In [ ]:
```python
import timeit
import random
from typing import List
```

Part 1

This is the review for assignment 1 done with my partner `Eliezer`.

Part 2

1. Paraphrase the problem in your own words.

This question is seraching for the root to leaf paths for a given roots of a binary tree. It needs to return list of lists for whole tree where lists inside the list refers to the paths.

2. Create 1 new example that demonstrates you understand the problem. Trace/walkthrough 1 example that your partner made and explain it.

In [ ]:
```python
'''
Suppose we have the following tree

       5
    /    \
   3      4
 /  \    / \
6    7  8   9

'''

# tree = [5,3,4,6,7,8,9]

# paths = [[5,3,6],[5,3,7],[5,4,8],[5,4,9]]
```

Out[ ]:
```
'\nSuppose we have the following tree\n\n       5\n    /      3      4\n /  \\    / 6    7  8   9
\n\n'
```

3. Copy the solution your partner wrote.

In [ ]:
```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# Create an extra function in order to distribute the root values
def insertLevelOrder(arr, root, i, n):
    # Base case for recursion
    if i < n:
        temp = TreeNode(arr[i])
        root = temp

        # insert left child
        root.left = insertLevelOrder(arr, root.left, 2 * i + 1, n)

        # insert right child
        root.right = insertLevelOrder(arr, root.right, 2 * i + 2, n)
    return root

# Create the function use the started code provided as base
def bt_path(root: TreeNode) -> List[List[int]]:
    if not root:
        return []

    paths = []
    def find_paths(node, current_path):
        if node:
            # Append the current node's value to the path
            current_path.append(node.val)

            # If it's a leaf node, add the path to the paths list
            if not node.left and not node.right:
                paths.append(list(current_path))
            else:
                find_paths(node.left, current_path)
                find_paths(node.right, current_path)

            # Backtrack to explore other paths
            current_path.pop()

    find_paths(root, [])
    return paths

# Insert the root provided in the q2.md file to confirm output
arr = [1, 2, 2, 3, 5, 6, 7]
n = len(arr)

# Configure insertLevelOrder function
root = None
root = insertLevelOrder(arr, root, 0, n)

# Call the bt_path function and print paths
print(bt_path(root))
```

```
[[1, 2, 3], [1, 2, 5], [1, 2, 6], [1, 2, 7]]
```

```
In [ ]: # Check to see if code runs successfully

        # Given another root, which is my own example provided above
        arr = [2, 3, 4, 6, 9]
        n = len(arr)
        root = None
        root = insertLevelOrder(arr, root, 0, n)

        # Call the bt_path function and print paths
        print(bt_path(root))
```

```
[[2, 3, 6], [2, 3, 9], [2, 4]]
```

4. Explain why their solution works in your own words.

Eliezer's solution works great because the `insertLevelOrder` function ensures the binary tree is constructed accurately from the list, following level order. This means the tree structure reflects the input list's structure, assuming the list represents a complete binary tree. In the next step, the `find_paths` function traverses the tree using depth-first search (DFS). Exploring all branches (left and right subtrees) and backtracking after reaching leaf nodes guarantees that all root-to-leaf paths are found and recorded. In the end, Eliezer uses `backtracking` to add and then remove nodes from the current path. This part ensures that the `current_path` list accurately represents the path from the root to the current node at any point in the recursion. This way, when a leaf node is reached, `current_path` contains the path from the root to that leaf.

5. Explain the problem’s time and space complexity in your own words.

The time complexity of Eliezer's solution is `O(n)` because it creates the tree from a level order traversal for forming the `n` number of nodes in the tree. Also, it iterates through the list once to create all nodes and set their left and right children. Then, because of using the `Depth-First Search (DFS)` traversal of the tree, in the worst case, it visits each node exactly once, which gives a time complexity of `O(n)`. Eliezer's solution has a space complexity of `O(n)` because, based on the recursive DFS to find all paths, the maximum depth of the recursion call stack can be equal to the height of the tree, which, in the worst case, can be `O(n)`.

6. Critique your partner's solution, including explanation, if there is anything should be adjusted.

I found Eliezer's solution optimized for tree construction and finding paths. The tree construction method in this solution is straightforward and correctly builds the binary tree from a given array. The algorithm for finding root-to-leaf paths is efficient because it visits each node exactly once, but it could be considered slightly inefficient in terms of space because it creates a new list for every path found.

Potential adjustments for Eliezer's solution can be: Instead of appending the current path to a list of paths at each leaf node, an alternative approach could involve using a global list to store paths and a temporary list to store the current path. This approach would still require copying the current path list when a leaf is reached, but it's a minor optimization that clarifies the intention of storing completed paths separately. While Eliezer's recursive method for constructing the tree is intelligent and straightforward, an iterative queue approach could be more intuitive and align more closely with the level-order traversal concept. The recursive depth-first search approach for finding paths is optimal in terms of ensuring that all paths are explored. However, the recursion could lead to a large call stack in extremely deep trees. An iterative stack approach could be considered, though it would complicate the logic for managing the current path state.

In conclusion, Eliezer's solution is effective and demonstrates a good understanding of binary trees, level-order construction, and depth-first search for pathfinding.

Part 3: Reflection

Undertaking Assignment 1 was a significant leap in my journey of understanding complex problem-solving techniques. This project involved identifying missing numbers in a given range. The first step was to analyze the problem statement in detail. I spent time understanding the solution to the task, which involved identifying missing elements in a specified range of integers. To solve this problem, I revisited fundamental concepts in the lectures and Python, such as list manipulation, loops, and conditional statements, through various online resources and documentation. Developing the solution was iterative and involved testing different strategies. I explored algorithms that could handle the problem's constraints more elegantly, such as utilizing arrays to identify missing numbers for faster processes.

Reviewing my partner's assignment were invaluable, and being familiar with his process and solution helped me gain new perspectives on solving problems related to binary trees. It was informative to see how my partner Eliezer could tackle the problem, underscoring the diversity of problem-solving in programming. This peer review enhanced my ability to evaluate a solution critically. These two assignments were not just about finding missing numbers or evaluating another person's solution; they were comprehensive analytical thinking, problem-solving, and collaborative learning exercises.

Thanks to `Salaar, Tina, and Eliezer` for making this valuable experience possible.

In [ ]:
```

```