Most frequently asked coding questions with solutions in Python
Part-2

Table of Contents

Linked List Cycle

Given a linked list, determine if it has a cycle in it.

Follow up: Can you solve it without using extra space?

URL: https://leetcode.com/problems/linked-list-cycle/

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def hasCycle(self, head):
        """
        :type head: ListNode
        :rtype: bool
        """
        if head == None:
            return False
        else:
            fast = head
            slow = head

            while fast != None and fast.next != None:
                slow = slow.next
                fast = fast.next.next
                if fast == slow:
                    break

            if fast == None or fast.next == None:
                return False
            elif fast == slow:
                return True

            return False
```

Reverse Linked List

Reverse a singly linked list.

URL: https://leetcode.com/problems/reverse-linked-list/

```
# Definition for singly-linked list.
# class ListNode(object):
#     def___init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def reverseList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if head == None:
            return None
        elif head != None and head.next == None:
            return head
        else:
            temp = None
            next_node = None
            while head != None:
                next_node = head.next
                head.next  =  temp
                temp = head
                head = next_node

            return temp
```

Delete Node in a Linked List

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Supposed the linked list is 1 -> 2 -> 3 -> 4 and you are given the third node with value 3, the linked list should become 1 -> 2 -> 4 after calling your function.

URL: https://leetcode.com/problems/delete-node-in-a-linked-list/

```
# Definition for singly-linked list.
# class ListNode(object):
#     def___init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def deleteNode(self, node):
        """
        :type node: ListNode
        :rtype: void Do not return anything, modify node in-plac
e instead.
        """
        if node == None:
            pass
        else:
            next_node = node.next
            node.val = next_node.val
            node.next = next_node.next
```

Merge Two Sorted Lists

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

URL: https://leetcode.com/problems/merge-two-sorted-lists/

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def mergeTwoLists(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        if l1 == None and l2 == None:
            return None
        elif l1 != None and l2 == None:
            return l1
        elif l2 != None and l1 == None:
            return l2
        else:
            dummy = ListNode(0)
            p = dummy

            while l1 != None and l2 != None:
                if l1.val < l2.val:
                    p.next = l1
                    l1 = l1.next
                else:
                    p.next = l2
                    l2 = l2.next
                p = p.next

            if l1 != None:
                p.next = l1

            if l2 != None:
                p.next = l2

            return dummy.next
```

Intersection of Two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

A: a1 → a2 ↘ c1 → c2 → c3 ↗
B: b1 → b2 → b3 begin to intersect at node c1.

Notes:

If the two linked lists have no intersection at all, return null. The linked lists must retain their original structure after the function returns. You may assume there are no cycles anywhere in the entire linked structure. Your code should preferably run in O(n) time and use only O(1) memory.

URL: https://leetcode.com/problems/intersection-of-two-linked-lists/

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def getIntersectionNode(self, headA, headB):
        """
        :type head1, head1: ListNode
        :rtype: ListNode
        """
        if headA == None and headB == None:
            return None
        elif headA == None and headB != None:
            return None
        elif headA != None and headB == None:
            return None
        else:
            len_a = 0
```

```
              len_b = 0

              current = headA
              while current != None:
                  current = current.next
                  len_a += 1

              current = headB
              while current != None:
                  current = current.next
                  len_b += 1

              diff = 0
              current = None
              if len_a > len_b:
                  diff = len_a - len_b
                  currentA = headA
                  currentB = headB
              else:
                  diff = len_b - len_a
                  currentA = headB
                  currentB = headA

              count = 0
              while count  <  diff:
                  currentA = currentA.next
                  count += 1

              while currentA != None and currentB != None:
                  if currentA == currentB:
                      return currentA
                  else:
                      currentA = currentA.next
                      currentB = currentB.next
```

Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Note: Do not modify the linked list.

Follow up: Can you solve it without using extra space?

URL: https://leetcode.com/problems/linked-list-cycle-ii/

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def detectCycle(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if head == None:
            return head
        else:
            fast = head
            slow = head

            has_cycle = False
            while fast != None and fast.next != None:
                slow = slow.next
                fast = fast.next.next
                if fast == slow:
                    has_cycle = True
                    break

            if has_cycle == False:
                return None

            slow = head
            while fast != slow:
                fast = fast.next
                slow = slow.next

            return slow
```

Palindrome Linked List

Given a singly linked list, determine if it is a palindrome.

Follow up: Could you do it in O(n) time and O(1) space?

URL: https://leetcode.com/problems/palindrome-linked-list/

```
# Definition for singly-linked list.
# class ListNode(object):
#     def___init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def isPalindrome(self, head):
        """
        :type head: ListNode
        :rtype: bool
        """
        if head == None:
            return True
        elif head != None and head.next == None:
            return True
        else:
            fast = head
            slow = head
            stack = []
            while fast != None and fast.next != None:
                stack.append(slow.val)
                slow    =    slow.next
                fast = fast.next.next

            #madam
            if fast != None:
                slow = slow.next

            while slow != None:
                if slow.val != stack.pop():
                    return False
                else:
                    slow = slow.next

            return True
```

Remove Linked List Elements

Remove all elements from a linked list of integers that have value val.

