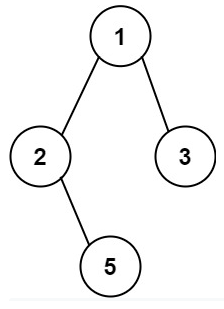
<https://leetcode.com/problems/binary-tree-paths/>

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 left 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 left 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.

**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

for(String leftResult : leftResults) {

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

}

for(String rightResult : rightResults) {

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

}

return 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;

}