Tree Problems

[310. Minimum Height Trees (https://leetcode.com/problems/minimum-height-trees/)](https://leetcode.com/problems/minimum-height-trees/)

Medium

For an undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Format

The graph contains n nodes which are labeled from 0 to n - 1. You will be given the number n and a list of undirected edges (each edge is a pair of labels).

You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.

Example 1 :

```
Input: n = 4, edges = [[1, 0], [1, 2], [1, 3]]

        0
        |
        1
       / \
      2   3

Output: [1]
```

Example 2 :

```
Input: n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]

     0  1  2
      \ | /
        3
        |
        4
        |
        5

Output: [3, 4]
```

Note:

According to the definition of tree on Wikipedia: “a tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.” The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

Solution

Keep removing leaves until there are less than 2 nodes left. The remaining nodes are the roots of the MHTs.

Performance

- Runtime: 100 ms, faster than 67.06% of Python3 online submissions for Minimum Height Trees.
- Memory Usage: 15.6 MB, less than 89.19% of Python3 online submissions for Minimum Height Trees.

Complexity Analysis

```
O(n + m) in time.
O(n + m) in space.
```

In [1]:

```python
from collections import defaultdict

class Solution:
    def findMinHeightTrees(self, n: 'int', edges: 'List[List[int]]') -> 'List[int]':
        if n == 1:
            return [0]

        degree = [0]*n
        adj_dict = defaultdict(list)
        for edge in edges:
            degree[edge[0]] += 1
            degree[edge[1]] += 1
            adj_dict[edge[0]].append(edge[1])
            adj_dict[edge[1]].append(edge[0])

        leaves = [i for i, d in enumerate(degree) if d == 1]
        n_nodes = n
        while n_nodes > 2:
            n_leaves = len(leaves)
            n_nodes -= n_leaves
            for i in range(n_leaves):
                leaf = leaves.pop(0)
                linked_node = adj_dict[leaf][0]
                degree[leaf] -= 1
                degree[linked_node] -= 1
                adj_dict[leaf] = []
                adj_dict[linked_node].remove(leaf)
                if degree[linked_node] == 1:
                    leaves.append(linked_node)
        return leaves

my_sol = Solution()
print('Should print [1]:', my_sol.findMinHeightTrees(4, [[1, 0], [1, 2], [1, 3]]))
print('Should print [3, 4]:', my_sol.findMinHeightTrees(6, [[0, 3], [1, 3], [2, 3], [4,
print('Should print [0]:', my_sol.findMinHeightTrees(1, []))
print('Should print [0]:', my_sol.findMinHeightTrees(3, [[0,1],[0,2]]))
```

```
Should print [1]: [1]
Should print [3, 4]: [3, 4]
Should print [0]: [0]
Should print [0]: [0]
```

[37. House Robber III (https://leetcode.com/problems/house-robber-iii/)](https://leetcode.com/problems/house-robber-iii/)

Medium

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

Example 1:

```
Input: [3,2,3,null,3,null,1]

     (3)
     / \
    2   3
     \   \
     (3) (1)

Output: 7
```

Explanation: Maximum amount of money the thief can rob = (3) + (3) + (1) = 7.

Example 2:

```
Input: [3,4,5,1,3,null,1]

      3
     / \
   (4) (5)
   / \   \
  1   3   1

Output: 9
```

Explanation: Maximum amount of money the thief can rob = (4) + (5) = 9.

Solution

- The solution is recursive
- Our recursive function calculates the max value in both cases
  - it returns (max_val_if_notrobbed, max_val_if_robbed)
  - the final answer is the max over these
- If we don't rob the current node the solution is the max robbed value of the two sub trees (over both cases)
- If we do rob the current node, the soltion is the sum of the value of the node and the max robbed value of the subtrees assuming neither child is robbed

Performance

- Runtime: 60 ms, faster than 70.49% of Python3 online submissions for House Robber III.
- Memory Usage: 15.3 MB, less than 5.09% of Python3 online submissions for House Robber III.