Example Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, val = 6 Return: 1 --> 2 --> 3 --> 4 --> 5

URL: https://leetcode.com/problems/remove-linked-list-elements/

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def removeElements(self, head, val):
        """
        :type head: ListNode
        :type val: int
        :rtype: ListNode
        """
        if head == None:
            return head
        elif head != None and head.next == None:
            if head.val == val:
                return None
            else:
                return head
        else:
            dummy = ListNode(0)
            dummy.next = head
            prev = dummy

            while head != None:
                if head.val == val:
                    prev.next = head.next
                    head = prev
                prev  =  head
                head = head.next

            return dummy.next
```

Remove Duplicates from Sorted Linked List

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example, Given 1->1->2, return 1->2. Given 1->1->2->3->3, return 1->2->3.

URL: https://leetcode.com/problems/remove-duplicates-from-sorted-list/

```
# Definition for singly-linked list.
# class ListNode(object):
#     def___init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def deleteDuplicates(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if head == None:
            return head
        elif head != None and head.next == None:
            return head
        else:
            lookup = {}
            current = head
            prev = head
            while current != None:
                if current.val in lookup:
                    prev.next = prev.next.next
                else:
                    lookup[current.val] = True
                    prev = current
                current = current.next

            return head
```

Remove Duplicates from Sorted Linked List II

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example, Given 1->2->3->3->4->4->5, return 1->2->5. Given 1->1->1->2->3, return 2->3.

URL: https://leetcode.com/problems/remove-duplicates-from-sorted-list-ii/

```
# Definition for singly-linked list
# class ListNode(object):
#     def___init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def deleteDuplicates(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if head == None:
            return head
        else:
            dup_dict = {}
            current = head
            while current != None:
                if current.val in dup_dict:
                    dup_dict[current.val] += 1
                else:
                    dup_dict[current.val] = 1
                current = current.next

            list_values = []
            current = head
            while current != None:
```

```
                if dup_dict[current.val] > 1:
                    pass
                else:
                    list_values.append(current.val)
                current = current.next
            if list_values == []:
                return None
            else:
                node1 = ListNode(list_values[0])
                head = node1
                for entries in list_values[1:]:
                    new_node = ListNode(entries)
                    node1.next = new_node
                    node1 = new_node

                return head
```

Remove Nth node from End of List

Given a linked list, remove the nth node from the end of list and return its head.

For example,

Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.
Note: Given n will always be valid. Try to do this in one pass.

URL: https://leetcode.com/problems/remove-nth-node-from-end-of-list/

```
# Definition for singly-linked list.
# class ListNode(object):
#     def___init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def removeNthFromEnd(self, head, n):
        """
        :type head: ListNode
        :type n: int
        :rtype: ListNode
        """
        if head == None:
            return head
        else:
            dummy = ListNode(0)
            dummy.next = head
            fast = dummy
            slow = dummy
            for i in range(n):
                fast = fast.next

            while fast.next != None:
                fast = fast.next
                slow = slow.next

            slow.next = slow.next.next

            return dummy.next
```

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

For example, you may serialize the following tree

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

as

`"[1,2,3,null,null,4,5]"`

, just the same as

how LeetCode OJ serializes a binary tree

. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

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

URL: https://leetcode.com/problems/serialize-and-deserialize-binary-tree/

```
# 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___init__(self):
        self.serialized_array = []
        self.index = 0

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

        :type root: TreeNode
        :rtype: str
        """
        self.serialization_help(root)
        return self.serialized_array


    def serialization_help(self, root):
        if root == None:
            self.serialized_array.append(None)
            return
        self.serialized_array.append(root.val)
        self.serialize(root.left)
        self.serialize(root.right)

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

        :type data: str
        :rtype: TreeNode
        """
        if self.index == len(data) or data[self.index] == None:
            self.index += 1
            return None

        root = TreeNode(data[self.index])
        self.index += 1
        root.left = self.deserialize(data)
        root.right = self.deserialize(data)
        return root
```

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

Preorder Traversal

Given a binary tree, return the preorder traversal of its nodes' values.

For example: Given binary tree {1,#,2,3}, 1 \ 2 / 3 return [1,2,3].

Note: Recursive solution is trivial, could you do it iteratively?

URL: https://leetcode.com/problems/binary-tree-preorder-traversal/

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

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def preorderTraversal(self, root):
        if root == None:
            return []
        else:
            preorderList = []
            stack = []
            stack.append(root)
            while(stack != []):
                node = stack.pop()
                preorderList.append(node.val)
                if node.right:
                    stack.append(node.right)
                if node.left:
                    stack.append(node.left)
            return preorderList
```

BST Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

Note: next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

URL: https://leetcode.com/problems/binary-search-tree-iterator/

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

class BSTIterator:
    # @param root, a binary search tree's root node
    def___init__(self, root):
        self.stack = []
        node = root
        while node != None:
            self.stack.append(node)
            node = node.left


    # @return a boolean, whether we have a next smallest number
    def hasNext(self):
        return len(self.stack) != 0


    # @return an integer, the next smallest number
    def next(self):
        nextNode = self.stack.pop()
        currentNode = nextNode.right
        while currentNode != None:
            self.stack.append(currentNode)
            currentNode = currentNode.left
        return nextNode.val


# Your BSTIterator will be called like this:
# i, v = BSTIterator(root), []
# while i.hasNext(): v.append(i.next())
```

Inorder Traversal

Given a binary tree, return the inorder traversal of its nodes' values.

For example: Given binary tree [1,null,2,3], 1 \ 2 / 3 return [1,3,2].

Note: Recursive solution is trivial, could you do it iteratively?

URL: https://leetcode.com/problems/binary-tree-inorder-traversal/

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

class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def inorderTraversal(self, root):
        if root == None:
            return []
        else:
            result = []
            stack = []
            node = root
            while stack or node:
                if node:
                    stack.append(node)
                    node = node.left
                else:
                    node = stack.pop()
                    result.append(node.val)
                    node = node.right
            return result
```

Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

Note: Bonus points if you could solve it both recursively and iteratively.

