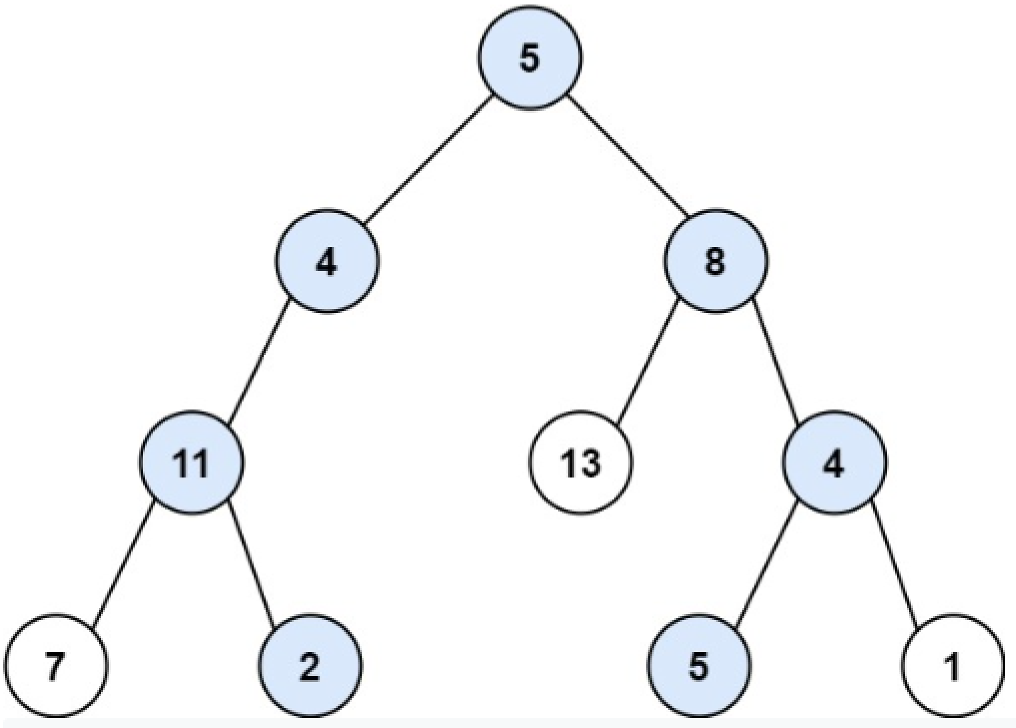
https://leetcode.com/problems/path-sum-ii/

Given the root of a binary tree and an integer targetSum, return *all **root-to-leaf** paths where the sum of the node values in the path equals*
targetSum*. Each path should be returned as a list of the node **values**, not node references*.

A **root-to-leaf** path is a path starting from the root and ending at any leaf node. A **leaf** is a node with no children.

Example 1:



Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22

Output: [[5,4,11,2],[5,8,4,5]]

Explanation: There are two paths whose sum equals targetSum:

5 + 4 + 11 + 2 = 22

5 + 8 + 4 + 5 = 22

Example 2:

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

Output: []

Example 3:

Input: root = [1,2], targetSum = 0

Output: []

Constraints:

- The number of nodes in the tree is in the range [0, 5000].
- -1000 <= Node.val <= 1000
- -1000 <= targetSum <= 1000

Attempt 1: 2022-11-04

Solution 1: Recursive traversal with Deep Copy on passed in ArrayList to find and store paths first and calculate target sum, fully based on L257.Binary Tree Paths (10min)

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        helper(root, result, sum, new ArrayList<Integer>());
        return result;
    }

    private void helper(TreeNode root, List<List<Integer>> result, int sum, List<Integer> list) {
        if(root == null) {
            return;
        }
        List<Integer> tmp = new ArrayList<Integer>(list);
        tmp.add(root.val);
        if(root.left == null && root.right == null) {
            if(sum == root.val) {
                result.add(new ArrayList<Integer>(tmp));
            }
        }
```