Complexity Analysis

```
O(?) in time.
O(n) in space.
```

In [2]:
```python
class BinaryTree:
    def __init__(self, root):
        self.root = TreeNode(root)

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution:
    def rob(self, root: 'TreeNode') -> 'int':
        return max(self.robRec(root))

    # returns (max_val_if_notrobbed, max_val_if_robbed)
    def robRec(self, node: 'TreeNode') -> ('int', 'int'):
        if not node:
            return 0, 0
        left, right = self.robRec(node.left), self.robRec(node.right)
        # (node not robbed, node robbed and left not robbed and right not robbed)
        return max(left) + max(right), node.val + left[0] + right[0]

my_sol = Solution()

# [3,2,3,null,3,null,1]
tree = BinaryTree(3)
tree.root.left = TreeNode(2)
tree.root.right = TreeNode(3)
#tree.root.left.left = TreeNode()
tree.root.left.right = TreeNode(3)
#tree.root.right.left = TreeNode()
tree.root.right.right = TreeNode(1)
print('Should print 7:', my_sol.rob(tree.root))

# [3,4,5,1,3,null,1]
tree = BinaryTree(3)
tree.root.left = TreeNode(4)
tree.root.right = TreeNode(5)
tree.root.left.left = TreeNode(1)
tree.root.left.right = TreeNode(3)
#tree.root.right.left = TreeNode()
tree.root.right.right = TreeNode(1)
print('Should print 9:', my_sol.rob(tree.root))
```

```
Should print 7: 7
Should print 9: 9
```

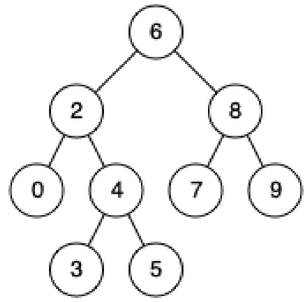
[235. Lowest Common Ancestor of a Binary Search Tree (https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/)](https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/)

Easy

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Given binary search tree: root = [6,2,8,0,4,7,9,null,null,3,5]



Example 1:

```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
```

Explanation: The LCA of nodes 2 and 8 is 6.

Example 2:

```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
```

Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

Note:

- All of the nodes' values will be unique.
- p and q are different and both values will exist in the BST.

Solution

- We traverse the tree looking for the lca
- There are 3 cases to consider at each node
  1. if p and q are both larger than the current node, the lca is a right descendent
  2. if p and q are both smaller than the current node, the lca is a left descendent
  3. if p and q are either side of the current node, the current node is the lca

Performance

- Runtime: 88 ms, faster than 65.50% of Python3 online submissions for Lowest Common Ancestor of a Binary Search Tree.
- Memory Usage: 17.3 MB, less than 5.18% of Python3 online submissions for Lowest Common Ancestor of a Binary Search Tree.

Complexity Analysis

```
O(log n) average case, O(n) worst case in time.
O(1) in space.
```

In [3]:
```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self, root):
        self.root = TreeNode(root)

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> '
        if q.val == p.val: return p.val
        if q.val <  p.val: return self.lowestCommonAncestor(root, q, p)
        a, b = p.val, q.val
        node = root
        while node:
            val = node.val
            if a <= val <= b: break
            if val < a < b: node = node.right
            if a < b < val: node = node.left
        return node.val

my_sol = Solution()
#[6,2,8,0,4,7,9,null,null,3,5]
tree = BinaryTree(6)
tree.root.left = TreeNode(2)
tree.root.right = TreeNode(8)
tree.root.left.left = TreeNode(0)
tree.root.left.right = TreeNode(4)
tree.root.right.left = TreeNode(7)
tree.root.right.right = TreeNode(9)
#tree.root.left.left.left = TreeNode()
#tree.root.left.left.right = TreeNode()
tree.root.left.right.left = TreeNode(3)
tree.root.left.right.right = TreeNode(5)
#tree.root.right.left.left = TreeNode()
#tree.root.right.left.right = TreeNode()
#tree.root.right.right.left = TreeNode()
#tree.root.right.right.right = TreeNode()
print('Should print 6:', my_sol.lowestCommonAncestor(tree.root, TreeNode(2), TreeNode(8
print('Should print 2:', my_sol.lowestCommonAncestor(tree.root, TreeNode(2), TreeNode(4
```

```
Should print 6: 6
Should print 2: 2
```

[230. Kth Smallest Element in a BST (https://leetcode.com/problems/kth-smallest-element-in-a-bst/)](https://leetcode.com/problems/kth-smallest-element-in-a-bst/)

Medium

Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.