URL: https://leetcode.com/problems/symmetric-tree/

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

class Solution(object):
    def isSymmetric(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        if root == None:
            return True
        else:
            return self.isMirror(root.left, root.right)

    def isMirror(self, root1, root2):
        if root1 == None and root2 == None:
            return True
        elif root1 == None or root2 == None:
            return False
        else:
            if root1.val == root2.val:
                return self.isMirror(root1.left, root2.right) an
d self.isMirror(root1.right, root2.left)
            else:
                return False
```

Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

URL: https://leetcode.com/problems/balanced-binary-tree/

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

class Solution:
    # @param {TreeNode} root
    # @return {boolean}

    def getHeight(self, root):
        if root == None:
            return 0

        leftHeight = self.getHeight(root.left)
        if leftHeight == -1:
            return -1

        rightHeight = self.getHeight(root.right)
        if rightHeight == -1:
            return -1

        heightDiff = abs(leftHeight - rightHeight)
        if heightDiff > 1:
            return -1
        else:
            return max(leftHeight,rightHeight)+1


    def isBalanced(self, root):
        if self.getHeight(root) == -1:
            return False
        else:
            return True
```

Closest Binary Search Tree Value

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note: Given target value is a floating point. You are guaranteed to have only one unique value in the BST that is closest to the target.

URL: https://leetcode.com/problems/closest-binary-search-tree-value/

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

class Solution(object):
    def closestValue(self, root, target):
        """
        :type root: TreeNode
        :type target: float
        :rtype: int
        """
        min_dif = float("inf")
        closestVal = None

        if root == None:
            return None
        else:
            while root:
                root_val = root.val
                val_dif = abs(root_val - target)
                if val_dif < min_dif:
                    min_dif = val_dif
                    closestVal = root_val
                if target < root_val:
                    if root.left != None:
                        root = root.left
                    else:
                        root = None
                else:
                    if root.right != None:
                        root = root.right
                    else:
                        root = None
        return closestVal
```

Postorder Traversal

Given a binary tree, return the postorder traversal of its nodes' values.

For example: Given binary tree {1,#,2,3}, 1 \ 2 / 3 return [3,2,1].

Note: Recursive solution is trivial, could you do it iteratively?

URL: https://leetcode.com/problems/binary-tree-postorder-traversal/

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

class Solution(object):
    def postorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        if root == None:
            return []
        else:
            stack = []
            out_stack = []
            stack.append(root)

            while stack != []:
                current = stack.pop()
                out_stack.append(current.val)
                if current.left != None:
                    stack.append(current.left)
                if current.right != None:
                    stack.append(current.right)

            return out_stack[::-1]
```

Maximum Depth of Binary Tree

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.

URL: https://leetcode.com/problems/maximum-depth-of-binary-tree/

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

class Solution(object):
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if root == None:
            return 0
        else:
            return max(self.maxDepth(root.left), self.maxDepth(r
oot.right)) + 1
```

Invert Binary Tree

Invert a binary tree.

```
    4
```

/ \ 2 7 / \ / \ 1 3 6 9 to 4 / \ 7 2 / \ / \ 9 6 3 1 Trivia: This problem was inspired by this original tweet by Max Howell: Google: 90% of our engineers use the software you wrote (Homebrew), but you can’t invert a binary tree on a whiteboard so fuck off. URL: https://leetcode.com/problems/invert-binary-tree/

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

class Solution(object):
    def invertTree(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        if root == None:
            return None
        else:
            stack = []
            stack.append(root)
            while stack != []:
                curr_node = stack.pop()
                if curr_node.left != None or curr_node.right !=
None:
                    temp = curr_node.left
                    curr_node.left = curr_node.right
                    curr_node.right = temp
                if curr_node.right != None:
                    stack.append(curr_node.right)
                if curr_node.left != None:
                    stack.append(curr_node.left)
            return root
```

Same Tree

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

URL: https://leetcode.com/problems/same-tree/

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


class Solution:
    # @param {TreeNode} p
    # @param {TreeNode} q
    # @return {boolean}
    def isSameTree(self, p, q):
        if p == None and q == None:
            return True
        else:
            if p == None or q == None:
                return False
            else:
                if p.val == q.val:
                    return self.isSameTree(p.left, q.left) and s
elf.isSameTree(p.right, q.right)
                else:
                    return False
```

Lowest Common Ancestor of a Binary Search Tree

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 v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).”

