# Notebook

August 6, 2024

[ ]:
```python
# Lowest Common Ancestor Problems

# LCA in Binary Search Tree
# https://github.com/doocs/leetcode/blob/main/solution/0200-0299/0235.Lowest%20Common%20Ancestor%20of%20a%20Binary%20Search%20Tree/README_EN.md

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        while 1:
            if root.val < min(p.val, q.val):
                root = root.right
            elif root.val > max(p.val, q.val):
                root = root.left
            else:
                return root

# LCA in Binary Tree
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        if root in (None, p, q): return root
        lans = self.lowestCommonAncestor(root.left, p, q)
        rans = self.lowestCommonAncestor(root.right, p, q)
        return root if lans and rans else (lans or rans)

# LCA in Binary Tree where p and q can be None
# Can you find the LCA traversing the tree, without checking nodes existence?

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        def dfs(root, p, q):
            if root is None:
                return False
            l = dfs(root.left, p, q)
            r = dfs(root.right, p, q)
```

```
            nonlocal ans
            if l and r:
                ans = root
            if (l or r) and (root.val == p.val or root.val == q.val):
                ans = root
            return l or r or root.val == p.val or root.val == q.val

        ans = None
        dfs(root, p, q)
        return ans

# Find LCA, instead of giving root, we will have "parent" pointer, for node
# https://github.com/doocs/leetcode/blob/main/solution/1600-1699/1650.
 ↪Lowest%20Common%20Ancestor%20of%20a%20Binary%20Tree%20III/README_EN.md

## Traver with p and add all parents to set, then traverse with q and check if␣
 ↪parent is in set and return
class Solution:
    def lowestCommonAncestor(self, p: "Node", q: "Node") -> "Node":
        vis = set()
        node = p
        while node:
            vis.add(node)
            node = node.parent
        node = q
        while node not in vis:
            node = node.parent
        return node

## Two pointers
class Solution:
    def lowestCommonAncestor(self, p: 'Node', q: 'Node') -> 'Node':
        a, b = p, q
        while a != b:
            a = a.parent if a.parent else q
            b = b.parent if b.parent else p
        return a

# https://github.com/doocs/leetcode/blob/main/solution/1600-1699/1676.
 ↪Lowest%20Common%20Ancestor%20of%20a%20Binary%20Tree%20IV/README_EN.md
# LCA of [p1, p2, p3, p4, ...] in Binary Tree

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', nodes: 'List[TreeNode]')␣
 ↪-> 'TreeNode':
        def dfs(root):
            if root is None or root.val in s:
```

```
                return root
            left, right = dfs(root.left), dfs(root.right)
            if left and right:
                return root
            return left or right

        s = {node.val for node in nodes}
        return dfs(root)


# https://leetcode.com/problems/lowest-common-ancestor-of-deepest-leaves
# LCA of Deepest Leaves
# use dfs to get depth and lca

class Solution:
    def lcaDeepestLeaves(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        def helper(root):
            if not root : return 0, None
            h1, lca1 = helper(root.left)
            h2, lca2 = helper(root.right)
            if h1 > h2: return h1 + 1, lca1
            if h1 < h2: return h2 + 1, lca2
            return h1 + 1, root
        return helper(root)[1]
```