Note: You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

Example 1:

```
Input: root = [3,1,4,null,2], k = 1
               3
              / \
             1   4
              \
               2
Output: 1
```

Example 2:

```
Input: root = [5,3,6,2,4,null,null,1], k = 3
                   5
                  / \
                 3   6
                / \
               2   4
              /
             1
Output: 3
```

Follow up:

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Solution

- Perform a depth first 'in-order' traversal, i.e. `left child => parent => right child`
- Record the path
- Return the kth value on the path

Performance

- Runtime: 60 ms, faster than 83.13% of Python3 online submissions for Kth Smallest Element in a BST.
- Memory Usage: 17.2 MB, less than 5.88% of Python3 online submissions for Kth Smallest Element in a BST.

Complexity Analysis

```
O(log n) average case, O(n) worst case in time.
O(k) in space.
```

In [4]:
```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self, root):
        self.root = TreeNode(root)

class Solution:
    def kthSmallest(self, root: 'TreeNode', k: 'int') -> 'int':
        path = []
        self.inOrderDFS(root, k, path)
        print(path)
        return path[k-1]

    def inOrderDFS(self, node, k, path):
        if node and len(path) < k:
            self.inOrderDFS(node.left, k, path)
            path.append(node.val)
            self.inOrderDFS(node.right, k, path)

my_sol = Solution()

#[3,1,4,null,2]
tree = BinaryTree(3)
tree.root.left = TreeNode(1)
tree.root.right = TreeNode(4)
#tree.root.left.left = TreeNode()
tree.root.left.right = TreeNode(2)
print('Should print 1:', my_sol.kthSmallest(tree.root, 1))

#[5,3,6,2,4,null,null,1]
tree = BinaryTree(5)
tree.root.left = TreeNode(3)
tree.root.right = TreeNode(6)
tree.root.left.left = TreeNode(2)
tree.root.left.right = TreeNode(4)
#tree.root.right.left = TreeNode()
#tree.root.right.right = TreeNode()
tree.root.left.left.left = TreeNode(1)
print('Should print 3:', my_sol.kthSmallest(tree.root, 3))
```

```
[1, 3]
Should print 1: 1
[1, 2, 3, 5]
Should print 3: 3
```

[113. Path Sum II (https://leetcode.com/problems/path-sum-ii/)](https://leetcode.com/problems/path-sum-ii/)

Medium

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

Note: A leaf is a node with no children.

Example:

```
Given the below binary tree and sum = 22,

                  5
                 / \
                4   8
               /   / \
              11  13  4
             /  \    / \
            7    2  5   1
Return:

[[5,4,11,2],
 [5,8,4,5]]
```

Solution

Recursive Depth First Search - 'pre-order' traversal

- We keep track of our current path as a list of nodes in a list
- Each time we reach a leaf check, we the sum along the path
- If the sum is as required, save the path to a list of paths
- Check all paths to leaves

Performance

- Runtime: 56 ms, faster than 83.14% of Python3 online submissions for Path Sum II.
- Memory Usage: 14.7 MB, less than 55.81% of Python3 online submissions for Path Sum II.

Complexity Analysis

```
O(log n) average case, O(n) worst case in time.
O(log n) in space.
```

In [5]:
```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self, root):
        self.root = TreeNode(root)

class Solution:
    def pathSum(self, root: 'TreeNode', path_sum: 'int') -> 'List[List[int]]':
        path = []
        paths = []
        self.findPathsRec(root, path_sum, path, paths)
        return paths

    def findPathsRec(self, node, path_sum, path, paths):
        if node:
            path.append(node.val)
            self.findPathsRec(node.left, path_sum, path, paths)
            self.findPathsRec(node.right, path_sum, path, paths)
            if not(node.left) and not(node.right) and sum(path) == path_sum:
                paths.append(path[:])
            path.pop()

my_sol = Solution()

#[5,4,8,11,null,13,4,7,2,null,null,null,null,5,2]
tree = BinaryTree(5)

tree.root.left = TreeNode(4)
tree.root.right = TreeNode(8)

tree.root.left.left = TreeNode(11)
#tree.root.left.right = TreeNode()
tree.root.right.left = TreeNode(13)
tree.root.right.right = TreeNode(4)

tree.root.left.left.left = TreeNode(7)
tree.root.left.left.right = TreeNode(2)
#tree.root.left.right.left = TreeNode()
#tree.root.left.right.right = TreeNode()
#tree.root.right.left.left = TreeNode()
#tree.root.right.left.right = TreeNode()
tree.root.right.right.left = TreeNode(5)
tree.root.right.right.right = TreeNode(1)
print('Should print [[5, 4, 11, 2], [5, 8, 4, 5]]:', my_sol.pathSum(tree.root, 22))
```

```
Should print [[5, 4, 11, 2], [5, 8, 4, 5]]: [[5, 4, 11, 2], [5, 8, 4, 5]]
```