```
        _______6______
       /              \
   ___2__          ___8__
```

/ \ / \ 0 _4 7 9 / \ 3 5 For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

URL: https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/

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

class Solution(object):

    def __init__(self):
        self.inorder_list = []
        self.postorder_list = []

    def lowestCommonAncestor(self, root, p, q):
        """
```

```
        :type root: TreeNode
        :type p: TreeNode
        :type q: TreeNode
        :rtype: TreeNode
        """
        if root == None:
            return None
        else:
            self.inorder_traversal(root)
            self.postorder_traversal(root)
            #get the positions of node1 and node2 in the inorder
 traversal of the tree
            index_node1 = self.inorder_list.index(p.val)
            index_node2 = self.inorder_list.index(q.val)

            if index_node1 < index_node2:
                between_elems = self.inorder_list[index_node1 :
index_node2 + 1]
            else:
                between_elems = self.inorder_list[index_node2 :
index_node1 + 1]


            lca_elem = self.find_elem_max_index(between_elems)

            return lca_elem

    def find_elem_max_index(self, between_elems):
        max_index = -1
        elem = None
        for entries in between_elems:
            elem_index = self.postorder_list.index(entries)
            if elem_index > max_index:
                max_index = elem_index
                elem = entries
        return elem

    def inorder_traversal(self, node):
        if node:
            self.inorder_traversal(node.left)
```

```
            self.inorder_list.append(node.val)
            self.inorder_traversal(node.right)

    def postorder_traversal(self, node):
        if node:
            self.postorder_traversal(node.left)
            self.postorder_traversal(node.right)
            self.postorder_list.append(node.val)
```

Lowest Common Ancestor in a Binary Tree

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

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

```
        ______3______
       /              \
    ___5__          ___1__
```

/ \ / \ 6 _2 0 8 / \ 7 4 For example, the lowest common ancestor (LCA) of nodes 5 and 1 is 3. Another example is LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

URL: https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/

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

class Solution(object):

    def lowestCommonAncestor(self, root, p, q):
        """
        :type root: TreeNode
        :type p: TreeNode
        :type q: TreeNode
        :rtype: TreeNode
        """
        if root == None:
            return None

        if root == p or root == q:
            return root

        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)

        if left != None and right != None:
            return root

        if left == None:
            return right
        else:
            return left
```

Unique Binary Search Trees

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

For example, 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

URL: https://leetcode.com/problems/unique-binary-search-trees/

```
class Solution(object):


    def numTrees(self, n):
        """
        :type n: int
        :rtype: int
        """
        solutions = [-1]*(n)
        return self.numUniqueBST(n, solutions)

    def numUniqueBST(self, n, solutions):

        if n < 0:
            return 0

        if n == 0 or n == 1:
            return 1

        possibilities = 0

        for i in range(0, n):
            if solutions[i] == -1:
                solutions[i] = self.numUniqueBST(i, solutions)
            if solutions[n-1-i] == -1:
                solutions[n-1-i] = self.numUniqueBST(n-1-i, solu
tions)
            possibilities += solutions[i]*solutions[n-1-i]
        return possibilities
```

Unique Binary Search Trees II

Given an integer n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example, Given n = 3, your program should return all 5 unique BST's shown below.

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

URL: https://leetcode.com/problems/unique-binary-search-trees-ii/

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

class Solution(object):
    def generateTrees(self, n):
        """
        :type n: int
        :rtype: List[TreeNode]
        """
        if n == 0:
            return []
        else:
            return self.tree_constructor(1, n)

    def tree_constructor(self, m, n):
        results = []
        if m > n:
            results.append(None)
            return results

        for i in range(m, n+1):
            l = self.tree_constructor(m, i-1)
            r = self.tree_constructor(i+1, n)
            for left_trees in l:
                for right_trees  in  r:
                    curr_node = TreeNode(i)
                    curr_node.left = left_trees
                    curr_node.right = right_trees
                    results.append(curr_node)


        return results
```

Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example: Given the below binary tree and sum = 22, 5 / \ 4 8 / / \ 11 13 4 / \ \ 7 2 1 return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

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

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

class Solution(object):

    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        if root == None:
            return False
        else:
            current = root
            s = []
            s.append(current)
            s.append(current.val)

            while s != []:
                pathsum = s.pop()
                current = s.pop()

                if not current.left and not current.right:
```

```
        if pathsum == sum:
            return True

    if current.right:
        rightpathsum = pathsum + current.right.val
        s.append(current.right)
        s.append(rightpathsum)

    if current.left:
        leftpathsum = pathsum + current.left.val
        s.append(current.left)
        s.append(leftpathsum)

return False
```

Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.

For example: Given the below binary tree,

```
     1
    / \
   2   3
```

Return 6

URL: https://leetcode.com/problems/binary-tree-maximum-path-sum/

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

class Solution(object):
    def___init__(self):
        self.maxSum = -sys.maxint - 1

    # @param {TreeNode} root
    # @return {integer}
    def maxPathSum(self, root):
        self.findMax(root)
        return self.maxSum

    def findMax(self,root):
        if root == None:
            return 0
        left = self.findMax(root.left)
        right = self.findMax(root.right)
        self.maxSum = max(root.val + left + right, self.maxSum)
        ret = root.val + max(left,right)
        if ret < 0:
            return 0
        else:
            return ret
```

Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example: Given binary tree [3,9,20,null,null,15,7], 3 / \ 9 20 / \ 15 7 return its level order traversal as: [[3], [9,20], [15,7]]