```
        helper(root.left, result, sum - root.val, tmp);
        helper(root.right, result, sum - root.val, tmp);
        // How we remove the last element on 'tmp' list without explicit backtrack ?
        // Because DFS naturally a type of backtrack but implicit only backtrack
        // when pass over the leaf nodes during a tree traversal, such as example
        // here, after pass over the leaf nodes, it will encounter 'null' and return
        // to previous recursion level which also "auto remove" the last element like a
        // backtrack but implicitly, and to explain the "auto remove" is because the
        // input parameter as 'list' in each recursion level never changed, the changed
        // object is not 'list' but a deep copy of this 'list' as 'tmp', the impact
        // range of 'tmp' is limited in current recursion level, so when return to
        // previous recursion level, the 'tmp' will gone, the only remain we will find
        // is our unchanged 'list' object
    }
}

Time Complexity: O(n^2), where n is number of nodes in the Binary Tree
Space Complexity: O(n)
```

Solution 2: Recursive traversal without Deep Copy on passed in ArrayList but use Backtracking to find and store paths first and calculate target sum, fully based on L257.Binary Tree Paths (10min)

Style 1: 2ms beats 85.58%

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
```

```
        helper(root, result, sum, new ArrayList<Integer>());
        return result;
    }

    private void helper(TreeNode root, List<List<Integer>> result, int sum, List<Integer> list) {
        if(root == null) {
            return;
        }
        // No deep copy of input 'list' here such as 'List<Integer> tmp = new ArrayList<Integer>(list)',
        // instead the change directly happen on input 'list' as adding new value on it, which change
        // the 'list' object and will pass into onwards recursion, to remove the change of 'list' object,
        // we have to use backtrack technic
        list.add(root.val);
        if(root.left == null && root.right == null) {
            if(sum == root.val) {
                result.add(new ArrayList<Integer>(list));
            }
        }
        helper(root.left, result, sum - root.val, list);
        // Do not backtrack here(before the right branch recursion), since we suppose to change
        // on 'list' should reflect in both left and right branch, if add backtrack here will
        // make right branch onwards recursion based on wrong version of 'list' that without change
        helper(root.right, result, sum - root.val, list);
        // Backtrack: Remove the last element on list for next recursion
        // We have to add explicit backtrack on 'list' because there is no deep copy as 'tmp'
        // in this solution, the change directly happen on input 'list' and if no rollback on
        // that change, the change will pass through all recursion levels, if we have a deep
        // copy 'tmp', the change will only happen on 'tmp' and impact current recursion level
        // which when return to previous recursion level, the 'tmp' impact will gone, we will
        // find our unchanged 'list' back in previous recursion level, that's implicit backtrack
```

```
        // in deep copy DFS style, since no deep copy here requires explicit backtrack on 'list'
        list.remove(list.size() - 1);
    }
}

Time Complexity: O(n^2), where n is number of nodes in the Binary Tree
Space Complexity: O(n)
```

Style 2: 1ms beats 100%, the promotion comes from direct Backtrack and Return on leaf node, it will save two more next recursion calls which will eventually return when root == null

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        helper(root, result, sum, new ArrayList<Integer>());
        return result;
    }

    private void helper(TreeNode root, List<List<Integer>> result, int sum, List<Integer> list) {
        if(root == null) {
            return;
        }
        list.add(root.val);
        if(root.left == null && root.right == null) {
            if(sum == root.val) {
                result.add(new ArrayList<Integer>(list));
                // The promotion comes from direct Backtrack and Return on leaf node,
```

```
                // it will save two more next recursion calls which will eventually
                // return when root == null
                list.remove(list.size() - 1);
                return;
            }
        }
        helper(root.left, result, sum - root.val, list);
        // Do not backtrack here(before the right branch recursion), since we suppose to change
        // on 'list' should reflect in both left and right branch, if add backtrack here will
        // make right branch onwards recursion based on wrong version of 'list' that without change
        helper(root.right, result, sum - root.val, list);
        // Backtrack: Remove the last element on list for next recursion
        list.remove(list.size() - 1);
    }
}

Time Complexity: O(n^2), where n is number of nodes in the Binary Tree
Space Complexity: O(n)
```

Solution 3: Iterative Inorder traversal with One Stack (360 min, based on L94.Binary Tree Inorder Traversal)