[ ]:
```
# Path Sum

# https://leetcode.com/problems/path-sum/description/
class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        def dfs(root, target):
            if root == None: return False
            if root.val == target and root.left == None and root.right == None:␣
 ↪return True
            return dfs(root.left, target-root.val) or dfs(root.right,␣
 ↪target-root.val)
        return dfs(root, targetSum)

# https://leetcode.com/problems/path-sum-ii/
# return all paths which sum to target
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) ->␣
 ↪List[List[int]]:
        def dfs(root, s):
            if root is None:
                return
            s += root.val
```

```
            t.append(root.val)
            if root.left is None and root.right is None and s == targetSum:
                ans.append(t[:]) # copy
            dfs(root.left, s)
            dfs(root.right, s)
            t.pop()

        ans = []
        t = []
        dfs(root, 0)
        return ans

# https://leetcode.com/problems/path-sum-iii/
# find paths with sum to target, can start from any node

## Brute force
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        def dfs(node, s):
            if node is None:
                return 0
            s += node.val
            return (s == targetSum) + dfs(node.left, s) + dfs(node.right, s)

        return dfs(root, 0) + (self.pathSum(root.left, targetSum) if root else␣
↪0) + (self.pathSum(root.right, targetSum) if root else 0)

## optimized
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        def dfs(node, s):
            if node is None:
                return 0
            s += node.val
            ans = cnt[s - targetSum]
            cnt[s] += 1
            ans += dfs(node.left, s)
            ans += dfs(node.right, s)
            cnt[s] -= 1
            return ans

        cnt = Counter({0: 1})
        return dfs(root, 0)

# https://leetcode.com/problems/binary-tree-maximum-path-sum/description/
# get maximum path sum in binary tree
```

```python
class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        max_path = float('-inf')
        def get_max_path(node):
            nonlocal max_path
            if node is None: return 0

            gain_on_left = max(get_max_path(node.left), 0)
            gain_on_right = max(get_max_path(node.right), 0)
            curr_max_path = node.val + gain_on_left + gain_on_right
            max_path = max(max_path, curr_max_path)

            return node.val + max(gain_on_left, gain_on_right)

        get_max_path(root)
        return max_path

# https://leetcode.com/problems/longest-path-with-different-adjacent-characters/
 ↪description/
# Longest Path with Different Adjacent Characters, here tree is not given␣
 ↪instead list of node whose parent is given
#Input: parent = [-1,0,0,1,1,2], s = "abacbe"  Output: 3

# create child dictionary and then do dfs to get longest path
# maintain two max variables to get longest path
class Solution:
    def longestPath(self, parent: List[int], s: str) -> int:
        child = defaultdict(list)
        for i in range(1, len(parent)):
            child[parent[i]].append(i)

        def dfs(curr_node):
            nonlocal ans
            mx1, mx2 = 0, 0
            for child_node in child[curr_node]:
                lenmx = dfs(child_node)
                if s[curr_node] == s[child_node]:
                    continue
                if lenmx > mx1:
                    mx2, mx1 = mx1, lenmx
                else:
                    mx2 = max(mx2, lenmx)
            ans = max(ans, mx1 + mx2 + 1)
            return 1 + mx1

        ans = 1
        dfs(0)
```

```
        return ans

# https://github.com/doocs/leetcode/blob/main/solution/2300-2399/2378.
  ↪Choose%20Edges%20to%20Maximize%20Score%20in%20a%20Tree/README_EN.md
```

[ ]:
```
# Binary Tree Vertical Order Traversal
# here when values are at same level, we need to sort them in ascending order

class Solution:
    def verticalTraversal(self, root: Optional[TreeNode]) -> List[List[int]]:
        if root is None:
            return []
        q = deque([(root, 0)])
        d = defaultdict(list)
        while q:
            tq = deque()
            td = defaultdict(list)
            for _ in range(len(q)):
                root, offset = q.popleft()
                td[offset].append(root.val)
                if root.left:
                    tq.append((root.left, offset - 1))
                if root.right:
                    tq.append((root.right, offset + 1))
            for i in td:
                d[i].extend(sorted(td[i]))
            q = tq
        return [v for _, v in sorted(d.items())]

# # https://leetcode.com/problems/binary-tree-right-side-view/description/
# Binary Tree Right Side View
# dfs right first, then left, and keep track of level

class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        def dfs(node, level):
            if node is None:
                return
            if level == len(ans):
                ans.append(node.val)
            dfs(node.right, level + 1)
            dfs(node.left, level + 1)
        ans = []
        dfs(root, 0)
        return ans

# Bottom View of Binary Tree
```

```
class Solution:
    def bottomView(self, root):
        # code here
        mp = {}
        dq = deque([(0, root)])

        while dq:
            dist, curr = dq.popleft()
            mp[dist] = curr.data
            if curr.left != None:
                dq.append((dist-1, curr.left))
            if curr.right != None:
                dq.append((dist+1, curr.right))

        mpitems = sorted(mp.items())

        return [v for k, v in mpitems]
```