URL: https://leetcode.com/problems/binary-tree-level-order-traversal/

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

import Queue
class Solution:
    # @param {TreeNode} root
    # @return {integer[][]}
    def levelOrder(self, root):
        if root == None:
            return []
        else:
            q = Queue.Queue()
            q.put(root)
            q.put("#")
            levelOrderTraversal = []
            level = []
            while q.empty() == False:
                node = q.get()
                if node == "#":
                    if q.empty() == False:
                        q.put("#")
                    levelOrderTraversal.append(level)
                    level = []
                else:
                    level.append(node.val)
                    if node.left:
                        q.put(node.left)
                    if node.right:
                        q.put(node.right)

            return levelOrderTraversal
```

Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key. The right subtree of a node contains only nodes with keys greater than the node's key. Both the left and right subtrees must also be binary search trees. Example 1: 2 / \ 1 3 Binary tree [2,1,3], return true. Example 2: 1 / \ 2 3 Binary tree [1,2,3], return false.

URL: https://leetcode.com/problems/validate-binary-search-tree/

```
# Definition for a binary tree node.
# class TreeNode:
#     def___init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
import sys
class Solution:
    def___init__(self):
        self.lastPrinted = -sys.maxsize-1
    # @param {TreeNode} root
    # @return {boolean}
    def isValidBST(self, root):
        if root == None:
            return True

        if self.isValidBST(root.left) == False:
            return False

        data = root.val
        if data <= self.lastPrinted:
            return False

        self.lastPrinted = data

        if self.isValidBST(root.right) == False:
            return False

        return True
```

Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

URL: https://leetcode.com/problems/minimum-depth-of-binary-tree/

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

class Solution(object):
    def minDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if root == None:
            return 0

        if root.left == None and root.right == None:
            return 1

        if root.left != None:
            left = self.minDepth(root.left)
        else:
            left = sys.maxsize

        if root.right != None:
            right = self.minDepth(root.right)
        else:
            right = sys.maxsize

        return 1 + min(left, right)
```

Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

URL: https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/

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

class Solution(object):
    def sortedArrayToBST(self, nums):
        """
        :type nums: List[int]
        :rtype: TreeNode
        """
        if nums == []:
            return None
        elif len(nums) == 1:
            return TreeNode(nums[0])
        else:
            start = 0
            end = len(nums) - 1
            return self.to_bst(nums, start, end)


    def to_bst(self, arr, start, end):
        if len(arr) == 0 or start > end:
            return None
        else:
            mid = (start + end) // 2
            node = TreeNode(arr[mid])
            node.left = self.to_bst(arr, start, mid - 1)
            node.right = self.to_bst(arr, mid + 1, end)
              return node`
```

Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

For example, Given

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

The flattened tree should look like: 1 \ 2 \ 3 \ 4 \ 5 \ 6

URL: https://leetcode.com/problems/flatten-binary-tree-to-linked-list/

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

class Solution(object):
    def flatten(self, root):
        """
        :type root: TreeNode
        :rtype: void Do not return anything, modify root in-plac
e instead.
        """
        if root == None:
            return root

        stack = []
        current = root

        while((stack != []) or (current != None)):

            if current.right != None:
                stack.append(current.right)

            if current.left != None:
                current.right = current.left
                current.left = None
            else:
                if stack != []:
                    temp = stack.pop()
                    current.right = temp

            current = current.right
```

Construct Binary Tree from Inorder and Preorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

Note: You may assume that duplicates do not exist in the tree.

URL: https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/

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

class Solution(object):
    def buildTree(self, preorder, inorder):
        """
        :type preorder: List[int]
        :type inorder: List[int]
        :rtype: TreeNode
        """
        if len(inorder) == 1:
            return TreeNode(inorder[0])
        return self.create_tree(inorder, 0, len(inorder) - 1, pr
eorder, 0, len(preorder) - 1)

    def search_divindex(self, inorder, low_inorder, high_inorder
, val):
        for i in range(low_inorder, high_inorder+1):
            if inorder[i] == val:
                return i
        return -1

    def create_tree(self, inorder, low_inorder, high_inorder, pr
eorder, low_preorder, high_preorder):
```

```
        if (low_preorder > high_preorder) or (low_inorder > high
_inorder):
            return None

        root = TreeNode(preorder[low_preorder])
        div_index = self.search_divindex(inorder, low_inorder, h
igh_inorder, root.val)
        size_left_subtree = div_index - low_inorder
        size_right_subtree = high_inorder - div_index

        root.right = self.create_tree(inorder, div_index + 1, hi
gh_inorder, preorder,
                                      low_preorder + 1 + size_le
ft_subtree,
                                      low_preorder + size_left_s
ubtree + size_right_subtree)

        root.left = self.create_tree(inorder, low_inorder, div_i
ndex - 1, preorder,
                                     low_preorder + 1, low_preor
der + size_left_subtree)

        return root
```

Binary Tree Paths

Given a binary tree, return all root-to-leaf paths.

For example, given the following binary tree:

1 / \ 2 3 \ 5 All root-to-leaf paths are:

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

URL: https://leetcode.com/problems/binary-tree-paths/

```
# class TreeNode:
#     def___init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {TreeNode} root
    # @return {string[]}
    def binaryTreePaths(self, root):
        if root == None:
            return []
        else:
            paths = []
            current = root
            s = []
            s.append(current)
            s.append(str(current.val))


            while s != []:
                #pathsum = s.pop()
                path = s.pop()
                current = s.pop()

                if not current.left and not current.right:
                    paths.append(path)
```

```
                if current.right:
                    rightstr = path + "->" + str(current.right.v
al)
                    s.append(current.right)
                    s.append(rightstr)

                if current.left:
                    leftstr = path + "->" + str(current.left.val
)
                    s.append(current.left)
                    s.append(leftstr)
            return paths
```

Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note: A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

URL: https://leetcode.com/problems/recover-binary-search-tree/

```
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def __init__(self):
        self.__prev = None
        self.__node1 = None
        self.__node2 = None

    def recoverTree(self, root):
        """
        :type root: TreeNode
        :rtype: void Do not return anything, modify root in-place instead.
        """
        self.recoverTreeHelp(root)
        temp = self.__node1.val
        self.__node1.val = self.__node2.val
        self.__node2.val = temp

    def recoverTreeHelp(self, root):
        if root == None:
            return None

        self.recoverTreeHelp(root.left)
        if self.__prev != None:
            if self.__prev.val > root.val:
                if self.__node1 == None:
                    self.__node1 = self.__prev
                self.__node2 = root
        self.__prev = root
        self.recoverTreeHelp(root.right)
```

Path Sum II

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

For 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]]

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

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

class Solution(object):
    def pathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: List[List[int]]
        """
        if root == None:
            return []
        else:
            stack = []
            paths = []

            current = root
            stack.append(current)
            stack.append([current.val])
            stack.append(current.val)
            while stack != []:
                pathsum = stack.pop()
                path = stack.pop()
                curr = stack.pop()
```

```
            if curr.left == None and curr.right == None:
                if pathsum == sum:
                    paths.append(path)
            if curr.right:
                rightstr = path + [curr.right.val]
                rightsum = pathsum + curr.right.val
                stack.append(curr.right)
                stack.append(rightstr)
                stack.append(rightsum)
            if curr.left:
                leftstr = path + [curr.left.val]
                leftsum = pathsum + curr.left.val
                stack.append(curr.left)
                stack.append(leftstr)
                stack.append(leftsum)
        return paths
```

Binary Level Order Traversal II

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example: Given binary tree [3,9,20,null,null,15,7], 3 / \ 9 20 / \ 15 7 return its bottom-up level order traversal as: [[15,7], [9,20], [3]]

URL: https://leetcode.com/problems/binary-tree-level-order-traversal-ii/

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

import Queue
class Solution:
    # @param {TreeNode} root
    # @return {integer[][]}
    def levelOrderBottom(self, root):
        if root == None:
            return []
        else:
            q = Queue.Queue()
            q.put(root)
            q.put("#")
            levelOrderTraversal = []
            level = []
            stack = []

            while q.empty() == False:
                node = q.get()
                if node == "#":
                    if q.empty() == False:
                        q.put("#")
                    stack.append(level)
```

```
                    level = []
                else:
                    level.append(node.val)
                    if node.left:
                        q.put(node.left)
                    if node.right:
                        q.put(node.right)

        while stack:
            levelOrderTraversal.append(stack.pop())

        return levelOrderTraversal
```

Kth Smallest Element in a BST

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.

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?

Hint:

Try to utilize the property of a BST. What if you could modify the BST node's structure? The optimal runtime complexity is O(height of BST).

URL: https://leetcode.com/problems/kth-smallest-element-in-a-bst/

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

class Solution(object):
    def kthSmallest(self, root, k):
        """
        :type root: TreeNode
        :type k: int
        :rtype: int
        """
        if root == None:
            return None
        else:
            stack = []
            node = root
            count = 0
            while stack!= [] or node != None:
                if node != None:
                    stack.append(node)
                    node = node.left
                else:
                    inorder_node = stack.pop()
                    count += 1
                    if count == k:
                        return inorder_node.val
                    node = inorder_node.right
            return None
```

```
class Solution(object):
    def kthSmallest(self, root, k):
        """
        :type root: TreeNode
        :type k: int
        :rtype: int
        """
        stack = []*k
        while True:
            while root:
                stack.append(root)
                root = root.left

            root = stack.pop()
            if k == 1:
                return root.val
            else:
                k -= 1
                root = root.right
```

Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

Note: You may assume that duplicates do not exist in the tree.

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

class Solution(object):
    def buildTree(self, inorder, postorder):
        """
        :type inorder: List[int]
        :type postorder: List[int]
        :rtype: TreeNode
        """
        return self.create_tree(inorder, 0, len(inorder) -1 , po
storder, 0, len(postorder) - 1)


    def search_divindex(self, inorder, low_inorder, high_inorder
, val):
        for i in range(low_inorder, high_inorder+1):
            if inorder[i] == val:
                return i
        return -1

    def create_tree(self, inorder, low_inorder, high_inorder, po
storder, low_postorder, high_postorder):
        if (low_inorder > high_inorder) or (low_postorder > high
_postorder):
            return None
```

```
        root = TreeNode(postorder[high_postorder])

        div_index = self.search_divindex(inorder, low_inorder, h
igh_inorder, root.val)

        size_left_subtree = div_index - low_inorder
        size_right_subtree = high_inorder - div_index

        root.right = self.create_tree(inorder, div_index + 1, hi
gh_inorder, postorder,
                                      high_postorder - size_righ
t_subtree, high_postorder - 1)

        root.left = self.create_tree(inorder, low_inorder, div_i
ndex - 1,
                                     postorder,
                                     high_postorder - size_right
_subtree - size_left_subtree,
                                     high_postorder - size_right
_subtree - 1)

        return root
```

Binary Tree Right Side View

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.

For example: Given the following binary tree, 1 <--- / \ 2 3 <--- \ \ 5 4 <--- You should return [1, 3, 4].

URL: https://leetcode.com/problems/binary-tree-right-side-view/

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

import Queue
class Solution:
    # @param {TreeNode} root
    # @return {integer[]}
    def rightSideView(self, root):
        if root == None:
            return []
        else:
            q = Queue.Queue()
            q.put(root)
            q.put("#")
            rightSideView = []
            level = []
            while q.empty() == False:
                node = q.get()
                if node == "#":
                    if q.empty() == False:
                        q.put("#")
                    rightSideView.append(level[-1])
                    level = []
                else:
                    level.append(node.val)
                    if node.left != None:
                        q.put(node.left)
                    if node.right != None:
                        q.put(node.right)
            return rightSideView
```

Sum Root to Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path 1->2->3 which represents the number 123.

