意数量的字符，包括字母、数字、特殊字符等

5. 从根到任何节点的每条路径都表示一个单词或字符串

2.前缀树的代码实现：以26字母为例

```
class TrieNode:
        def __init__(self):
                # pointer array for child nodes of each node
                self.childNode = [None] * 26
                self.wordCount = 0

def insert_key(root, key):
        # Initialize the currentNode pointer with the root node
        currentNode = root

        # Iterate across the length of the string
        for c in key:
                # Check if the node exist for the current character in the Trie.
                if not currentNode.childNode[ord(c) - ord('a')]:
                        # If node for current character does not exist
                        # then make a new node
                        newNode = TrieNode()
                        # Keep the reference for the newly created node.
                        currentNode.childNode[ord(c) - ord('a')] = newNode
                # Now, move the current node pointer to the newly created node.
                currentNode = currentNode.childNode[ord(c) - ord('a')]
        # Increment the wordEndCount for the last currentNode
        # pointer this implies that there is a string ending at currentNode.
        currentNode.wordCount += 1

def search_key(root, key):
        # Initialize the currentNode pointer with the root node
        currentNode = root

        # Iterate across the length of the string
        for c in key:
                # Check if the node exist for the current character in the Trie.
                if not currentNode.childNode[ord(c) - ord('a')]:
                        # Given word does not exist in Trie
                        return False
                # Move the currentNode pointer to the already existing node for current charac
                currentNode = currentNode.childNode[ord(c) - ord('a')]

        return currentNode.wordCount > 0

def delete_key(root, word):
        currentNode = root
        lastBranchNode = None
        lastBrachChar = 'a'

        for c in word:
                if not currentNode.childNode[ord(c) - ord('a')]:
                        return False
                else:
                        count = 0
                        for i in range(26):
```

```
                        if currentNode.childNode[i]:
                                count += 1
                if count > 1:
                        lastBranchNode = currentNode
                        lastBrachChar = c
                currentNode = currentNode.childNode[ord(c) - ord('a')]

        count = 0
        for i in range(26):
                if currentNode.childNode[i]:
                        count += 1

        # Case 1: The deleted word is a prefix of other words in Trie.
        if count > 0:
                currentNode.wordCount -= 1
                return True

        # Case 2: The deleted word shares a common prefix with other words in Trie.
        if lastBranchNode:
                lastBranchNode.childNode[ord(lastBrachChar) - ord('a')] = None
                return True
        # Case 3: The deleted word does not share any common prefix with other words in Trie.
        else:
                root.childNode[ord(word[0]) - ord('a')] = None
                return True
```

3.时间复杂度分析

插入操作：O(n)

搜索操作：O（n）

删除操作：O（n）

其中n指的是字符串的长度

十、笔试问题

[1] 一个叶子结点是二叉树前序序列的最后一个结点，也不代表着一定是中序遍历结果的最后一个结点（考虑只有一个左子结点的情况）

[2] 在二叉树三种不同的周游序列中，叶子结点的相对次序不会发生改变

[3] 在映射抽象数据类型（ADT Map）的不同实现方法中，适合对动态查找表进行高效率查找的组织结构是二叉排序树

[4] 有n个结点的二叉排序树中，树高最小的二叉排序树的搜索效率最好

[5] 如果只想得到1000个元素的序列中最小的前5个元素，堆排序最快

[6] 最初建堆的时间复杂度为O(n)