[Second largest element in BST (https://www.geeksforgeeks.org/second-largest-element-in-binary-search-tree-bst/)](https://www.geeksforgeeks.org/second-largest-element-in-binary-search-tree-bst/)

Given a Binary Search Tree(BST), find the second largest element.

Example 1:

```
Input: Root of below BST

              10
             /
            5

Output: 5
```

Example 1:

```
Input: Root of below BST

              10
             /  \
            5    20
                   \
                    30

Output: 20
```

Solution

- We traverse the tree going right until we get to a node we get to a node that has a right child but whose right child does not have a right child.
- There are three important cases to consider.
  - In the diagrams below a and c are the first and third largest nodes respectively.
  - In each case, using the algorithm described above we and up at 'node' and return *.

  (i) node = c, node.left and not(node.right)

  ```
  c
   \
    a
   /
  *
  ```

  (ii) node = *, not(node.left) and node.right

  ```
  c
   \
    *
     \
      a
  ```

  (iii) node = *, node.left and node.right

  ```
   \
    *
   / \
  c   a
  ```

Complexity Analysis

```
O(log n) in time.
O(1) in space.
```

In [6]:

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self, root):
        self.root = TreeNode(root)

class Solution:
    def SecondLargest(self, root: 'TreeNode') -> 'int':
        # if there are less than 2 nodes in the tree, return None
        if not(root) or (not(root.left) and not(root.right)):
            return None
        node = root
        # keep going right until there is only one right descendent
        while node.right and node.right.right:
            node = node.right
        if node.right.left:
            return node.right.left.val
        return node.val

my_sol = Solution()

# [2, 5, 6]
tree = BinaryTree(5)
tree.root.left = TreeNode(2)
tree.root.right = TreeNode(6)
print('Should print 5:', my_sol.SecondLargest(tree.root))

tree = BinaryTree(4)
tree.root.left = TreeNode(2)
tree.root.left.left = TreeNode(1)
tree.root.left.right = TreeNode(3)
tree.root.right = TreeNode(6)
tree.root.right.left = TreeNode(5)
tree.root.right.right = TreeNode(7)
print('Should print 6:', my_sol.SecondLargest(tree.root))

tree = BinaryTree(4)
tree.root.left = TreeNode(2)
tree.root.left.left = TreeNode(1)
tree.root.left.right = TreeNode(3)
tree.root.right = TreeNode(6)
tree.root.right.left = TreeNode(5)
#tree.root.right.right = TreeNode(7)
print('Should print 5:', my_sol.SecondLargest(tree.root))

tree = BinaryTree(4)
tree.root.left = TreeNode(2)
tree.root.left.left = TreeNode(1)
tree.root.left.right = TreeNode(3)
tree.root.right = TreeNode(6)
#tree.root.right.left = TreeNode(5)
tree.root.right.right = TreeNode(7)
print('Should print 6:', my_sol.SecondLargest(tree.root))
```

```
Should print 5: 5
Should print 6: 6
Should print 5: 5
Should print 6: 6
```

[104. Maximum Depth of Binary Tree (https://leetcode.com/problems/maximum-depth-of-binary-tree/)](https://leetcode.com/problems/maximum-depth-of-binary-tree/)

Easy

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Note: A leaf is a node with no children.

Example:

Given binary tree [3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
   15   7
```

return its depth = 3.

Solution

- We calculate this recursively

  depth(tree) = 1 + max(depth(left_sub_tree, right_sub_tree))

Performance

- Runtime: 48 ms, faster than 87.33% of Python3 online submissions for Maximum Depth of Binary Tree.
- Memory Usage: 15.6 MB, less than 5.13% of Python3 online submissions for Maximum Depth of Binary Tree.

