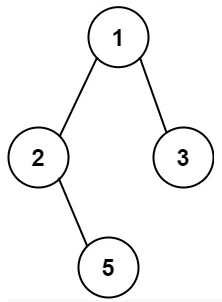
<https://leetcode.com/problems/binary-tree-paths/>

Given the root of a binary tree, return *all root-to-leaf paths in* ***any order***.

A **leaf** is a node with no children.

**Example 1:**



Input: root = [1,2,3,null,5]

Output: ["1->2->5","1->3"]

**Example 2:**

Input: root = [1]

Output: ["1"]

**Constraints:**

The number of nodes in the tree is in the range [1, 100].

-100 <= Node.val <= 100

**Attempt 1: 2022-10-29**

**Solution 1:  Recursive traversal with String (10min)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\*    int val;

\*    TreeNode left;

\*    TreeNode right;

\*    TreeNode() {}

\*    TreeNode(int val) { this.val = val; }

\*    TreeNode(int val, TreeNode left, TreeNode right) {

\*        this.val = val;

\*        this.left = left;

\*        this.right = right;

\*    }

\* }

\*/

class Solution {

    public List<String> binaryTreePaths(TreeNode root) {

        List<String> result = new ArrayList<String>();

        helper(result, root, "");

        return result;

    }

    private void helper(List<String> result, TreeNode root, String path) {

        if(root == null) {

            return;

        }

        // No matter if current node is leaf node or internal node, we need

        // to record its value into 'path', so 'path' not only update inside

        // leaf node base case, it always update during all recursion, and

        // when encounter leaf node we add full build one 'path' into result

        // Update 'path' two scenarios:

        // 1.Initial node on the path, no need to add "->" before node value

        // 2.Other nodes on the path, add "->" before node value

        path = path.length() == 0 ? path + root.val : path + "->" + root.val;

        // Base case: leaf node

        if(root.left == null && root.right == null) {

            result.add(path);

            return;

        }

        helper(result, root.left, path);

        helper(result, root.right, path);

    }

}

Time Complexity: O(nlogn) ~ O(n^2) -> best ~ worst case explain below

Space Complexity: O(logn) ~ O(n) -> based on best ~ worst case of tree height

**Solution 2:  Recursive traversal with StringBuilder and Backtracking (10min)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\*    int val;

\*    TreeNode left;

\*    TreeNode right;

\*    TreeNode() {}

\*    TreeNode(int val) { this.val = val; }

\*    TreeNode(int val, TreeNode left, TreeNode right) {

\*        this.val = val;

\*        this.left = left;

\*        this.right = right;

\*    }

\* }

\*/

class Solution {

    public List<String> binaryTreePaths(TreeNode root) {

        List<String> result = new ArrayList<String>();

        StringBuilder sb = new StringBuilder();

        helper(result, root, sb);

        return result;

    }

    private void helper(List<String> result, TreeNode root, StringBuilder sb) {

        if(root == null) {

            return;

        }

        // No matter if current node is leaf node or internal node, we need

        // to record its value into 'path', so 'path' not only update inside

        // leaf node base case, it always update during all recursion, and

        // when encounter leaf node we add full build one 'path' into result

        // Update 'path' two scenarios:

        // 1.Initial node on the path, no need to add "->" before node value

        // 2.Other nodes on the path, add "->" before node value

        if(sb.length() == 0) {

            sb.append(root.val);

        } else {

            sb.append("->").append(root.val);

        }

        // Base case: leaf node

        if(root.left == null && root.right == null) {

            result.add(sb.toString());

            return;

        }

        // Why needs backtrack logic on StringBuilder ? But no need on String ?

        // StringBuilder is an object which will maintain a change on the String,

        // if no backtrack implement, then changes on current String will pass

        // into next recursion, for String is different, even String is object but

        // behavior similar to primitive type since immutable, when String pass

        // into next recursion, the changes on current String won't inherit, but

        // for StringBuilder object we have to do backtrack

        //

        // How to backtrack for StringBuilder in classic traversal recursion ?