[ ]:
```
# https://leetcode.com/problems/serialize-and-deserialize-binary-tree/
 ↪description/
# use preorder traversal to serialize and deserialize

class Codec:
    def serialize(self, root):
        if root == None: return "#"
        return str(root.val) + "," + self.serialize(root.left) + "," + self.
 ↪serialize(root.right)

    def deserialize(self, data):

        nodes = data.split(",")
        self.idx = 0
        def dfs():
            if self.idx == len(nodes): return None
            nodeVal = nodes[self.idx]
            self.idx += 1
            if nodeVal == "#": return None
            root = TreeNode(int(nodeVal))
            root.left = dfs()
            root.right = dfs()
            return root
        return dfs()

# https://leetcode.com/problems/serialize-and-deserialize-bst/description/
# serialize and deserialize binary search tree
```

```python
class Codec:

    def serialize(self, root: Optional[TreeNode]) -> str:
        if not root:
            return ''
        return str(root.val) + ',' + self.serialize(root.left) + ',' + self.
 ↪serialize(root.right)


    def deserialize(self, data: str) -> Optional[TreeNode]:
        deq = deque(int(val) for val in data.split(',') if val)

        def build_tree(lower_bound, upper_bound):
            if deq and lower_bound < deq[0] < upper_bound:
                val = deq.popleft()
                return TreeNode(val, build_tree(lower_bound, val),␣
 ↪build_tree(val, upper_bound))

        return build_tree(float('-inf'), float('inf'))

# Serialize and Deserialize N-ary Tree
# similar to binary tree, but we need to store number of children for each node␣
 ↪, so we store count of children

class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []

class Codec:
    def serialize(self, root: 'Node') -> str:
        """Encodes an N-ary tree to a single string."""
        def serial(node):
            if not node:
                return "#,"
            s = str(node.val) + ","
            s += str(len(node.children)) + ","
            for child in node.children:
                s += serial(child)
            return s

        return serial(root)

    def deserialize(self, data: str) -> 'Node':
        """Decodes your encoded data to tree."""
        if not data:
            return None
```

```
        def build_tree(queue):
            val = queue.popleft()
            if val == "#":
                return None
            node = Node(int(val))
            children_count = int(queue.popleft())
            node.children = []
            for _ in range(children_count):
                node.children.append(build_tree(queue))
            return node

        queue = deque(data.split(","))
        return build_tree(queue)


# https://leetcode.com/problems/construct-string-from-binary-tree/description/
# convert tree to string 1(2(4))(3)

class Solution:
    def tree2str(self, root: Optional[TreeNode]) -> str:
        res = []
        def dfs(root):
            if root is None: return
            res.append(str(root.val))
            if root.left is None and root.right is None:
                return
            res.append('(')
            dfs(root.left)
            res.append(')')
            if root.right:
                res.append('(')
                dfs(root.right)
                res.append(')')
        dfs(root)
        return ''.join(res)
```

[ ]:
```
# https://leetcode.com/problems/find-duplicate-subtrees/description/

# serialize the tree and store in hash map, then check if any subtree is␣
 ↪duplicate
class Solution:
    def findDuplicateSubtrees(self, root: Optional[TreeNode]) ->␣
 ↪List[Optional[TreeNode]]:
        ans = []
        mp = defaultdict(list)
        def serialize(root):
```

```
            if root == None: return ""
            s = "(" + serialize(root.left) + str(root.val) + serialize(root.
 ↪right) + ")"
            mp[s].append(root)
            return s
        serialize(root)
        for k, v in mp.items():
            if len(v) > 1: ans.append(v[0])
        return ans
```

[ ]:
```
# https://leetcode.com/problems/create-binary-tree-from-descriptions/
 ↪description/

# Input: descriptions = [[20,15,1],[20,17,0],[50,20,1],[50,80,0],[80,19,1]]
# 1 - left child, 0 - right child
# Output: [50,20,80,15,17,19]

class Solution:
    def createBinaryTree(self, descriptions: List[List[int]]) ->␣
 ↪Optional[TreeNode]:
        children = set()
        m = {}
        for p, c, l in descriptions:
            np = m.setdefault(p, TreeNode(p))
            nc = m.setdefault(c, TreeNode(c))
            if l:
                np.left = nc
            else:
                np.right = nc
            children.add(c)
        root = (set(m) - set(children)).pop()
        return m[root]

# create parent node and child node, then link them together, finally find the␣
 ↪root node if it is not in the children set
class Solution:
    def createBinaryTree(self, descriptions: List[List[int]]) ->␣
 ↪Optional[TreeNode]:
        mp = {}
        seen = set()
        for p, c, left in descriptions:
            if p not in mp: mp[p] = TreeNode(p)
            if c not in mp: mp[c] = TreeNode(c)
            if left: mp[p].left = mp[c]
            else: mp[p].right = mp[c]
            seen.add(c)
        for p, _, _ in descriptions:
```

```
            if p not in seen: return mp[p]
```