Find the total sum of all root-to-leaf numbers.

For example,

```
1
```

/ \ 2 3 The root-to-leaf path 1->2 represents the number 12. The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

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

class Solution(object):
    def sumNumbers(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if root == None:
            return 0
        else:
            stack = []
            paths = []
            stack.append(root)
            stack.append(str(root.val))
            while stack != []:
                path = stack.pop()
                current = stack.pop()
                if current.left == None and current.right == Non
e:
                    paths.append(int(path))
                if current.right:
                    rightstr = path + str(current.right.val)
                    stack.append(current.right)
                    stack.append(rightstr)
                if current.left:
                    leftstr = path + str(current.left.val)
                    stack.append(current.left)
                    stack.append(leftstr)
            return sum(paths)
```

Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example: Given binary tree [3,9,20,null,null,15,7], 3 / \ 9 20 / \ 15 7 return its zigzag level order traversal as: [[3], [20,9], [15,7]]

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

import Queue
class Solution:
    # @param {TreeNode} root
    # @return {integer[][]}
    def zigzagLevelOrder(self, root):
        if root == None:
            return []
        else:
            q = Queue.Queue()
            q.put(root)
            q.put("#")
            levelOrderTraversal = []
            level = []
            levelNo = 0
            while q.empty() == False:
                node = q.get()

                if node == "#":
                    if q.empty() == False:
                        q.put("#")

                    if levelNo == 0 or levelNo % 2 == 0:
                        levelOrderTraversal.append(level)
```

```
                    else:
                        levelOrderTraversal.append(level[::-1])
                    level = []
                    levelNo += 1
                else:
                    level.append(node.val)
                    if node.left:
                        q.put(node.left)
                    if node.right:
                        q.put(node.right)

            return levelOrderTraversal
```

House Robber III

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: 3 / \ 2 3 \ \ 3 1 Maximum amount of money the thief can rob = 3 + 3 + 1 = 7. Example 2: 3 / \ 4 5 / \ \ 1 3 1 Maximum amount of money the thief can rob = 4 + 5 = 9.

URL: https://leetcode.com/problems/house-robber-iii/

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

class Solution(object):
    def rob(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if root == None:
            return 0
        else:
            result = self.rob_max(root)
            return max(result[0], result[1])

    def rob_max(self, root):
        if root == None:
            return [0, 0]
        else:
            left_res = self.rob_max(root.left)
            right_res = self.rob_max(root.right)
            result = [0]*2
            result[0] = root.val + left_res[1] + right_res[1]
            result[1] = max(left_res[0], left_res[1]) + max(righ
t_res[0], right_res[1])
            return result
```

Inorder Successor in BST

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

Note: If the given node has no in-order successor in the tree, return null.

URL: https://leetcode.com/problems/inorder-successor-in-bst/

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

class Solution(object):
    def inorderSuccessor(self, root, p):
        """
        :type root: TreeNode
        :type p: TreeNode
        :rtype: TreeNode
        """
        successor = None

        while root != None and root.val != p.val:
            if root.val > p.val:
                successor = root
                root = root.left
            else:
                root = root.right

        if root == None:
            return None

        if root.right == None:
            return successor


        root = root.right
        while root.left != None:
            root = root.left
        return root
```

Binary Tree Longest Consecutive Sequence

Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

For example, 1 \ 3 / \ 2 4 \ 5 Longest consecutive sequence path is 3-4-5, so return 3. 2 \ 3 / 2
/ 1 Longest consecutive sequence path is 2-3,not3-2-1, so return 2.

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

from Queue import Queue
import sys
class Solution(object):
    def longestConsecutive(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if root == None:
            return 0
        if root.right == None and root.left == None:
            return 1
        else:
            max_size = 1
            size_q = Queue()
            node_q = Queue()
            node_q.put(root)
```

```
        size_q.put(1)
        while node_q.empty() == False:
            curr_node = node_q.get()
            curr_size = size_q.get()

            if curr_node.left:
                left_size = curr_size
                if curr_node.val == curr_node.left.val - 1:
                    left_size += 1
                    max_size = max(max_size, left_size)
                else:
                    left_size = 1

                node_q.put(curr_node.left)
                size_q.put(left_size)

            if curr_node.right:
                right_size = curr_size
                if curr_node.val == curr_node.right.val - 1:
                    right_size += 1
                    max_size = max(max_size, right_size)
                else:
                    right_size = 1

                node_q.put(curr_node.right)
                size_q.put(right_size)

        return max_size
```

Verify Preorder Sequence in Binary Search Tree

Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree.

You may assume each number in the sequence is unique.

Follow up: Could you do it using only constant space complexity?

URL: https://leetcode.com/problems/verify-preorder-sequence-in-binary-search-tree/