        // Record current StringBuilder object length right before step into

        // next level recursion by "len = sb.length()", and after next level

        // recursion done immediately remove appending String section during next

        // level recursion by "sb.setLength(len)"

        int len = sb.length();

        helper(result, root.left, sb);

        sb.setLength(len);

        helper(result, root.right, sb);

        sb.setLength(len);

    }

}

Time Complexity: O(nlogn) ~ O(n^2) -> best ~ worst case explain below

Space Complexity: O(logn) ~ O(n) -> based on best ~ worst case of tree height

**1. Why needs backtrack logic on StringBuilder ? But no need on String ?**

<https://leetcode.com/problems/binary-tree-paths/discuss/68258/Accepted-Java-simple-solution-in-8-lines/70169>

**"StringBuilder" is a mutable object, it will hold its value after returning. Whereas String creates a copy in every recursion, you don't need to worry about the "side-effect" when backtrack.**

**StringBuilder is an object which will maintain a change on the String, if no backtrack implement, then changes on current String will pass into next recursion, for String is different, even String is object but behavior similar to primitive type since immutable, when String pass into next recursion, the changes on current String won't inherit, but for StringBuilder object we have to do backtrack**

**2. How to backtrack for StringBuilder in classic traversal recursion ?**

**Record current StringBuilder object length right before step into next level recursion by "len = sb.length()", and after next level recursion done immediately remove appending String section during next level recursion by "sb.setLength(len)"**

**3. StringBuilder vs. String in recursion which is better ?**

**Refer to**

<https://leetcode.com/problems/binary-tree-paths/discuss/68258/Accepted-Java-simple-solution-in-8-lines/70139>

**Time Complexity Analysis**

**Case Study**

First let's work on the **balanced** tree situation:

It should be obvious to see now that each node will contribute to the total time cost an amount of length of the path from the root to this node. The problem is to see how to sum up these paths' lengths for N nodes altogether.

Denote the time complexity for N nodes as T(N).

Suppose we do have that balanced tree now (and also N is 2^N-1 for simplicity of discussion). And we know that N/2 nodes lie at the leaf/deepest level of the BST since it's balanced binary tree.

We easily have this **recurrence formula**:

T(N) = T(N/2) + (N/2) \* lgN

Which means, we have N nodes, with half lying on the deepest (the lgNth) level. The sum of path lengths for N nodes equals to sum of path lengths for all nodes except those on the lgN-th level plus the sum of path lengths for those nodes on the lgN-th level.

This recurrence is not hard to solve. I did not try to work out the exact solution since the discussion above in itself are in essence a little blurry on corner cases, but it is easy to discover that

T(N) = O(NlgN).

**To Generalize: Let's Start with Worst-Case**

The problem left here now, is a balanced tree the **best-case** or the **worst-case**? I was convinced it was the worst case before I doodled some tree up and found otherwise.

The **worst case** is actually when all nodes lie up to a single line like a linked list, and the complexity in this case is easily calculable as

O(N^2). But how do we prove that?

The proof is easier than you think. Just use induction. Suppose we have N - 1 nodes in a line, and by inductive hypothesis we claim that this tree is the max-path-sum tree for N-1 nodes. We just have to prove that the max-path-sum tree for N nodes is also a single line. How do you prove it? Well suppose that you see the N-1-node line here, and you want to add the N-th node, where would you put it? Of course the deepest level so that the new node gets maximum depth.

**Best-Case**

Proving that the **best case** is the balanced tree can be a little trickier. By [some definition](http://www.geeksforgeeks.org/how-to-determine-if-a-binary-tree-is-balanced/), A tree where no leaf is much farther away from the root than any other leaf. Suppose we define **much farther** as like 2 steps farther. Then for the purpose of contradiction, suppose the

min-path-sum tree for N nodes is not balanced, then there is a leaf A that is at least 2 steps further to the root than another leaf B. Then we can always move A to be the direct descendant of B (since B is leaf, there is an opening) resulting in a tree with smaller sum of paths. Thus the contradiction. This proof is a little informal, but I hope the idea is clear.

**Conclusion: Upper and Lower Bounds**

In conclusion, the complexity of this program is Ω(NlgN) ~ O(N^2).

**Optimization: How About Mutability**

I do think that the main cost incurred in this algorithm is a result of the immutability of String and the consequent allocation and copying. Intuitively, it seems like that StringBuilder could help. Using StringBuilder, we could make sure that String allocation and copying only happens when we are at a leaf. That means, the cost would be now sum of length of all root-to-leaf paths rather than sum of length of all root-to-node paths. This is assuming that StringBuilder.append and StringBuilder.setLength can work in O(1) or at least less than O(N) time.

This is my original version using String:

public class Solution {

    public List binaryTreePaths(TreeNode root) {

        List res = new ArrayList<>();

        if (root == null) return res;

        dfs(root, res, "");

        return res;

    }

    private void dfs(TreeNode root, List ls, String accum) {

        if (root == null) return;

        accum += (accum.length() == 0 ? "" : "->") + root.val;

        if (root.left == null && root.right == null) {

            ls.add(accum);

            return;

        }

        dfs(root.left, ls, accum);

        dfs(root.right, ls, accum);

    }

}

And two submissions reported 15ms and 17ms.

