LeetCode: Solution Manual

Hanchao Zhang

Contents

1 Binary Tree

Question 1.1: LeetCode 144, preorder traversal

1. res append val
2. stack append right
3. stack append left

Listing 1: pre-order traversal dfs

```
class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        stack, res = [root], []

        while stack:
            node = stack.pop()
            if node:
                res.append(node.val)
                stack.append(node.right)
                stack.append(node.left)

        return res
```

Listing 2: pre-order traversal dp

```

class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        def dfs(node, res):
            if node:
                res.append(node.val)
                dfs(node.left, res)
                dfs(node.right, res)

        res = []
        dfs(root, res)
        return res
```

Question 1.2: LeetCode 94, inorder traversal

iterative solution

1. while cur or stack, stack and keep going to the left node to the last left node
2. cur = stack.pop and append to the res, then cur goes to the right

recursive solution

1. inorder(node.left, res)
2. res.append val
3. inorder(node.right, res)

Listing 3: inorder traversal iterative

```
    class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res = []
        stack = []
```

```python
        cur = root

        while cur or stack:
            while cur:
                stack.append(cur)
                cur = cur.left

            cur = stack.pop()
            res.append(cur.val)
            cur = cur.right

        return res
```

Listing 4: in-order traversal dp

```python
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:

        def inorder(node, res):
            if node:
                inorder(node.left, res)
                res.append(node.val)
                inorder(node.right, res)

        res = []
        inorder(root, res)

        return res
```

Question 1.3: LeeCode 145, Postorder Traversal

itervative

1. start with root stack, visited False, res empty
2. while stack, cur = stack.pop, v = visited.pop
3. if cur, if v: res.append(val)
4. if cur, if not v: stack cur, visited true, stack cur.right and cur.left and visited both false

recursive

1. postorder(node.left, res)
2. postorder(node.right, res)
3. res.append(node.val)

Listing 5: postorder traversal iterative

```python
    class Solution:
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        stack = [root]
        visited = [False]
        res = []

        while stack:
            cur = stack.pop()
            v = visited.pop()
            if cur:
                if v:
                    res.append(cur.val)
                else:
```

```
                stack.append(cur)
                visited.append(True)
                stack.append(cur.right)
                visited.append(False)
                stack.append(cur.left)
                visited.append(False)

    return res
```

Listing 6: postorder traversal dp

```
class Solution:
def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
    res = []

    def postorder(node, res):
        if node:
            postorder(node.left, res)
            postorder(node.right, res)
            res.append(node.val)

    postorder(root, res)
```

Question 1.4: LeetCode 102, level order traversal

recursive

- dfs(node, level, levelMap) if not root, return
- if level not in levelMap, levelMap[level] = [node.val]
- if level in levelMap, levelMap[level].append(node.val)
- dfs(node.left, level+1,levelMap), dfs(node.right, level+1, levelMap)
- dfs(root, 0, levelMap)
- return [x for x in levelMap.keys()]

iterative

- q with deque([root]), while root
- level with empty list, for loop in range(len(q))
- node = q.popleft, level append node.val, if left q append left, if right, q append right
- result append level

Listing 7: level ordr traversal dp

```
class Solution:
def dfs(self, node, level, levelMap):
    if not node:
        return

    if level not in levelMap:
        levelMap[level] = [node.val]
    else:
        levelMap[level].append(node.val)

    self.dfs(node.left, level + 1, levelMap)
    self.dfs(node.right, level + 1, levelMap)


def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
    levelMap = {}
    self.dfs(root, 0, levelMap)
    return [levelMap[level] for level in sorted(levelMap.keys())]
```

Listing 8: level order traversal bfs

```
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:

        if not root:
            return root

        # bfs
        from collections import deque

        # create queue and res for return
        q = deque()
        q.append(root)
        res = []

        while q:
            level = []
            for _ in range(len(q)):
                node = q.popleft()
                level.append(node.val)

                if node.left:
                    q.append(node.left)

                if node.right:
                    q.append(node.right)
            res.append(level)
        return res
```

Question 1.5: LeetCode 104: Maxium Depth

recursive dfs

- if not root, return 0
- return max(depth(root.left), depth(root.right)) + 1

bfs

- general bfs
- count the level number

iterative dfs

- stack = [[root, 1]]
- while stack, pop the depth and root from stack
- res = max(res, depth), append([node.left, depth+1]), append([node.right, depth+1])
- return res

Listing 9: dfs recursive

```
class Solution:
def maxDepth(self, root: Optional[TreeNode]) -> int:
    if not root:
        return 0

    return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1
```

Listing 10: bfs

```
class Solution:
def maxDepth(self, root: Optional[TreeNode]) -> int:
    from collections import deque
    if not root:
        return 0

    q = deque([root])
    count = 0
    while q:
        for _ in range(len(q)):
            cur = q.popleft()
            if cur.left:
                q.append(cur.left)
            if cur.right:
                q.append(cur.right)

        count += 1

    return count
```

Listing 11: dfs iterative

```
class Solution:
def maxDepth(self, root: Optional[TreeNode]) -> int:
    stack = [[root, 1]]
    res = 0

    while stack:
        node, depth = stack.pop()
        if node:
            res = max(res, depth)
            stack.append([node.left, depth + 1])
            stack.append([node.right, depth + 1])

    return res
```

Question 1.6: LeetCode 101, symmetric tree

recursive

1. if not node1 and not node2 return True
2. if only node1 or node2 exist, return false
3. if node1.val == node2.val, check the node1.left, node2.right and node1.right and node2.left

iterative

- split to left and right node
- bfs root1 left right, node2 right left, and see if the tree are the same

Listing 12: recursion

```
class Solution:
def check_mirror(self, node1, node2):
    if not node1 and not node2:
        return True

    if (node1 and not node2) or (not node1 and node2):
        return False

    if node1.val == node2.val:
        return self.check_mirror(node1.left, node2.right) and self.check_mirror(node1.
            right, node2.left)

def isSymmetric(self, root: Optional[TreeNode]) -> bool:
    return self.check_mirror(root, root)
```

Listing 13: iterative

```
class Solution:
def check_mirror(self, node1, node2):
    if root.left and not root.right:
        return False
    elif root.right and not root.left:
        return False
    elif not root.left and not root.right:
        return True
    leftT,rightT = [root.left.val],[root.right.val]
    q1,q2 = deque(),deque()
    q1.append(root.left)
    q2.append(root.right)
    while q1:
        node = q1.popleft()
        if node.left:
            leftT.append(node.left.val)
            q1.append(node.left)
        if not node.left:
            leftT.append(1000)
        if node.right:
            leftT.append(node.right.val)
            q1.append(node.right)
        if not node.right:
            leftT.append(1000)
    while q2:
        node = q2.popleft()
        if node.right:
            rightT.append(node.right.val)
            q2.append(node.right)
        if not node.right:
            rightT.append(1000)
        if node.left:
            rightT.append(node.left.val)
            q2.append(node.left)
        if not node.left:
            rightT.append(1000)
    return leftT==rightT
```

Question 1.7: LeetCode 114, path sum

recursive

1. if not root, return False
2. targetsum - node.val

3. if the end of leaves, not node.left and node.right check the left and right with targetsum hasPathSum(root.left, targetSum) or hasPathSum(root.right, targetSum)

Listing 14: path sum recursion

```
class Solution:
def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
    if not root:
        return False

    targetSum -= root.val
    if not root.left and not root.right:
        return targetSum == 0

    return self.hasPathSum(root.left, targetSum) or self.hasPathSum(root.right,
        targetSum)
```

Question 1.8: LeetCode 250, count univalue subtree

1. if not node, return True, it is univalue
2. go dfs(node.left) dfs(node.right), if both of them are false, return false
3. if the node.left.val exist and does not equal to node.val, not a univalue subtree, return false
4. if the node.right.val exist and does not equal to node.val, not a univalue subtree, return false
5. otherwise, count += 1, and return True

```
class Solution:
def countUnivalSubtrees(self, root: Optional[TreeNode]) -> int:
    self.count = 0

    def dfs(node):
        if not node:
            return True

        l = dfs(node.left)
        r = dfs(node.right)

        if not l or not r:
            return False

        if node.left and node.val != node.left.val:
            return False

        if node.right and node.val != node.right.val:
            return False


        self.count += 1

        return True

    dfs(root)

    return self.count
```

Question 1.9: LeetCode 106, construct binary tree, inorder and postorder

1. Hashmap for inorder, number: index
2. if l>r return None
3. root = TreeNode(postorder.pop)
4. get index of root for inorder indx = Hashmap[root.val]
5. root.right = helper(idx+1, r)
6. root.left = helper(l, idx-1)
7. return root, return helper(0, len(inorder)-1)

Listing 15: construct binary tree

```
class Solution:
def buildTree(self, inorder: List[int], postorder: List[int]) -> Optional[TreeNode]:
    inorderIndx = {v:i for i, v in enumerate(inorder)}

    def helper(l, r):

        if l > r:
            return None

        root = TreeNode(postorder.pop())
        idx = inorderIndx[root.val]
        root.right = helper(idx+1, r)
        root.left = helper(l, idx-1)

        return root

    return helper(0, len(inorder) - 1)
```

Question 1.10: LeetCode 105, construct binary tree, preorder and inorder

1. HashMap for inorder
2. if l>r, return None
3. root preorder pop(0)
4. find the index of root in inorder
5. root.right = helper(idx+1, r)
6. root.left = helper(l, idx-1)
7. return root, return helper(0, len(inorder) - 1)

```
class Solution:
def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
    inorderIndex = {v: i for i, v in enumerate(inorder)}

    def helper(l, r):
        if l > r:
            return None

        root = TreeNode(preorder.pop(0))
        idx = inorderIndex[root.val]
        root.left = helper(l, idx-1)
```

```
            root.right = helper(idx+1,r)
            return root

        return helper(0, len(preorder) - 1)
```