Complexity Analysis

- O(n) in time.
- O(n) in space.

In [7]:
```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if not(root):
            return 0
        return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1

my_sol = Solution()
root = TreeNode(3)
root.left = TreeNode(9)
root.right = TreeNode(20)
root.right.left = TreeNode(15)
root.right.right = TreeNode(7)
print('Should print 3:', my_sol.maxDepth(root))
```

```
Should print 3: 3
```

[449. Serialize and Deserialize BST (https://leetcode.com/problems/serialize-and-deserialize-bst/)](https://leetcode.com/problems/serialize-and-deserialize-bst/)

Medium

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary search tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary search tree can be serialized to a string and this string can be deserialized to the original tree structure.

The encoded string should be as compact as possible.

Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

Example 1:

```
string: '3 1 2 4'

tree
           3
          / \
         1   4
          \
           2
```

Example 2:

```
string: '5 3 2 1 4 6'

tree:
              5
             / \
            3   6
           / \
          2   4
         /
        1
```

Solution

- Serializing is easy, just preorder traverse the tree using recursion to create an array of values
- This is easily converted to a string with a node separator of choice - in my case a space.
- Deserializing is trickier
- First convert the string to a queue of values
- We turn the values into nodes, building the tree recursively,
- Each call returns the node it builds
  - First we build the root node
  - Then build the left tree
  - Then build the right tree
- Once a node has been built it can be popped from the queue and the new queue is passed to the recursion
- The recursive function takes the min and max values of the tree it is building
- To start we set max and min values at ±inf
  - For the left tree, the max value is updated to the root value
  - For the right tree, the min value is updated to the root value

Performance

- Runtime: 68 ms, faster than 65.58% of Python online submissions for Serialize and Deserialize BST.
- Memory Usage: 19.7 MB, less than 8.14% of Python online submissions for Serialize and Deserialize BST.

Complexity

- O(n) in time.
- O(n) in space.

In [8]:
```python
import sys
from collections import deque

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Codec:

    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        strings = self.preOrder(root)
        return ' '.join(strings)

    def preOrder(self, node):
        if node:
            return [str(node.val)] + self.preOrder(node.left) + self.preOrder(node.righ
        return []

    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        vals = deque([int(val) for val in data.split()])
        return self.buildTree(vals, -sys.maxsize, sys.maxsize)

    def buildTree(self, vals, min_val, max_val):
        if vals and min_val < vals[0] < max_val:
            val = vals.popleft()
            node = TreeNode(val)
            node.left = self.buildTree(vals, min_val, val)
            node.right = self.buildTree(vals, val, max_val)
            return node

# Your Codec object will be instantiated and called as such:
# codec = Codec()
# codec.deserialize(codec.serialize(root))

my_codec = Codec()

#[3,1,4,null,2]
root = TreeNode(3)
root.left = TreeNode(1)
root.right = TreeNode(4)
#root.left.left = TreeNode()
root.left.right = TreeNode(2)
print('Should print 3 1 2 4:', my_codec.serialize(root))
print('Should print 3 1 2 4:', my_codec.serialize(my_codec.deserialize('3 1 2 4')))

#[5,3,6,2,4,null,null,1]
root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(6)
root.left.left = TreeNode(2)
root.left.right = TreeNode(4)
```

```
#root.right.left = TreeNode()
#root.right.right = TreeNode()
root.left.left.left = TreeNode(1)
print('Should print 5 3 2 1 4 6:', my_codec.serialize(root))
print('Should print 5 3 2 1 4 6:', my_codec.serialize(my_codec.deserialize('5 3 2 1 4 6
```

```
Should print 3 1 2 4: 3 1 2 4
Should print 3 1 2 4: 3 1 2 4
Should print 5 3 2 1 4 6: 5 3 2 1 4 6
Should print 5 3 2 1 4 6: 5 3 2 1 4 6
```

96. Unique Binary Search Trees (https://leetcode.com/problems/unique-binary-search-trees/?tab=Description)

Medium

Given n, how many structurally unique BST's (binary search trees) that store values 1 ... n?

