**Time Complexity**

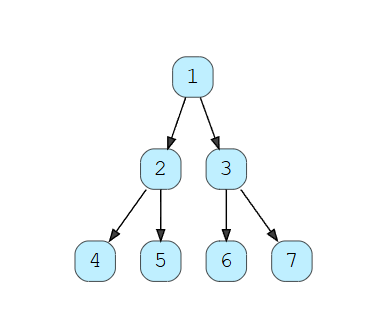
The time complexity of the above algorithm is O(N2), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once (which will take O(N)), and for every leaf node, we might have to store its path (by making a copy of the current path) which will take O(N).

We can calculate a tighter time complexity of O(N logN) from the space complexity discussion below.

**Space Complexity**

If we ignore the space required for the allPaths list, the space complexity of the above algorithm will be O(N) in the worst case. This space will be used to store the recursion stack. The worst-case will happen when the given tree is a linked list (i.e., every node has only one child).

How can we estimate the space used for the allPaths array? Take the example of the following balanced tree:



Here we have seven nodes (i.e., N = 7). Since, for binary trees, there exists only one path to reach any leaf node, we can easily say that total root-to-leaf paths in a binary tree can’t be more than the number of leaves. As we know that there can’t be more than  leaves in a binary tree, therefore the maximum number of elements in allPaths will be O((N+1)/2)) = O(N) . Now, each of these paths can have many nodes in them. For a balanced binary tree (like above), each leaf node will be at maximum depth. As we know that the depth (or height) of a balanced binary tree is O(logN) we can say that, at the most, each path can have logN nodes in it. This means that the total size of the allPaths list will be O(NlogN). If the tree is not balanced, we will still have the same worst-case space complexity.

From the above discussion, we can conclude that our algorithm’s overall space complexity is O(N LogN).

Also, from the above discussion, since for each leaf node, in the worst case, we have to copy log(N) nodes to store its path; therefore, the time complexity of our algorithm will also be O(N LogN).

