



Courses

Practice

Roadmap

Pro



# Algorithms and Data Structures for Beginners

22 / 35

## 22 - Tree Maze

13:58



Mark Lesson Complete











View Code

Prev

Next

## Backtracking

## Suggested Problems

Status	Star	Problem 	Difficulty 	Video Solution	Code
<input type="checkbox"/>		Path Sum	Easy		
<input type="checkbox"/>		Subsets	Medium		
<input type="checkbox"/>		Combination Sum	Medium		

## Backtracking

Backtracking is an algorithm that's similar to DFS on binary trees, which we have already discussed. It operates on a brute-force approach. Imagine that we had to search for "all possible combinations of a pattern lock". We would have to do an exhaustive search to find all possible combinations i.e. there really is not a better algorithm to get all combinations than to search for all of the combinations one by one. This is the idea behind backtracking. We explore a possible way to perform a

task and if we do not succeed, we backtrack and explore other ways until we find a solution.

## Motivation with Example

Having discussed briefly what backtracking is and taking what we know about DFS into consideration, let's see how we would go about solving the following question.

*Q: Determine if a path exists from the root of the tree to a leaf node. It may not contain any zeroes.*

The problem is basically asking us if we can traverse from the root node to the leaf node without having a value of `0`. We return true if there exists a path and false if there does not.

The first thing that comes to mind is using depth-first search. Our constraint is that we cannot have a node with value `0` in our path. We also know that if the tree is empty, then there cannot exist a valid path either. Finally, if we reach a leaf node and have not returned false, we can return true since it means there is a path that exists from root to leaf.

For the sake of this problem, let's assume that there exists exactly one path, so it must exist either in the right-subtree or the left-subtree. Arbitrarily, we choose to try

the left side before right. If the answer was not found in the left-subtree, the algorithm will search in the right-subtree and if the path exists, it will return true.

Given the tree, `[4,0,1,null,7,2,0]` , the valid path would look like the following, as shown in the visual. A path is invalid if it has a `0` in it.

Now that we know our base cases, translating this into code is simple.

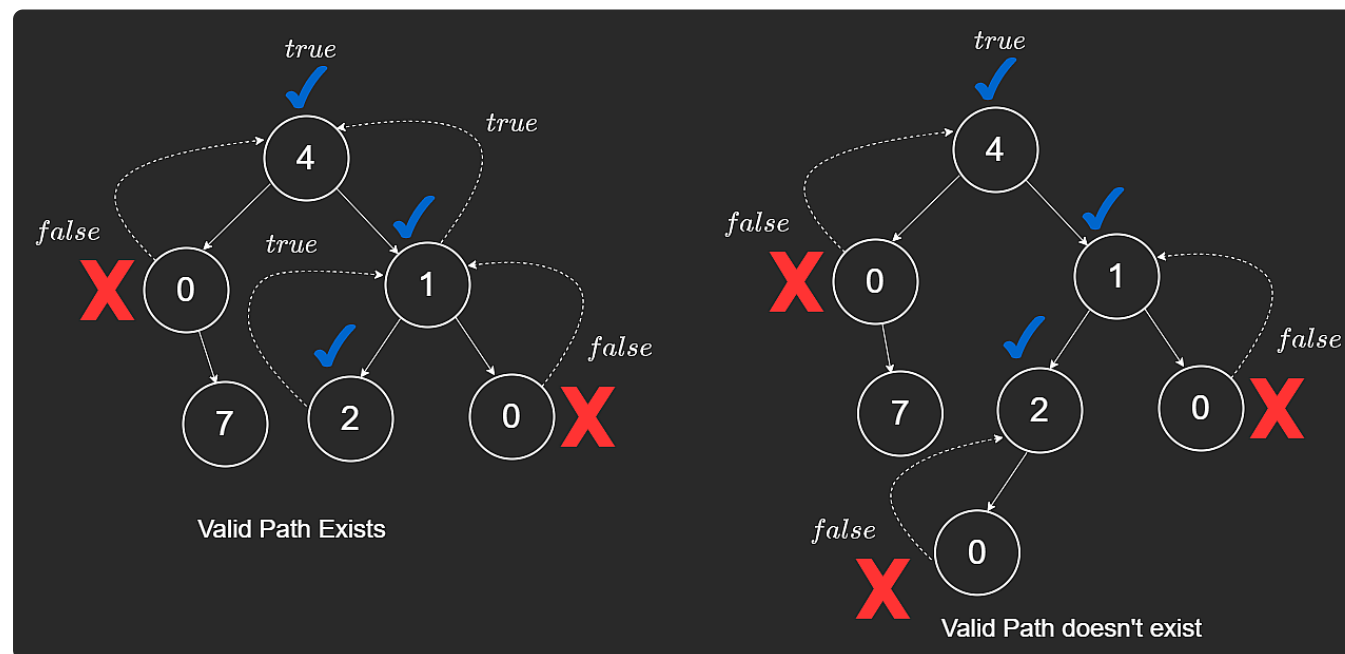
```
class TreeNode:
    constructor(val, left, right):
        val = val
        left = left
        right = right

fn canReachLeaf(root):
    // Immediately return false if null node or node's value is 0
    if root is null or root.val == 0:
        return false
    // If we have reached a valid path
    if root.left is null and root.right is null:
        return true
    // Recursively explore the left subtree
    if canReachLeaf(root.left):
        return true
    // Recursively explore the right subtree
    if canReachLeaf(root.right):
```

```

return true
return false

```



Let's take a look at a slight variation of the question where we have to return the values of the path instead of simply returning a boolean.

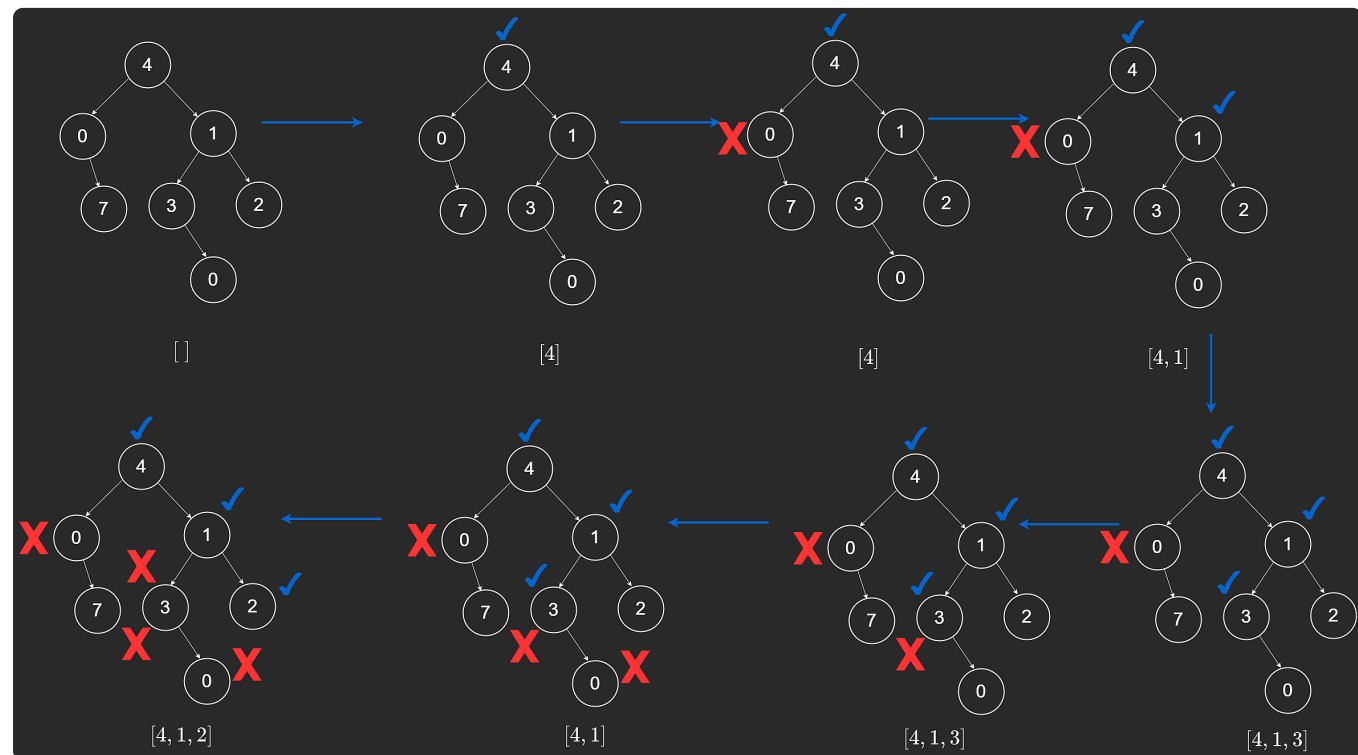
In this problem, we can pass a parameter `path`, which is a list to store all the nodes that are in the valid path. So, given the tree

`[4,0,1,null,7,3,2,null,null,null,0]`, we first add the root node to our list.

Since there is only one valid path, it will either be in the left-subtree or the right-subtree. Prioritizing left over right, the left-subtree is invalid because 4's left child is 0. We return false and now recursively check the right-subtree. Going to the right,

1 is valid, so we add it to our list. Now, we check 3, which is valid, so it gets added to our list. 3's left child is null, so we return false. Checking 3's right child, we again hit the base case. Now, we must remove 3 from our stack because if there existed a valid path, we would have returned true already. We go back up to the 3's parent, which is 1, and check its right subtree. We add 2 to our list. We then explore 2 but 2 is a leaf node, which makes the recursive call return true, after which the function returns true. Our valid path is [4, 1, 2].

The visual below demonstrates this process.



```
def leafPath(root, path):  
    if not root or root.val == 0:  
        return False  
    path.append(root.val)  
  
    if not root.left and not root.right:  
        return True  
    if leafPath(root.left, path):  
        return True  
    if leafPath(root.right, path):  
        return True  
    path.pop()  
    return False
```

## Time Complexity

Given that the tree has  $n$  nodes, the time complexity will be  $O(n)$  because we have to traverse the whole tree. Just like any brute force algorithm, we will have to traverse the whole input, which in this case is just the size of the tree.

## Closing Notes

Backtracking is an abstract algorithm and binary trees are not the only data structure that it can be applied to. We shall see later in the course how we can apply this algorithm to other data structures.