Example:

```
Input: 3
Output: 5
```

Explanation:

Given n = 3, there are a total of 5 unique BST's:

```
1         3     3      2      1
 \       /     /      / \      \
  3     2     1      1   3      2
 /     /       \                 \
2     1         2                 3
```

Solution

```
G(n) = the number of unique BSTs that can be made from n nodes.
F(n, j) = the number of unique BSTs that can be made from n nodes  with node i
 at the root
```

Clearly then

```
G(n) = sum[ F(n, j) ]    for    1 <= j <= n

G(0) = G(1) = 1
```

Since the left sub-tree can only contain nodes which are smaller than i and the right sub-tree can only contain nodes that are larger than i, we have:

```
F(n, j) = G(j-1) * G(n-j)    for    1 <= j <= n
```

Combining formulas we can obtain a recursive formula exclusively in `G`.

```
G(n) = sum[ G(j-1) * G(n-j) ]    for    1 <= j <= n

     = G(0) * G(n-1) + G(1) * G(n-2) + G(2) * G(n-3) + ... + G(n-1) * G(0)
```

We simply calculate create a list `G[i]` and calculate the values for `1 <= i <= n` in ascending order.

Performance

- Runtime: 44 ms, faster than 21.06% of Python3 online submissions for Unique Binary Search Trees.
- Memory Usage: 13.1 MB, less than 5.56% of Python3 online submissions for Unique Binary Search Trees.

Complexity Analysis

- O(n^2) in time.
- O(n) in space.

In [9]:
```python
class Solution:
    def numTrees(self, n: int) -> int:
        G = [0]*(n+1)
        G[0] = G[1] = 1
        for i in range(2, n+1):
            for j in range(1, i+1):
                G[i] += G[j-1] * G[i-j]
        return G[n]

my_sol = Solution()
print('Should print 5:', my_sol.numTrees(3))
```

```
Should print 5: 5
```

[199. Binary Tree Right Side View (https://leetcode.com/problems/binary-tree-right-side-view/?tab=Description)](https://leetcode.com/problems/binary-tree-right-side-view/?tab=Description)

Medium

Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

Example:

```
Input: [1,2,3,null,5,null,4]
Output: [1, 3, 4]
```

Explanation:

```
     1            <===
   /   \
  2     3         <===
   \     \
    5     4       <===
```

Solution

- Like a reverse pre-order DFS where you go right instead of left
- Keep track of the depth in the tree, only record the node if the depth is

Some tests:

```
Input:
           3
          / \
         1   4
          \
           2

Output: [3, 4, 2]
```

Example 2:

```
Input:
               5
              / \
             3   6
            / \
           2   4
          /
         1

Output: [5, 6, 4, 1]
```

Performance

- Runtime: 40 ms, faster than 94.34% of Python3 online submissions for Binary Tree Right Side View.
- Memory Usage: 13.4 MB, less than 5.13% of Python3 online submissions for Binary Tree Right Side View.

Complexity Analysis

```
O(n) in time.
O(log n) in space.
```

In [10]:

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution:
    def rightSideView(self, root: TreeNode) -> 'List[int]':
        nodes = []
        self.preOrderRightDFS(root, 0, nodes)
        return nodes

    def preOrderRightDFS(self, node, depth, nodes):
        #print('test1: val =', node.val, 'depth =', depth, 'right side =', nodes)
        if node:
            if depth == len(nodes):
                nodes.append(node.val)
                #print('test2: val =', node.val, 'depth =', depth, 'right side =', node
            if node.right:
                self.preOrderRightDFS(node.right, depth+1, nodes)
            if node.left:
                self.preOrderRightDFS(node.left, depth+1, nodes)

my_sol = Solution()

#[3,1,4,null,2]
root = TreeNode(3)
root.left = TreeNode(1)
root.right = TreeNode(4)
#root.left.left = TreeNode()
root.left.right = TreeNode(2)
print('Should print [3, 4, 2]:', my_sol.rightSideView(root))
print()

#[5,3,6,2,4,null,null,1]
root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(6)
root.left.left = TreeNode(2)
root.left.right = TreeNode(4)
#root.right.left = TreeNode()
#root.right.right = TreeNode()
root.left.left.left = TreeNode(1)
print('Should print [5, 6, 4, 1]:', my_sol.rightSideView(root))
```

```
Should print [3, 4, 2]: [3, 4, 2]

Should print [5, 6, 4, 1]: [5, 6, 4, 1]
```

In [ ]: