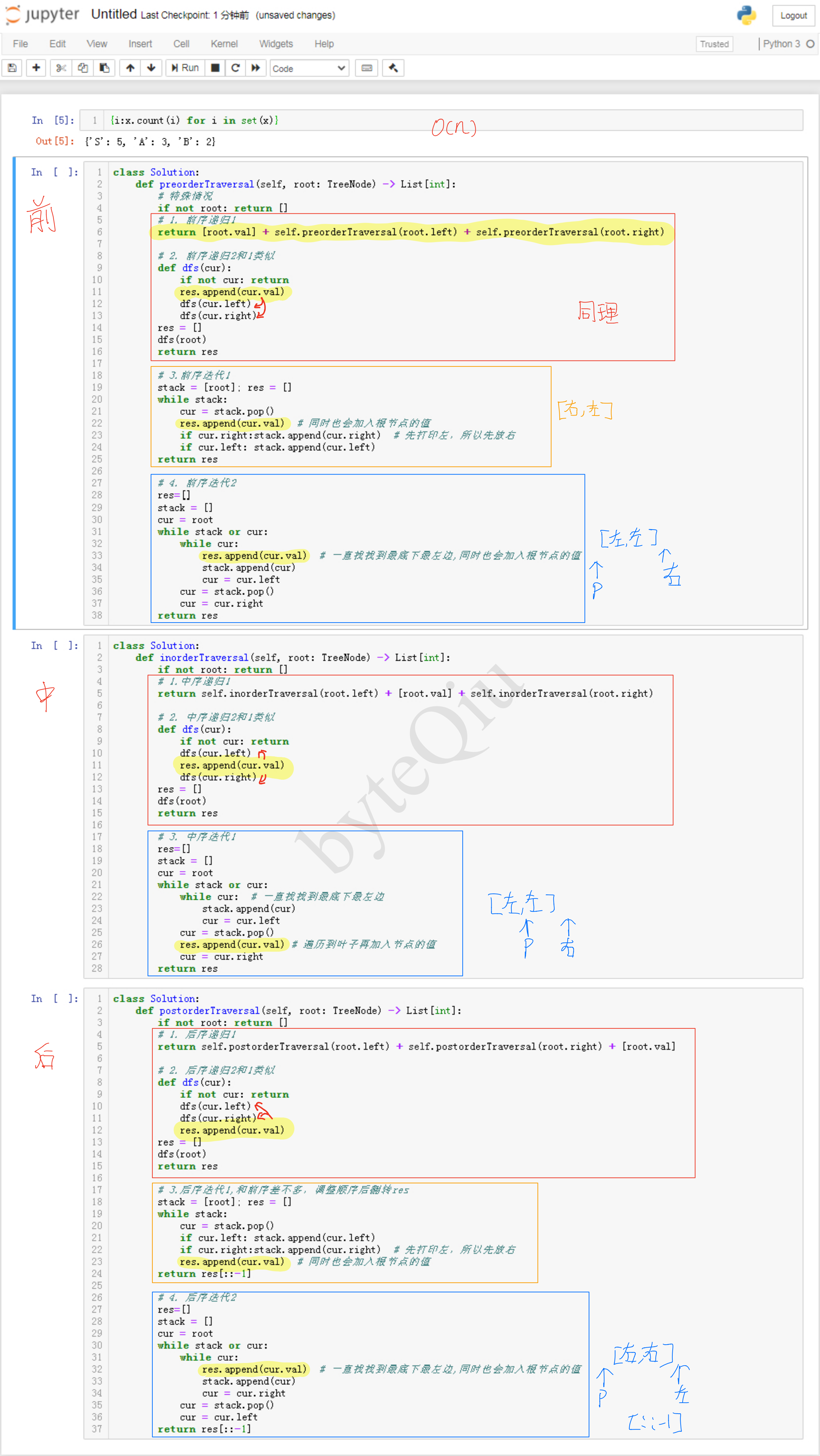

In [5]:

```
{i:x.count(i) for i in set(x)}
```

O(n)

Out[5]:

```
{'S': 5, 'A': 3, 'B': 2}
```

In [ ]:

