```
!pip install tree
!pip install binary_tree
```

```
Requirement already satisfied: tree in /usr/local/lib/python3.10/dist-
packages (0.2.4)
Requirement already satisfied: Pillow in
/usr/local/lib/python3.10/dist-packages (from tree) (9.4.0)
Requirement already satisfied: svgwrite in
/usr/local/lib/python3.10/dist-packages (from tree) (1.4.3)
Requirement already satisfied: setuptools in
/usr/local/lib/python3.10/dist-packages (from tree) (67.7.2)
Requirement already satisfied: click in
/usr/local/lib/python3.10/dist-packages (from tree) (8.1.7)
Requirement already satisfied: binary_tree in
/usr/local/lib/python3.10/dist-packages (0.0.1)
```

```python
import binary_tree
from typing import List
```

```python
first_name = 'Fatima'
result = (hash(first_name) % 3) + 1
```

```python
print(result)
```

```
2
```

```python
# The task is to devise a Python function that, given the root of a
binary tree, produces all the possible paths from the root to the
leaves of the tree.
#The order of these paths in the output is flexible.
```

```python
# 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

def bt_path(root: TreeNode) -> List[List[int]]:
    paths = []  # To store the result

    def dfs(node, current_path):
        if not node:
            return
        current_path.append(node.val)

        if not node.left and not node.right:  # Leaf node
            paths.append(list(current_path))
        else:
            dfs(node.left, current_path)
            dfs(node.right, current_path)
        current_path.pop()  # Backtrack

    dfs(root, [])
    return paths

# Example 1
#     1
#    / \
#   2   3
#  / \
# 4   5

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

result = bt_path(root)
print(result)
```

```
[[1, 2, 4], [1, 2, 5], [1, 3]]
```

```
# Example 2
#     8
#    / \
#   3   10
#  / \    \
# 1   6    14
#    / \   /
#   4   7 13

root = TreeNode(8)
root.left = TreeNode(3)
root.right = TreeNode(10)
root.left.left = TreeNode(1)
root.left.right = TreeNode(6)
root.left.right.left = TreeNode(4)
root.left.right.right = TreeNode(7)
root.right.right = TreeNode(14)
root.right.right.left = TreeNode(13)

result = bt_path(root)
print(result)
```

```
[[8, 3, 1], [8, 3, 6, 4], [8, 3, 6, 7], [8, 10, 14, 13]]
```

Explanation of Why the Solution Works:

The solution works because it employs a depth-first search (DFS) approach to traverse the binary tree, maintaining the current path from the root to the current node. When a leaf node is reached, the current path is appended to the list of paths. This process continues recursively until all paths from the root to the leaves are explored. The depth-first search ensures that all possible paths are considered, and the backtracking step (current_path.pop()) ensures that the algorithm explores all paths without duplications.

Explanation of Time and Space Complexity:

Time Complexity: The time complexity is O(N), where N is the number of nodes in the binary tree. In the worst case, the algorithm needs to visit every node to explore all possible root-to-leaf paths.

Space Complexity: The space complexity is O(H), where H is the height of the binary tree. This is because the algorithm uses recursion, and the maximum depth of the recursion stack is determined by the height of the tree. In the worst case, for a skewed tree, the height is equal to the number of nodes, resulting in O(N) space complexity. However, for a balanced tree, the height is log(N), leading to better space efficiency.

Alternative Solution Proposal:

Instead of going deep into the tree (DFS), we can take a broad approach (BFS) by exploring level by level. Start with the root, and for each level, record the path from the root to each node. We use a queue to keep track of nodes and paths. As we process each node, we update and enqueue its children along with their paths.

Steps:

1. Begin with an empty queue.
2. Enqueue the root node and an empty path list.
3. While there are nodes in the queue:

- Dequeue a node and its associated path.
- If the node is a leaf, add the path to the list of paths.
- Enqueue the left child with an updated path.
- Enqueue the right child with an updated path.

This straightforward approach ensures we cover all root-to-leaf paths. It's handy when the tree is wide, providing a balanced exploration.

Time Complexity: O(N) - Visiting each node once. Space Complexity: O(N) - Considering the queue size.

The simplicity lies in its breadth-first strategy, making it easy to implement with basic queue operations and no explicit backtracking.