```
/**
 * 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;
 *     }
```

```java
 * }
 */
class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int targetSum) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        if(root == null) {
            return result;
        }
        TreeNode prev = null;
        int pathSum = 0;
        List<Integer> path = new ArrayList<Integer>();
        Stack<TreeNode> stack = new Stack<TreeNode>();
        // No modification on tree structure, can use original object 'root' to traverse
        // Similar style as L94.Binary Tree Inorder Traversal
        while(root != null || !stack.isEmpty()) {
            // Find as left as possible from root to leaf
            while(root != null) {
                stack.push(root);
                path.add(root.val);
                pathSum += root.val;
                root = root.left;
            }
            root = stack.peek();
            // Check if current node has right subtree and not a duplicate go through,
            // only when its first visit the right subtree we go to right subtree root,
            // for a new right subtree we should start over from the outside while loop
            // all find as left as possible steps
            if(root.right != null && root.right != prev) {
                root = root.right;
                continue;
            }
            // Check leaf node for potential path
            if(root.left == null && root.right == null && pathSum == targetSum) {
                result.add(new ArrayList<Integer>(path));
            }
            // Remove current node
            stack.pop();
```

```
                prev = root;
                // Subtract current node's val from path sum
                pathSum -= root.val;
                // As this current node is done, remove it from the current path
                path.remove(path.size() - 1);
                // Reset current node to null, so check the next item from the stack
                root = null;
            }
        return result;
    }
}

Time Complexity: O(n^2), where n is number of nodes in the Binary Tree
Space Complexity: O(n)
```

Refer to

https://leetcode.com/problems/path-sum-ii/discuss/36695/Java-Solution:-iterative-and-recursive/34840

```
public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> list = new ArrayList<>();
        if (root == null) return list;
        List<Integer> path = new ArrayList<>();
        Stack<TreeNode> s = new Stack<>();
        // sum along the current path
        int pathSum = 0;
        TreeNode prev = null;
        TreeNode curr = root;
        while (curr != null || !s.isEmpty()){
            // go down all the way to the left leaf node
            // add all the left nodes to the stack
            while (curr != null){
                s.push(curr);
                // record the current path
                path.add(curr.val);
                // record the current sum along the current path
                pathSum += curr.val;
```

```
                curr = curr.left;
            }
            // check left leaf node's right subtree
            // or check if it is not from the right subtree
            // why peek here?
            // because if it has right subtree, we don't need to push it back
            curr = s.peek();
            if (curr.right != null && curr.right != prev){
                curr = curr.right;
                continue; // back to the outer while loop
            }
            // check leaf
            if (curr.left == null && curr.right == null && pathSum == sum){
                list.add(new ArrayList<Integer>(path));
                // why do we need new arraylist here?
                // if we are using the same path variable path
                // path will be cleared after the traversal
            }
            // pop out the current value
            s.pop();
            prev = curr;
            // subtract current node's val from path sum
            pathSum -= curr.val;
            // as this current node is done, remove it from the current path
            path.remove(path.size()-1);
            // reset current node to null, so check the next item from the stack
            curr = null;
        }
        return list;
    }
```

How iterative Inorder traversal with One Stack step by step ?

```
e.g
               5
              /   \
             3     6
```

```
           /  \    \
          2    4    7
Go down all the way to the left leaf node add all the left nodes to the stack
                                                                    ===
                                                                     2  push 2
                                       ===                          ---
                                        3  push 3                    3
          ===                          ---                          ---
push 5 ->  5  ---> curr=3, push 3 --->  5  ---> curr=2, push 2 --->  5  ---> curr=null -> curr=s.peek()=2 --->
          ===                          ===                          ===                curr.right==null
         curr=5                       curr=3                       curr=2
         root=5                       root=5                       root=5
         prev=null                    prev=null                    prev=null
         path={5}                     path={5,3}                   path={5,3,2}
         pathSum=5                    pathSum=8                    pathSum=10

                                                 ===
                                                  3
check leaf                                       ---
curr.left=null ---> 1.pop out current value --->  5  ---> curr=null -> curr=s.peek()=3 -> curr=curr.right=4,continue ->
curr.right=null     prev=curr=2                  ===      curr.right=4!=null, curr.right=4!=prev=2
pathSum=10!=sum=12  2.subtract current node's    prev=2
                    val from path sum            pathSum=8
                    pathSum=10-2=8               path={5,3}
                    3.reset current node to      curr=null
                    null, so check the next
                    item from the stack
                    curr=null

          ===
           4  push 4
          ---                                                                                ===
           3                                                                                  3
          ---                                          check leaf                            ---
```

```
push 4 ->  5  ---> curr=null -> curr=s.peek()=4 --->  curr.left=null ---> 1. pop out current value --->  5  --->
          ===      curr.right=null                    curr.right=null     prev=curr=4                 ===
         curr=4                                       pathSum=12==sum=12  2. subtract current node's
         root=5                                       result={{5,3,4}}    val from path sum
         prev=2                                                           pathSum=12-4=8
         path={5,3,4}                                                     3.reset current node to
         pathSum=12                                                       null, so check the next
                                                                          item from the stack
                                                                          curr=null

                                           check leaf          ===
curr=null -> curr=s.peek()=3 ------------>  curr=3 not a leaf ---> 1. pop out current value --->  5  --->
curr.right=4!=null, curr.right=4==prev=4                          prev=curr=3          ===
we have visited curr.right=4 node before,                         2. subtract current node's
no need to visit again                                            val from path sum
                                                                  pathSum=8-3=5
                                                                  3.reset current node to
                                                                  null, so check the next
                                                                  item from the stack
                                                                  curr=null
                                                                       ===
                                                                        6  push 6
                                                                       ---
curr=null -> curr=s.peek()=5 -> curr=curr.right=6,continue -> push 6 ->  5  ---> curr=null -> curr=s.peek()=6... etc
curr.right=6!=null, curr.right=6!=prev=3                               ===
                                                                      curr=6
                                                                      root=5
                                                                      prev=3
                                                                      path={5,6}
```

```
pathSum=11
```

Why do we have a "prev" there? What does "curr.right != prev" exactly do?

https://leetcode.com/problems/path-sum-ii/discuss/36695/Java-Solution:-iterative-and-recursive/759308

It ensures that we **do not visit a right subtree again**. Let's say we did not have curr.right != prev check before we visit the right subtree. Consider the following case of 3 nodes:

```
      parent
      / \
   node1  node2
```

1. We start moving left until curr becomes null (curr = node1.left = null), adding parent and node1 to the stack. Then we set curr to stack.peek() which is the node1.Since node1.right is null, we do not need to traverse its right subtree. We then pop node1 from the stack, set curr to null and pre to node1.
2. In the next iteration of while loop, since curr is null, we skip the part where we continually traverse left. We set curr to stack.peek(), which is parent.Now we check if parent.right exists, and it does, so we will set curr to parent.right = node2. Since node2 has no children, it is exactly the same scenario as node1 and just like before in step 1), we will pop node2 from stack after traversing, setting curr to null and pre to node2.
3. This is where the problem will happen without the curr.right != prev check. Since curr is null, we skip the part where we continually traverse left.We set curr to stack.peek(), which is parent. Now when we want to traverse right, since parent.right != null we would have revisited the right subtree again if we did not check if curr.right != prev. So you can see how the prev variable actually stores the **most recently visited subtree when some nodes have "resolved"** so that in the event it is the right subtree of a parent node, we do not get stuck in an infinite loop revisiting the same right subtree.

Why not "curr=s.pop() early + no need s.pop() later" ?

Failed on test:

```
Input: [5,4,8,11,null,13,4,7,2,null,null,5,1], 22

                   5
                 /   \
                4      8
               /      /   \
              11     13     4
             /  \          /  \
            7     2       5     1
```

```
Output: [[5,4,11,2]]
Expected: [[5,4,11,2],[5,8,4,5]]
```

```java
/**
 * 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<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> list = new ArrayList<>();
        if (root == null) return list;
        List<Integer> path = new ArrayList<>();
        Stack<TreeNode> s = new Stack<>();
        // sum along the current path
        int pathSum = 0;
        TreeNode prev = null;
        TreeNode curr = root;
        while (curr != null || !s.isEmpty()){
            // go down all the way to the left leaf node
            // add all the left nodes to the stack
            while (curr != null){
                s.push(curr);
                // record the current path
                path.add(curr.val);
```

```java
                // record the current sum along the current path
                pathSum += curr.val;
                curr = curr.left;
            }
            // check left leaf node's right subtree
            // or check if it is not from the right subtree
            // why peek here?
            // because if it has right subtree, we don't need to push it back
            //curr = s.peek();
            curr = s.pop();
            if (curr.right != null && curr.right != prev){
                curr = curr.right;
                continue; // back to the outer while loop
            }
            // check leaf
            if (curr.left == null && curr.right == null && pathSum == sum){
                list.add(new ArrayList<Integer>(path));
                // why do we need new arraylist here?
                // if we are using the same path variable path
                // path will be cleared after the traversal
            }
            // pop out the current value
            //s.pop();
            prev = curr;
            // subtract current node's val from path sum
            pathSum -= curr.val;
            // as this current node is done, remove it from the current path
            path.remove(path.size()-1);
            // reset current node to null, so check the next item from the stack
            curr = null;
        }
        return list;
    }
    public static void main(String[] args) {
        /**
                5
               /  \
             4     8
            /     /  \
          11    13    4
```

```java
             /  \        /  \
            7    2      5    1
            */
            Test b = new Test();
            TreeNode five_a = b.new TreeNode(5);
            TreeNode four_a = b.new TreeNode(4);
            TreeNode eight = b.new TreeNode(8);
            TreeNode eleven = b.new TreeNode(11);
            TreeNode thirteen = b.new TreeNode(13);
            TreeNode four_b = b.new TreeNode(4);
            TreeNode seven = b.new TreeNode(7);
            TreeNode two = b.new TreeNode(2);
            TreeNode five_b = b.new TreeNode(5);
            TreeNode one = b.new TreeNode(1);

            five_a.left = four_a;
            five_a.right = eight;
            four_a.left = eleven;
            eight.left = thirteen;
            eight.right = four_b;
            eleven.left = seven;
            eleven.right = two;
            four_b.left = five_b;
            four_b.right = one;
            List < List < Integer >> result = b.pathSum(five_a, 22);
            System.out.println(result);
        }

        private class TreeNode {
            public int val;
            public TreeNode left, right;
            public TreeNode(int val) {
                this.val = val;
                this.left = this.right = null;
            }
        }
    }
```

Wrong code version with "curr=s.pop() + no need s.pop() later"

Take the above example, the issue is happen if we pop out current node early as 'curr=s.pop()' before check right subtree, in our example, after first round as left as possible traversal, stack s={5, 4, 11, 7}, and following pop out current node early logic it will pop 7 out, s={5,4,11}, then since 7 is leaf node no right substree after it and not match target sum condition, at the end of first round we set 'curr=null' and move ahead to next round which suppose check next element on stack, till now, no difference between correct logic as "curr=s.peek() + s.pop() later" and wrong logic as "curr=s.pop() early", but in second round, if follow wrong logic, it will pop out 11 and s={5,4}, since 11 has right subtree, after reach its right subtree leaf node 2, it will hit 'continue' logic and in third round we will push 2 onto stack, s ={5,4,2}, then directly pop 2 out, s={5,4}, yes, then the logic superficially still looks fine since it will hit target sum match logic and result get one path as {5,4,11,2}, but stack status is quite wrong, it will pop out 4 now and s={5}, then pop out 5 and s={}.

In conclusion, wrong stack status flow:

s={5, 4, 11, 7}

s={5, 4, 11}

s={5, 4} --> wrong operation as s.pop() early

s={5, 4, 2}

s={5, 4}

s={5}

s={} --> wrong operation as s.pop() early

Which eventually result into missing of second combination as {5,8,4,5} majorly locate on right subtree.

To compare, the correct stack status flow:

s={5, 4, 11, 7}

s={5, 4, 11}

s={5, 4, 11, 2} --> correct operation as s.peek()

s={5, 4, 11}

s={5, 4}

s={5}

s={5, 8} --> wrong operation as s.pop() early

.... etc

So it suppose not pop out current node early, have to reserve the current node but check only by peek() function in case current node has right subtree and requires direct continue to next round, otherwise when pop out current node early and move on to next round with 'continue'

we will wrongly pop out same path parent nodes stored on stack and miss other branch check.

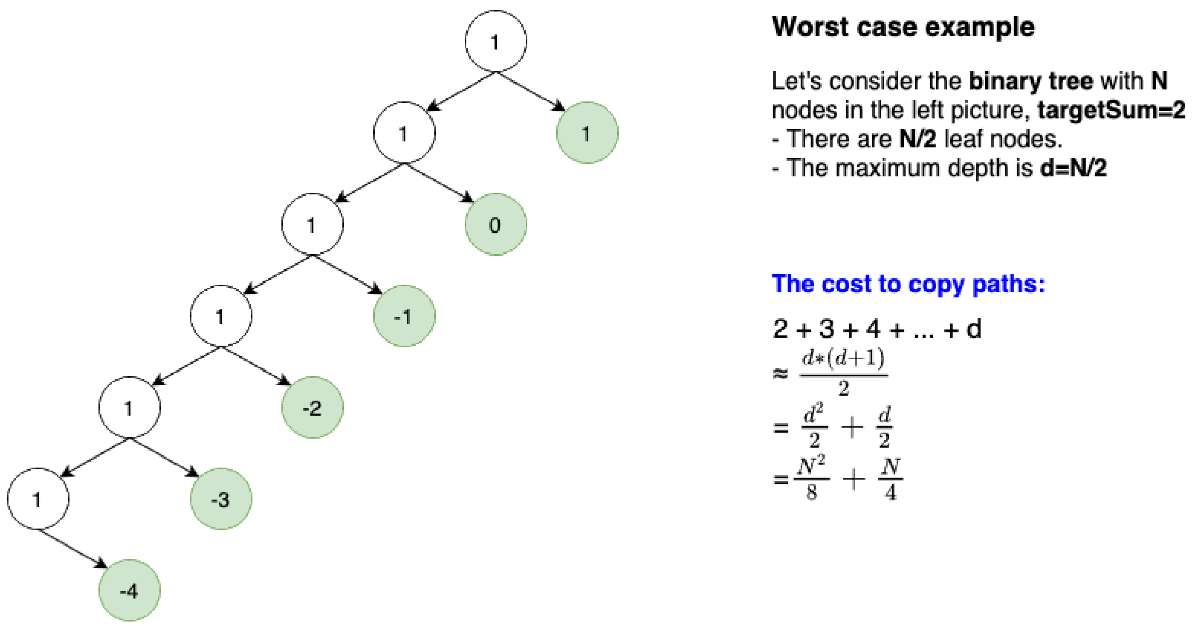
In conclusion: Reserve current node but check its status with peek() function before identify if right subtree exist or not, then after handling leaf node we are able to pop out current node and prepare for next round.

Complexity Analysis

https://leetcode.com/problems/path-sum-ii/discuss/1382332/C%2B%2BPython-DFS-Clean-and-Concise-Time-complexity-explained

Time: O(N^2), where N <= 5000 is the number of elements in the binary tree.

- First, we think the time complexity is O(N) because we only visit each node once.
- But we forgot to calculate the cost to copy the current path when we found a valid path, which in the worst case can cost O(N^2), let see the following example for more clear.



- Extra Space (without counting output as space): O(H), where H is height of the binary tree. This is the space for stack recursion or keeping path so far.

Refer to

L94.Binary Tree Inorder Traversal (Ref.L98,L230,L144,L145)

L112.P9.1.Path Sum (Ref.L257,L113)

L257.Binary Tree Paths (Ref.L1430,L549,L124)