前

```
class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        # 特殊情况
        if not root: return []
        # 1. 前序递归1
        return [root.val] + self.preorderTraversal(root.left) + self.preorderTraversal(root.right)

        # 2. 前序递归2和1类似
        def dfs(cur):
            if not cur: return
            res.append(cur.val)
            dfs(cur.left)
            dfs(cur.right)
        res = []
        dfs(root)
        return res

        # 3.前序迭代1
        stack = [root]; res = []
        while stack:
            cur = stack.pop()
            res.append(cur.val)  # 同时也会加入根节点的值
            if cur.right:stack.append(cur.right)  # 先打印左，所以先放右
            if cur.left: stack.append(cur.left)
        return res

        # 4. 前序迭代2
        res=[]
        stack = []
        cur = root
        while stack or cur:
            while cur:
                res.append(cur.val)  # 一直找找到最底下最左边,同时也会加入根节点的值
                stack.append(cur)
                cur = cur.left
            cur = stack.pop()
            cur = cur.right
        return res
```

同理

[右,左]

[左,左]
↑ P
↑ 右

In [ ]:

中

```
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        if not root: return []
        # 1.中序递归1
        return self.inorderTraversal(root.left) + [root.val] + self.inorderTraversal(root.right)

        # 2. 中序递归2和1类似
        def dfs(cur):
            if not cur: return
            dfs(cur.left)
            res.append(cur.val)
            dfs(cur.right)
        res = []
        dfs(root)
        return res

        # 3. 中序迭代1
        res=[]
        stack = []
        cur = root
        while stack or cur:
            while cur:  # 一直找找到最底下最左边
                stack.append(cur)
                cur = cur.left
            cur = stack.pop()
            res.append(cur.val) # 遍历到叶子再加入节点的值
            cur = cur.right
        return res
```

[左,左]
↑ P
↑ 右

In [ ]:

后

```
class Solution:
    def postorderTraversal(self, root: TreeNode) -> List[int]:
        if not root: return []
        # 1. 后序递归1
        return self.postorderTraversal(root.left) + self.postorderTraversal(root.right) + [root.val]

        # 2. 后序递归2和1类似
        def dfs(cur):
            if not cur: return
            dfs(cur.left)
            dfs(cur.right)
            res.append(cur.val)
        res = []
        dfs(root)
        return res

        # 3.后序迭代1,和前序差不多, 调整顺序后翻转res
        stack = [root]; res = []
        while stack:
            cur = stack.pop()
            if cur.left: stack.append(cur.left)
            if cur.right:stack.append(cur.right)  # 先打印左，所以先放右
            res.append(cur.val)  # 同时也会加入根节点的值
        return res[::-1]

        # 4. 后序迭代2
        res=[]
        stack = []
        cur = root
        while stack or cur:
            while cur:
                res.append(cur.val)  # 一直找找到最底下最左边,同时也会加入根节点的值
                stack.append(cur)
                cur = cur.right
            cur = stack.pop()
            cur = cur.left
        return res[::-1]
```

[右,右]
↑ P
↑ 左
[::-1]