[ ]:
```
# https://leetcode.com/problems/
 ↪step-by-step-directions-from-a-binary-tree-node-to-another/description/
"""
Input: root = [5,1,2,3,null,6,4], startValue = 3, destValue = 6
Output: "UURL"
Explanation: The shortest path is: 3 → 1 → 5 → 2 → 6.
"""

# Find the lowest common ancestor of the two nodes, then traverse from the LCA␣
 ↪to the start node and dest node
# from start node to LCA, all the path should be 'U', from LCA to dest node it␣
 ↪should be traversed as it is

class Solution:
    def getDirections(self, root: Optional[TreeNode], startValue: int,␣
 ↪destValue: int) -> str:
        def lca(root, start, dest):
            if root is None: return None
            if root.val == start or root.val == dest: return root
            la = lca(root.left, start, dest)
            ra = lca(root.right, start, dest)
            if (la and ra): return root
            return la if la else ra

        def traverse(root, path, val):
            if not root:
                return False

            if root.val == val:
                return True

            path.append('L')
            if traverse(root.left, path, val):
                return True  # If found, then return
            path.pop()  # If not found, then backtrack

            path.append('R')
            if traverse(root.right, path, val):
                return True  # If found, then return
            path.pop()  # If not found, then backtrack
            return False

        lca_root = lca(root, startValue, destValue)
        path_start, path_dest = [], []
        traverse(lca_root, path_start, startValue)
```

```
        traverse(lca_root, path_dest, destValue)
        for i in range(len(path_start)):
            path_start[i] = 'U'

        return "".join(path_start + path_dest)
```

[ ]:
```
# https://leetcode.com/problems/delete-nodes-and-return-forest/description/

"""
Input: root = [1,2,3,4,5,6,7], to_delete = [3,5]
Output: [[1,2,null,4],[6],[7]]
"""

# first check if left and right child is in the delete set, if yes, then set it␣
 ↪to None
# if the current node is in the delete set, then add its left and right child␣
 ↪to the result
class Solution:
    def delNodes(self, root: Optional[TreeNode], to_delete: List[int]) ->␣
 ↪List[TreeNode]:
        ans = []
        dq = deque()
        del_set = set(to_delete)
        if root is None: return ans
        if root.val not in del_set: ans.append(root)
        dq.append(root)
        while dq:
            curr = dq.popleft()
            if curr.left != None: dq.append(curr.left)
            if curr.right != None: dq.append(curr.right)
            if curr.left != None and curr.left.val in del_set:
                curr.left = None
            if curr.right != None and curr.right.val in del_set:
                curr.right = None
            if curr.val in del_set:
                if curr.left != None: ans.append(curr.left)
                if curr.right != None: ans.append(curr.right)

        return ans
```

[ ]:
```
# https://leetcode.com/problems/number-of-good-leaf-nodes-pairs/description

"""
Input: root = [1,2,3,4,5,6,7], distance = 3
Output: 2
```

```
Explanation: The good pairs are [4,5] and [6,7] with shortest path = 2. The pair [4,6] is not good because the length of ther shortest path between them is 4.
"""
# use post order traversal to return the distances from the leaf nodes to the current node
# then use two for loops to find the good pairs

class Solution:
    def countPairs(self, root: TreeNode, distance: int) -> int:
        count = 0
        def dfs(node):
            nonlocal count
            if not node: return []
            if not node.left and not node.right: return [1]
            left = dfs(node.left)
            right = dfs(node.right)
            count += sum(l+r <= distance for l in left for r in right)
            return [n+1 for n in left + right if n+1 < distance]
        dfs(root)
        return count
```

[1]:
```
# https://leetcode.com/problems/sum-of-nodes-with-even-valued-grandparent/description/
# simple dfs traversal, if the current node is even, then add the sum of its grandchildren to the result

class Solution:
    def sumEvenGrandparent(self, root: TreeNode) -> int:
        ans = 0
        def dfs(root):
            nonlocal ans
            if root == None: return
            if root.val % 2 == 0 and root.left != None:
                if root.left.left != None:
                    ans += root.left.left.val
                if root.left.right != None:
                    ans += root.left.right.val
            if root.val % 2 == 0 and root.right != None:
                if root.right.left != None:
                    ans += root.right.left.val
                if root.right.right != None:
                    ans += root.right.right.val
            dfs(root.left)
            dfs(root.right)
        dfs(root)
```

```
        return ans
```

[ ]: