257. Binary Tree Paths
We will traverse the tree in pre order. When it's leaf node, we will append the current one to our res list. TC is O(n)

```
class Solution:
    def binaryTreePaths(self, root: TreeNode) -> List[str]:
        self.res = []
        if not root:
            return
        def traverse(node, cur):
            if not node.left and not node.right:
                self.res.append(cur + str(node.val))
                return
            if node.left:
                traverse(node.left, cur + str(node.val) + '->')
            if node.right:
                traverse(node.right, cur + str(node.val) + '->')
        traverse(root, '')
        return self.res
```

236. Lowest Common Ancestor of a Binary Tree
We will traverse the tree in post order to get left and right whether there is a p or q. If left and right both exist, we will return root or we will return p or q. TC is O(n)

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':
        if not root:
            return None
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        if root == q or root == p:
            return root
        elif left and right:
            return root
        elif left or right:
            return left or right
        return None
```

235. Lowest Common Ancestor of a Binary Search Tree
It's similar to the previous one but we will return node when p and q are on its right branch and left branch. TC is O(n)

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':
```

```
        if not root:
            return None
        if q.val <= root.val <= p.val or p.val <= root.val <= q.val:
            return root
        if root.val >= p.val and root.val >= q.val:
            right = None
        else:
            right = self.lowestCommonAncestor(root.right, p, q)

        if root.val <= p.val and root.val <= q.val:
            left = None
        else:
            left = self.lowestCommonAncestor(root.left, p, q)

        if root == p or root == q:
            return root
        if left and right:
            return root
        if left or right:
            return left or right
        return None
```

297. Serialize and Deserialize Binary Tree
We will use preorder traversal to encode all nodes and then decode it using recursion. TC is O(n)

```
from collections import deque
class Codec:

    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        res = []
        def helper(node):
            if not node:
                res.append('#')
                return
            res.append(str(node.val))
            helper(node.left)
            helper(node.right)
        helper(root)
```

```
        return ','.join(res)

    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        res = deque(data.split(','))
        def helper():
            if not res:
                return None
            val = res.popleft()
            if val == '#':
                return None
            node = TreeNode(int(val))
            node.left = helper()
            node.right = helper()
            return node
        return helper()
```

508. Most Frequent Subtree Sum

We will use post order traversal to go through all nodes here and use a hashmap to record all subtree sums. Then we could get most frequent keys from dict. TC is O(n)

```
from collections import defaultdict
class Solution:
    def findFrequentTreeSum(self, root: TreeNode) -> List[int]:
        memo = defaultdict(int)
        res = []
        if not root:
            return res
        def traverse(node):
            if not node:
                return 0
            left = traverse(node.left)
            right = traverse(node.right)
            memo[node.val + left + right] += 1
            return node.val + left + right
        traverse(root)
        max_val = max(memo.values())
        for k, v in memo.items():
            if v == max_val:
                res.append(k)
```

```
return res
```