In [3]:

```
# 144 前序
def preorderTraversal(root):
    # 递归1 中左右
    if root == None: return []
    return [root.val]+self.preorderTraversal(root.left)+self.preorderTraversal(root.right)

    # 递归二
    res = []
    def dfs(root):
        if root == None : return
        res.append(root.val)
        if root.left: dfs(root.left)
        if root.right:dfs(root.right)
    dfs(root)
    return res

    # 迭代法
    if not root: return []
    res = []
    stack = [root]
    while stack:
        tempNode = stack.pop()
        res.append(tempNode.val)
        if tempNode.right: stack.append(tempNode.right)
        if tempNode.left: stack.append(tempNode.left)
    return res

# 145 后序
def postorderTraversal(root):
    # 递归1
    if not root: return []
    return self.postorderTraversal(root.left)+self.postorderTraversal(root.right)+[root.val]

    # 递归2
    res = []
    def dfs(root):
        if not root: return
        if root.left: dfs(root.left)
        if root.right: dfs(root.right)
        res.append(root.val)
    dfs(root)
    return res

    # 迭代1
    if not root: return []
    res = []
    stack = [root]
    while stack:
        tempNode = stack.pop() # 取了就=拿了
        res.append(tempNode.val)
        if tempNode.left : stack.append(tempNode.left) # 栈概念
        if tempNode.right: stack.append(tempNode.right)
    return res[::-1]  # 后向，中右左-> 左右中

# 94 中序
def inorderTraversal(root):
    # 递归1
    if not root: return []
    return self.inorderTraversal(root.left)+[root.val]+self.inorderTraversal(root.right)

    # 递归2
    res = []
    def dfs(root):
        if not root: return
        if root.left:dfs(root.left)
        res.append(root.val)
        if root.right:dfs(root.right)
    dfs(root)
    return res

    # 迭代 - 结合动态理解
    if not root: return []
    res = []
    stack = []
    cur = root
    while stack or cur:
        while cur:
            stack.append(cur)
            cur = cur.left  # 左到底
        cur = stack.pop() # 再取
        res.append(cur.val)
        cur = cur.right # 再右
    return res

# 116 层序遍历
def levelOrder(root):
    if not root:return []
    queue = [root]
    res = []
    while queue:
        temp = []
        length = len(queue)
        for i in range(length):   # 用for不用另外开空间
            tempNode = queue.pop(0)
            temp.append(tempNode.val)
            if tempNode.left : queue.append(tempNode.left)
            if tempNode.right: queue.append(tempNode.right)
        res.append(temp)
    return res

# 层序遍历右指针
def connect(root):
    if not root: return None
    queue = [root]
    while queue:
        length = len(queue)
        for i in range(length):
            tempNode = queue.pop(0)
            if tempNode.left: queue.append(tempNode.left)
            if tempNode.right:queue.append(tempNode.right)
            if i == length - 1: break  # 结束一条链表
            tempNode.next = queue[0]  # 连接节点
    return root

# 226 镜像二叉树
def invertTree(root):
    if not root: return root
    # 层次遍历
    queue = [root]
    while queue:
        length = len(queue)
        for i in range(length):
            tempNode = queue.pop(0)
            tempNode.left,tempNode.right = tempNode.right,tempNode.left  # 只改变了一下
            if tempNode.left:queue.append(tempNode.left)
            if tempNode.right:queue.append(tempNode.right)
    return root

    # 递归(前序)
    root.left,root.right = root.right,root.left
    self.invertTree(root.left)
    self.invertTree(root.right)

    #迭代(深度优先), 前序
    stack = [root]
    while stack:
        node = stack.pop()
        node.left,node.right = node.right,node.left  # 只改变这一点
        if node.right:stack.append(node.right)
        if node.left:stack.append(node.left)
    return root

# 227 对称二叉树
def isSymmetric(root):
    if not root: return False

    # 递归
    def compare(left,right):
        # 四种情况: 00,10,01,11
        if not left and not right: return True
        if not left or not right or left.val != right.val: return False
        return compare(left.left,right.right) and compare(left.right,right.left)
    return compare(root.left,root.right)

    # 队列
    queue = [root.left,root.right]
    while queue:
        leftNode = queue.pop(0)
        rightNode = queue.pop(0)
        if not leftNode and not rightNode: continue
        if not leftNode or not rightNode or leftNode.val != rightNode.val: return False
        queue.append(leftNode.left)
        queue.append(rightNode.right)
        queue.append(leftNode.right)
        queue.append(rightNode.left)
    return True

# 104 二叉树的最大深度
def maxDepth(root):
    if not root: return 0
    # 递归
    return 1+ max(self.maxDepth(root.left),self.maxDepth(root.right))

    # 迭代
    # 层序遍历模板
    queue = [root]
    res = 0
    while queue:
        length = len(queue)
        res += 1  # 接下来会遍历每一层
        for i in range(length):
            tempNode = queue.pop(0)
            if tempNode.left:queue.append(tempNode.left)
            if tempNode.right:queue.append(tempNode.right)
    return res

# 559 N叉树的最大深度——层序遍历
def maxDepth(self, root: 'Node') -> int:
    # 递归
    if not root: return 0
    res = 0
    for i in range(len(root.children)):
        res = max(res,self.maxDepth(root.children[i]))
    return 1+ res

    # 迭代法
    queue = [root]
    res = 0
    while queue:
        res += 1
        length = len(queue)
        for i in range(length):
            tempNode = queue.pop(0)
            if tempNode.children: queue.extend(tempNode.children)
    return res

# 111 二叉树的最小深度
def minDepth(root):
    # 递归法
    if not root: return 0
    if root.left and not root.right: return 1+ self.minDepth(root.left)
    if root.right and not root.left: return 1+ self.minDepth(root.right)
    return 1+ min(self.minDepth(root.left),self.minDepth(root.right))  # 两个都非空

    # 迭代法，层序遍历
    if not root: return 0
    queue = [root]
    res = 0
    while queue:
        length = len(queue)
        res += 1
        for i in range(length):
            tempNode = queue.pop(0)
            if tempNode.left: queue.append(tempNode.left)
            if tempNode.right:queue.append(tempNode.right)
            if tempNode.left == None and tempNode.right == None:  return res #退出条件
    return res

# 222 完全二叉树的根节点
def countNodes(root):
    # 递归
    if not root: return 0
    return 1+self.countNodes(root.left)+self.countNodes(root.right)

    # 普通二叉树的迭代法——层次遍历
    queue = [root]
    res = 0
    while queue:
        length = len(queue)
        res += length
        for i in range(length):
            tempNode = queue.pop(0)
            if tempNode.left: queue.append(tempNode.left)
            if tempNode.right:queue.append(tempNode.right)
    return res

    # 利用完全二叉树的性质
    def countDepth(root):
        # 计算最大深度
        r = 0
        while root:
            root = root.left
            r += 1
        return r
    if not root: return 0
    leftDepth = countDepth(root.left)
    rightDepth = countDepth(root.right)
    if leftDepth == rightDepth:
        # 左满右完全
        return 2**leftDepth + self.countNodes(root.right) # 不用减1,因为加了根节点
    else:
        # 右满左完全
        return 2**rightDepth + self.countNodes(root.left)

# 110 判断平衡二叉树
def isBalanced(root):
    # 递归法
    def getdepth(root):
        # 返回平衡树的高度，后续遍历
        if not root : return 0
        leftDepth = getdepth(root.left)
        rightDepth = getdepth(root.right)
        if leftDepth == -1 or rightDepth == -1 : return -1 # 有一边不是平衡树了
        return -1 if abs(leftDepth-rightDepth) > 1 else 1+max(leftDepth,rightDepth)
    return getdepth(root) != -1

# 257 二叉树的所有路径
def binaryTreePaths(root):
    # 回溯
    if not root: return
    res = []
    path = [str(root.val)]
    def backtrak(root):
        # 结束条件 叶子节点
        if not root: return
        if not root.left and not root.right : res.append(’->’.join(path[:])) # 叶子节点了
        if root.left:
            path.append(str(root.left.val))
            backtrak(root.left)
            path.pop()
        if root.right:
            path.append(str(root.right.val))
            backtrak(root.right)
            path.pop()
    backtrak(root)
    return res

# 100 相同的树
def isSameTree(p, q):
    # 层次遍历 56 % 60 ,[1,2][1,null,2]不过
    # 二叉树镜像,递归判断
    if not p and not q: return True
    if not p or not q : return False
    if p.val != q.val: return False
    return self.isSameTree(p.left,q.left) and self.isSameTree(p.right,q.right)

# 404 左叶子之和
def sumOfLeftLeaves(root):
    # 递归
    self.res = 0
    def findleft(root):
        if not root: return 0
        # 不能if not left: return 0,这排除了有右子树的情况
        if root.left and not root.left.left and not root.left.right:
            self.res += root.left.val  # 修改全局变量
        findleft(root.left)
        findleft(root.right)
    findleft(root)
    return self.res # 返回全局变量

# 513 最左边的叶子的值
def findBottomLeftValue(root):
    # 层序遍历
    queue = [root]
    while queue:
        length = len(queue)
        for i in range(length):
            tempNode = queue.pop(0)
            if i == 0:
                temp = tempNode.val
            if tempNode.left: queue.append(tempNode.left)
            if tempNode.right: queue.append(tempNode.right)
    return temp

# 112 路径总和
def hasPathSum(root, targetSum):
    # 递归
    if not root: return False
    targetSum -= root.val
    if not root.left and not root.right and targetSum == 0: return True