And I implemented another AC version with StringBuilder:

public class Solution {

    public List binaryTreePaths(TreeNode root) {

        List res = new ArrayList<>();

        StringBuilder sb = new StringBuilder();

        if (root == null) return res;

        dfs(root, res, sb);

        return res;

    }

    private void dfs(TreeNode root, List ls, StringBuilder accum) {

        if (root == null) return;

        accum.append((accum.length() == 0 ? "" : "->") + root.val);

        int len = accum.length();

        if (root.left == null && root.right == null) {

            ls.add(accum.toString());

            return;

        }

        dfs(root.left, ls, accum);

        accum.setLength(len);

        dfs(root.right, ls, accum);

        accum.setLength(len);

    }

}

Two submissions reported 18ms and 19ms.So the performance did not show significant improvement in the perspective of the OJ and the set of test cases we have. But I do think theoretically StringBuilder could help. Also, the best-case and worst-case analysis would change in the case of using StringBuilder.

**There is one more backtracking solution but implement by List<String>**

**Refer to**

<https://algo.monster/liteproblems/257>

**Problem Description**

The problem presents us with a binary tree, where each node contains an integer value and can have a left and/or right child. Our goal is to find all the unique paths from the root of the binary tree to its leaves. A leaf is defined as a node that has no children, which means neither a left nor a right child. The paths should be represented as strings, with each node's value on the path concatenated by "->". For instance, if the path goes through nodes with values 1, 2, and 3 in that order, the path string should be "1->2->3". We are required to return these path strings in any order.

**Intuition**

To solve this problem, we can utilize a technique called Depth-First Search (DFS). This strategy explores as far as possible along each branch before backtracking. Here's how we arrive at the solution approach:

We can start at the root and perform a DFS traversal on the binary tree.

**As we traverse, we keep track of the current path by noting the nodes visited so far in the sequence.**

**Whenever we reach a leaf node (a node without children), we record the path we've taken to get there. This is a complete root-to-leaf path, so we add this to our list of answers.**

**The key part of this process is backtracking. When we visit a node and explore all of its children, we backtrack, which means we remove the node from our current path and return to the previous node to explore other paths.**

We continue this process until all paths are explored and we have visited all the leaves.

The recursive function dfs() in the solution code is where the DFS takes place. It takes the current node as a parameter, and as the function is called recursively, a local variable t keeps track of the current path as a list of node values. If a leaf node is reached, the current path is converted to the path string format required and added to the list ans, which contains all the full path strings. After exploring a node's left and right children, the node's value is popped from the path list to allow backtracking to the previous node's state.

**Solution Approach**

The solution approach for the given problem utilizes a typical pattern for tree traversal problems, which is the Depth-First Search (DFS) algorithm. Here's a step-by-step explanation of how the solution is implemented:

**Define a Recursive Function (dfs):** The recursive function dfs is defined within the class Solution. It is invoked with the current node being visited. This function does not return any value but instead updates the ans list with the path strings.

**Base Case for Recursion:** In the function dfs, before going further into recursion, we check if the given root node is None, indicating that we've reached beyond a leaf node. In this case, the function simply returns without performing any further action.

**Track the Current Path: A local variable t in the class scope is used to keep track of the current path. It's a stack (implemented using a list in Python) that we update as we go down the tree. For each node, we convert its value to a string and append it to t.**

**Check for Leaf Node:** In the DFS process, if the current node is a leaf (both left and right child nodes are None), we convert the current path into the required string format, which is obtained by joining the sequence of node values in t with "->". This complete path string is added to the ans list.

**Recursive DFS Calls:** If the current node has children, we perform DFS for both the left child (if not None) and right child (if not None). The function calls itself with root.left and root.right correspondingly.

**Backtracking: After exploring both children from the current node, we need to backtrack. This ensures that when we return from the recursive call, the current path t reflects the state as if the current node was never visited. We achieve this by popping the last value from the t.**

**Return Paths:** Once the DFS is complete, the ans list will be populated with all the path strings from root to all the leaf nodes. This list is returned as the final result of the binaryTreePaths method.

The use of a stack for tracking the paths and the pattern of adding to the list only at leaf nodes are critical parts of the algorithm. The recursive DFS makes sure all potential paths are explored, while backtracking ensures that every unique path is properly captured without duplication.

