```
(hash('Nourin') % 3) + 1
```

```
2
```

```
# Importing necessary libraries
from typing import List

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# Function to solve the problem
def bt_path(root: TreeNode) -> List[List[int]]:
    paths = []  # To store the final paths

    def dfs(node, current_path):
        if not node:
            return

        current_path.append(node.val)  # Add the current node to the
path

        # Check if it's a leaf node
        if not node.left and not node.right:
            paths.append(list(current_path))  # Add a copy of the
current path to the final paths
        else:
            # Recursively traverse the left and right subtrees
            dfs(node.left, current_path)
            dfs(node.right, current_path)

        current_path.pop()  # Backtrack - remove the current node
while going back up

    # Start the DFS traversal
    dfs(root, [])

    return paths

# Example usage:
# Create a binary tree (you can modify this based on your actual tree
structure)
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(5)
```

```
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

# Call the function and print the result
result = bt_path(root)
print(result)
```

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

Explain why your solution works

The provided solution works due to its utilization of Depth-First Search (DFS) to traverse the binary tree. The DFS algorithm explores the tree by moving as far as possible along a branch before backtracking. Here's why the solution is effective:

DFS Traversal:

The algorithm employs a recursive Depth-First Search approach to explore the tree. At each step, it adds the current node to the current path.

Leaf Node Detection:

When a leaf node is encountered (a node without left or right children), the current path is considered a complete path from the root to a leaf. This path is appended to the list of final paths.

Backtracking:

The algorithm uses backtracking to explore other possible paths. After exploring one branch, it backtracks by removing the last added node, allowing exploration of other branches.

This combination of DFS, leaf node detection, and backtracking ensures that the algorithm captures all paths from the root to the leaves.

Explain the problem’s time and space complexity

Time Complexity:

The time complexity is O(N), where N is the number of nodes in the binary tree. The algorithm visits each node exactly once during the DFS traversal.

Space Complexity:

The space complexity is O(H), where H is the height of the binary tree. In the worst case, the recursion stack can go as deep as the height of the tree. Additionally, the paths list stores the results, and in the worst case, it could store all leaf-to-root paths.

Explain the thinking behind an alternative solution

An alternative solution could involve using Breadth-First Search (BFS) to traverse the tree level by level. The main points for an alternative BFS solution would be:

BFS Traversal:

The algorithm explores the tree level by level, visiting nodes at the same depth before moving on to the next level.

Queue for Paths:

Instead of using recursion, a queue could be employed to maintain the paths. At each level, the paths along with their respective nodes would be enqueued.

Leaf Node Detection:

Similar to the DFS approach, paths that reach leaf nodes would be considered complete and added to the final list of paths.

An implementation of this approach would involve initializing a queue, enqueuing the root along with an initial empty path, and then iteratively dequeuing and enqueueing nodes and paths until all paths are explored.

This alternative approach provides a different time and space complexity trade-off compared to DFS, and it could be beneficial in scenarios where the tree is wide rather than deep.