    return self.hasPathSum(root.left,targetSum) or self.hasPathSum(root.right,targetSum)

# 113 路径总和II
def pathSum(root,targetSum):
    if not root: return []
    res = []
    path = [root.val]
    def backtrack(cur,count):
        if not cur.left and not cur.right and count == 0: return res.append(path[:]) #
        if not cur.left and not cur.right: return  #
        if cur.left:
            path.append(cur.left.val)  #
            count -= cur.left.val
            backtrack(cur.left,count)
            path.pop()  #
            count += cur.left.val
        if cur.right:
            path.append(cur.right.val) #
            count -= cur.right.val
            backtrack(cur.right,count)
            path.pop()  #
            count += cur.right.val
```

```python
    backtrack(root,targetSum-root.val)
    return res

# 106 中序和后续构造二叉树
def buildTree(inorder, postorder):
    # 没有重复元素
    if not len(inorder) or not len(postorder): return
    rootval = postorder.pop()
    root = TreeNode(rootval)
    splitin = inorder.index(rootval)
    inleft = inorder[:splitin]
    inright = inorder[splitin+1:]
    postleft = postorder[:splitin]
    postright = postorder[splitin:]
    root.left = self.buildTree(inleft,postleft)
    root.right = self.buildTree(inright,postright)
    return root

# 105 前序和中序构造二叉树
def buildTree(preorder, inorder):
    if not len(inorder) or not len(preorder): return
    rootval = preorder.pop(0)
    root = TreeNode(rootval)
    splitpoint = inorder.index(rootval)
    root.left = self.buildTree(preorder[:splitpoint],inorder[:splitpoint])
    root.right = self.buildTree(preorder[splitpoint:],inorder[splitpoint+1:])
    return root

# 654 最大的二叉树
def constructMaximumBinaryTree(nums):
    if not nums: return
    rootval = max(nums)
    root = TreeNode(rootval)
    maxpoint = nums.index(rootval)
    leftnums = nums[:maxpoint]
    rightnums = nums[maxpoint+1:]
    root.left = self.constructMaximumBinaryTree(leftnums)
    root.right = self.constructMaximumBinaryTree(rightnums)
    return root

# 617 合并二叉树
def mergeTrees(root1, root2):
    if not root1 : return root2
    if not root2 : return root1
    if not root1 and not root2: return
    rootval = root1.val+ root2.val
    root = TreeNode(rootval)
    root.left = self.mergeTrees(root1.left,root2.left)
    root.right = self.mergeTrees(root1.right,root2.right)
    return root

# 700 二叉搜索树中搜索
def searchBST(root, val):
    if not root : return
    if root.val == val: return root
    # return self.searchBST(root.left,val) or self.searchBST(root.right,val)
    # 左子树所有节点均小于根节点
    # 右子树所有节点均大于根节点
    if root.val > val: return self.searchBST(root.left,val)
    if root.val < val: return self.searchBST(root.right,val)
    return None

    # 迭代
    while root:
        if root.val == val: return root
        if root.val < val: root =  root.right
        else: root = root.left
    return

# 98 验证二叉树
def isValidBST(root):
    # 中序遍历输出有序数组
    if not root: return True
    pre = float("-inf") # 保存前一个访问节点的值
    def traversal(root):
        nonlocal pre
        if not root: return True
        if not traversal(root.left):return False
        if root.val <= pre: return False
        else: pre = root.val
        if not traversal(root.right): return False
        return True
    return traversal(root)

    # 纯递归 + 判断数值
    if not root: return
    def traversal(root):
        if not root: return []
        return traversal(root.left)+[root.val]+traversal(root.right)
    res = traversal(root)
    for i in range(1,len(res)):
        if res[i] <= res[i-1]: return False
    return True

    # 迭代中序遍历
    if not root: return True
    stack = []
    pre = float('-inf') # 保存前一个访问节点值
    cur = root
    while stack or cur:
        while cur:
            stack.append(cur)
            cur = cur.left
        cur = stack.pop()
        if cur.val <= pre: return False
        else: pre = cur.val
        cur = cur.right
    return True

# 530 二叉搜索树的最小绝对差
def getMinimumDifference(root):
    # 递归
    res = float('inf')
    pre = float('-inf')
    def traversal(root):
        nonlocal res,pre
        if not root: return
        traversal(root.left)
        res = min(res,root.val-pre)
        pre = root.val
        traversal(root.right)
    traversal(root)
    return res

    # 迭代
    res = float(’inf’)
    pre = float(’-inf’)
    stack = []
    cur = root
    while stack or cur:
        while cur:
            stack.append(cur)
            cur = cur.left
        cur = stack.pop()
        res = min(res,cur.val-pre)
        pre = cur.val
        cur = cur.right
    return res