**Example Walkthrough**

Let's illustrate the solution approach using a small example binary tree:

Consider the following binary tree:

1

/ \

2 3

/ \

5 4

Here's how we would apply the solution approach:

We start with the root node which is 1.

We initialize t with the value of the current node: t = ["1"].

The node 1 is not a leaf, so we continue. We have two recursive DFS calls to make because both left and right children exist: root.left (which is 2) and root.right (which is 3).

First DFS call: We move to the left child which is 2 and update t to ["1", "2"].

Node 2 has a left child but no right child. We make a recursive DFS call to the left child which is 5.

On reaching the node 5, we update t to ["1", "2", "5"].

Node 5 is a leaf node, so we join the elements of t with "->" to form the path string "1->2->5". We add this to our ans list.

Backtracking: We finished visiting 5, so we pop it from t making t = ["1", "2"].

There are no more children for 2 to visit, so we pop it from t as well. Now t = ["1"].

Second DFS call: Now, we explore the right child of 1, which is 3. We update t to ["1", "3"].

Node 3 has a right child 4 but no left child. We make a recursive DFS call to the right child which is 4.

On reaching 4, we update t to ["1", "3", "4"].

Node 4 is a leaf node, so we join the elements of t with "->" to form the path string "1->3->4". We add this to our ans list.

Backtracking: We've visited 4, so we pop it from t, reverting t back to ["1", "3"].

After exploring node 3's right child, we backtrack one more time, popping 3 from t and t is now back to ["1"].

Finally, since all paths have been explored, our ans list contains the root-to-leaf path strings: ["1->2->5", "1->3->4"]. We'd return this list as the output of our function.

**Solution Implementation**

/\*\*

\* Definition for a binary tree node.

\*/

public class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

class Solution {

// A list to store all path strings

private List<String> allPaths = new ArrayList<>();

// A temporary list to keep the current path nodes

private List<String> currentPath = new ArrayList<>();

/\*\*

\* Finds all paths from root to leaf in a binary tree.

\*

\* @param root The root of the binary tree.

\* @return A list of all root-to-leaf paths in string format.

\*/

public List<String> binaryTreePaths(TreeNode root) {

// Perform a depth-first search starting from the root

depthFirstSearch(root);

return allPaths;

}

/\*\*

\* Performs a recursive depth-first search to find all paths.

\*

\* @param node The current node in the binary tree.

\*/

private void depthFirstSearch(TreeNode node) {

if (node == null) {

// Base case: if node is null, do nothing

return;

}

// Append current node's value to the path

currentPath.add(String.valueOf(node.val));

// If node is a leaf, add the path to the list of all paths

if (node.left == null && node.right == null) {

allPaths.add(String.join("->", currentPath));

} else {

// Recur for left and right children

depthFirstSearch(node.left);

depthFirstSearch(node.right);

}

// Backtrack: remove the last node from the path before returning

currentPath.remove(currentPath.size() - 1);

}

}

**Time and Space Complexity**

The given Python function binaryTreePaths traverses a binary tree to find all root-to-leaf paths. The primary operation is a Depth-First Search (DFS) defined in a nested function. Here's the analysis of its complexities:

**Time Complexity**

The time complexity is O(N), where N is the number of nodes in the tree. This is because each node in the binary tree is visited exactly once during the depth-first search. Therefore, the function's time complexity is directly proportional to the number of nodes.

**Space Complexity**

The space complexity of the function is also O(N). The major factors contributing to the space complexity are:

The recursive call stack of the DFS, which in the worst case could be O(N) when the binary tree degrades to a linked list.

The list t that holds the current path. In the worst case, when the binary tree is completely skewed (e.g., each parent has only one child), this list can also take up to O(N) space.

The list ans that contains all the paths. In the best case, where each node has two children, there will be N/2 leaf nodes resulting in N/2 paths. Although each path could be of varying length, the concatenation of all paths will still take O(N) space since each node's value is only included in one path string.

Considering the stack space for recursive calls and the space for storing the paths, the space complexity in the worst case scenario will be linear with respect to the number of nodes in the tree.

**Solution 3:  Divide and Conquer (30min)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\*    int val;

\*    TreeNode left;

\*    TreeNode right;

\*    TreeNode() {}

\*    TreeNode(int val) { this.val = val; }

\*    TreeNode(int val, TreeNode left, TreeNode right) {

\*        this.val = val;

\*        this.left = left;

\*        this.right = right;

\*    }

\* }

\*/

class Solution {

    public List<String> binaryTreePaths(TreeNode root) {

        List<String> result = new ArrayList<String>();

        // Base case: the end of one path

        if(root == null) {

            return result;

        }

        // Base case: leaf node

        if(root.left == null && root.right == null) {

            result.add("" + root.val);

        }

        // Divide

        List<String> leftResults = binaryTreePaths(root.left);

        List<String> rightResults = binaryTreePaths(root.right);

        // Conquer

        // The order is root.val before current results(path),

        // because for Divide and Conquer will first go to bottom

        // as leaf node and add it onto path, then assemble from

        // bottom up, e.g for below tree, the process is

/\*\*

\* 1

\* / \

\* 2 3

\* / \ / \

\* 4 5 6 7

\*/

// Following "Divide and Conquer", we will go 'Divide' logic

// till 4 and 5 as leaf node first, and now we have already

// opened 3 levels recursion, 1st level root = 1, 2nd level

// root = 2, 3rd level root = 4 or 5, and based on leaf node

// process step, result will add it as 'result.add("" + 4)'

// or 'result.add("" + 5)', and we continue proceed till next

// level as 4th level, but now the nodes on 4th level are

// actually empty node for both 4 and 5, it will return immediately

// to level 3, then continue to below "Conquer" logic on

// 'leftResults' and 'rightResults' which currently stored 4

// inside 'leftResults' and 5 inside 'rightResults', after "Conquer"

// here, it return result and back to level 2, where the root is 2,

// then with below "Conquer" logic again, it put root value ahead

// and build as "2->4" and "2->5"... etc.

        for(String leftResult : leftResults) {

            result.add(root.val + "->" + leftResult);

        }

        for(String rightResult : rightResults) {

            result.add(root.val + "->" + rightResult);

        }

        return result;

    }

}

**Test**

import java.util.ArrayList;

import java.util.List;

public class TreeSolution {

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

public List<String> binaryTreePaths(TreeNode root) {

List<String> result = new ArrayList<String>();

// Base case: the end of one path

if(root == null) {

return result;

}

// Base case: leaf node

if(root.left == null && root.right == null) {

result.add("" + root.val);

}

// Divide

List<String> leftResults = binaryTreePaths(root.left);

List<String> rightResults = binaryTreePaths(root.right);

// Conquer

// The order is root.val before current results(path),

// because for Divide and Conquer will first go to bottom

// as leaf node and add it onto path, then assemble from

// bottom up

for(String leftResult : leftResults) {

result.add(root.val + "->" + leftResult);

}

for(String rightResult : rightResults) {

result.add(root.val + "->" + rightResult);

}

return result;

}

public static void main(String[] args) {

/\*\*

\* 1

\* / \

\* 2 3

\* / \ / \

\* 4 5 6 7

\*/

TreeSolution so = new TreeSolution();

TreeNode one = so.new TreeNode(1);

TreeNode two = so.new TreeNode(2);

TreeNode three = so.new TreeNode(3);

TreeNode four = so.new TreeNode(4);

TreeNode five = so.new TreeNode(5);

TreeNode six = so.new TreeNode(6);

TreeNode seven = so.new TreeNode(7);

one.left = two;

one.right = three;

two.left = four;

two.right = five;

three.left = six;

three.right = seven;

List<String> result = so.binaryTreePaths(one);

System.out.println(result);

}

}

**Refer to**

<https://leetcode.com/problems/binary-tree-paths/discuss/287419/Java-O(N)-Divide-and-Conquer-Solution-100-Beat>

public List<String> binaryTreePaths(TreeNode root) {

        List<String> ans = new ArrayList<>();

        if(root == null){

            return ans;

        }

        // check if only root

        if(root.left == null && root.right == null){

            ans.add("" + root.val);

        }

        // divide left and right

        List<String> leftpaths = binaryTreePaths(root.left);

        List<String> rightpaths = binaryTreePaths(root.right);

        //merge two paths

        //add root val

        for(String str : leftpaths){

            ans.add(root.val + "->" + str);

        }

        for(String str : rightpaths){

            ans.add(root.val + "->" + str);

        }

        return ans;

    }

**Refer to**

[L1430.Check If a String Is a Valid Sequence from Root to Leaves Path in a B](note://WEBf107c8848f6f7fe875b1ed72bf156a44)

[L549.Binary Tree Longest Consecutive Sequence II (Ref.L257,L298)](note://857146268EE34C6EAC3D963EBC1C1199)

[L124.P9.7.Binary Tree Maximum Path Sum (Ref.L543)](note://8907B88FF6D14ADCB5A4E284B0CB00D7)