# 501 二叉搜索树的众数
def findMode(self, root: TreeNode) -> List[int]:
    # 普特二叉树对待
    dic = {}
    def search(root):
        if not root: return
        if root.val in dic: dic[root.val] += 1
        else: dic[root.val] = 1
        search(root.left)
        search(root.right)
    search(root)
    sdict = sorted(dic.items(),key = lambda x:x[1],reverse = True) # 排序字典
    res = []
    for key,val in dic.items():
        if val == sdict[0][1]:
            res.append(key)
    return res

    # 二叉搜索树+迭代遍历
    res = [] # 保存结果
    maxCount = 0 # 统计最大频率
    count = 0 # 统计当前频率
    stack = []
    cur = root
    pre = float(’-inf’)
    while stack or cur:
        while cur:
            stack.append(cur)
            cur = cur.left
        cur = stack.pop()
        print(res,maxCount)
        if cur.val == pre: count += 1
        else: count = 1
        if count == maxCount:
            res.append(cur.val)
        if count > maxCount:
            maxCount = count
            res = [cur.val] # 放弃之前所有元素
        pre = cur.val # 记录前面的值
        cur = cur.right
    return res

# 236 二叉树的最近公共祖先
def lowestCommonAncestor(root, p, q):
    # 从下往上递归
    if not root or root == q or root == p: return root
    left = self.lowestCommonAncestor(root.left,p,q)
    right = self.lowestCommonAncestor(root.right,p,q)
    if left and right: return root  # 有返回值，只有一个两边都有，后面都是一个有一个没有，相
    if not left and right : return right
    elif left and not right : return left
    else: return

# 235 二叉搜索树的最近公共祖先
def lowestCommonAncestor(root, p, q):
    if root.val > p.val and root.val > q.val:
        return self.lowestCommonAncestor(root.left,p,q) # 在左边
    elif root.val < p.val and root.val < q.val:
        return self.lowestCommonAncestor(root.right,p,q) # 在右边
    else: return root  # 找到了，回传

# 701 二叉搜索树的插入
def insertIntoBST(root, val):
    if not root:
        # 能走到none这个位置
        node = TreeNode(val)
        return node
    if root.val > val : root.left = self.insertIntoBST(root.left,val)
    if root.val < val: root.right = self.insertIntoBST(root.right,val)
    return root

# 450 删除二叉树的节点
def deleteNode(root, key):
    if not root: return root
    if root.val == key:
        if not root.left: return root.right
        elif not root.right: return root.left # 已经包含都为空情况了
        else:
            cur = root.right
            while cur.left:
                cur = cur.left
            cur.left = root.left
            temp = root
            root = root.right
            del temp
            return root
    if root.val > key: root.left = self.deleteNode(root.left,key)
    if root.val < key: root.right = self.deleteNode(root.right,key)
    return root

# 669 修剪二叉搜索树
def trimBST(root, low, high):
    if not root : return
    if root.val < low:
        right = self.trimBST(root.right,low,high)
        return right
    if root.val > high:
        left = self.trimBST(root.left,low,high)
        return left
    root.left = self.trimBST(root.left,low,high)
    root.right = self.trimBST(root.right,low,high)
    return root

# 108 构建一课二叉搜索树
def sortedArrayToBST(nums):
    def traversal(nums,left,right):
        if left > right: return None
        mid = left + (right-left) // 2
        root = TreeNode(nums[mid])
        root.left = traversal(nums,left,mid-1)  # 不能是nums[:mid]
        root.right = traversal(nums,mid+1,right)
        return root
    root = traversal(nums,0,len(nums)-1)
    return root

# 538 把二叉搜索树转换成累加树
def convertBST(root):
    # 递归 反中序遍历
    pre = 0
    def traversal(cur):
        nonlocal pre
        if not cur: return
        traversal(cur.right)
        cur.val += pre
        pre = cur.val
        traversal(cur.left)
    traversal(root)
    return root
    # 迭代
    if not root: return
    pre = 0
    stack = []
    cur = root
    while stack or cur:
        while cur:
            stack.append(cur)
            cur = cur.right
        cur = stack.pop()
        cur.val += pre  # 这两行
        pre = cur.val   # 不同而已
        cur = cur.left
    return root
```

In [140]:

```
# 698 桶装法
def canPartitionKSubsets(nums, k):
    # 每个子集的和为 nums / k
    if sum(nums) % k != 0: return False
    Sum = int(sum(nums) / k)
    bucker = [0]*k  # k个桶
    def backtrack(nums,startIndex,k):
        nonlocal Sum
        if startIndex == len(nums) : return True
        for i in range(k):
            if  bucker[i] + nums[startIndex] <= Sum:
                bucker[i] += nums[startIndex]
                if backtrack(nums,startIndex+1,k): return True
                bucker[i] -= nums[startIndex]
                if bucker[i] == 0: return False
        return False
    nums.sort(reverse=True)
    return backtrack(nums,0,k)
```