```
import sys
class Solution(object):
    def verifyPreorder(self, preorder):
        """
        :type preorder: List[int]
        :rtype: bool
        """
        stack = []
        root = -sys.maxsize-1

        for entries in preorder:

            if entries < root:
                return False

            while stack != [] and stack[-1] < entries:
                root = stack.pop()

            stack.append(entries)

        return True
```

Binary Tree Upside Down

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

For example: Given a binary tree {1,2,3,4,5}, 1 / \ 2 3 / \ 4 5 return the root of the binary tree [4,5,2,#,#,3,1]. 4 / \ 5 2 / \ 3 1

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

class Solution(object):
    def upsideDownBinaryTree(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        p = root
        parent = None
        parent_right = None

        while p:
            left = p.left
            p.left = parent_right
            parent_right = p.right
            p.right = parent
            parent = p
            p = left

        return parent
```

Count Univalue Subtrees

Given a binary tree, count the number of uni-value subtrees.

A Uni-value subtree means all nodes of the subtree have the same value.

For example: Given binary tree, 5 / \ 1 5 / \ \ 5 5 5 return 4.

URL: https://leetcode.com/problems/count-univalue-subtrees/

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

class Solution(object):
    def __init__(self):
        self.__count = 0

    def countUnivalSubtrees(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        self.count_univalue_subtrees(root)
        return self.__count

    def count_univalue_subtrees(self, root):
        if root == None:
            return True
        if root.left == None and root.right == None:
            self.__count += 1
            return True
        left = self.count_univalue_subtrees(root.left)
        right = self.count_univalue_subtrees(root.right)

        if (left and right) and (root.left == None or root.left.val == root.val) and (root.right == None or root.right.val == root.val):
            self.__count += 1
            return True
        else:
            return False
```

Number of Connected Components in an Undirected Graph

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

URL : https://leetcode.com/problems/number-of-connected-components-in-an-undirected-graph/

```
import sys
from queue import Queue

class Vertex:
    def___init__(self, node):
        self.id = node
        self.adjacent = {}
        # Set distance to infinity for all nodes
        self.distance = sys.maxsize
        # Mark all nodes unvisited
        self.visited = False
        # Mark all nodes color with white
        self.color = 'white'
        # Predecessor
        self.previous = None

    def addNeighbor(self, neighbor, weight=0):
        self.adjacent[neighbor] = weight

    def getConnections(self):
        return self.adjacent.keys()

    def getVertexID(self):
        return self.id

    def getWeight(self, neighbor):
        return self.adjacent[neighbor]
```

```
    def setDistance(self, dist):
        self.distance = dist

    def getDistance(self):
        return self.distance

    def setColor(self, color):
        self.color = color

    def getColor(self):
        return self.color

    def setPrevious(self, prev):
        self.previous = prev

    def setVisited(self):
        self.visited = True

    def __str__(self):
        return str(self.id) + ' adjacent: ' + str([x.id for x in self.adjacent])

class Graph:
    def __init__(self):
        self.vertDictionary = {}
        self.numVertices = 0

    def __iter__(self):
        return iter(self.vertDictionary.values())

    def  addVertex(self,  node):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(node)
        self.vertDictionary[node] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertDictionary:
            return self.vertDictionary[n]
```

```
        else:
            return None

    def addEdge(self, frm, to, cost=0):
        if frm not in self.vertDictionary:
            self.addVertex(frm)
        if to not in self.vertDictionary:
            self.addVertex(to)

        self.vertDictionary[frm].addNeighbor(self.vertDictionary
[to], cost)
        self.vertDictionary[to].addNeighbor(self.vertDictionary[
frm], cost)

    def getVertices(self):
        return self.vertDictionary.keys()

    def setPrevious(self, current):
        self.previous = current

    def getPrevious(self, current):
        return self.previous

class Solution(object):
    def countComponents(self, n, edges):
        """
        :type n: int
        :type edges: List[List[int]]
        :rtype: int
        """
        if n == 1 and edges == []:
            return 1
        else:
            G = Graph()
            for entries in edges:
                G.addEdge(entries[0], entries[1], 1)
            count = 0
            for vertex in G:
                if vertex.getColor() == "white":
                    count += 1
```

```
                    self.bfs(vertex)

            return count

    def bfs(self, vertex):
        vertex.setColor("gray")
        q = Queue()
        q.put(vertex)
        while q.empty() == False:
            curr_node = q.get()
            for nbr in curr_node.getConnections():
                if nbr.getColor() == "white":
                    nbr.setColor("gray")
                    q.put(nbr)
            curr_node.setColor("black")


if __name__ == "__main__":

    n = 5
    edges1 = [[0, 1], [1, 2], [3, 4]]
    edges2 = [[0, 1], [1, 2], [2, 3], [3, 4]]

    soln = Solution()
    print(soln.countComponents(n, edges1))
    print(soln.countComponents(n, edges2))
```

Course Schedule

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

For example:

2, [[1,0]] There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

2, [[1,0],[0,1]] There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

Note: The input prerequisites is a graph represented by a list of edges, not adjacency matrices. Read more about how a graph is represented.

URL: https://leetcode.com/problems/course-schedule/

```
class Vertex:

    def __init__(self, key):
        self.id = key
        self.adjacent = {}
        self.indegree = 0
        self.outdegree = 0
        self.predecessor = None
        self.visit_time = 0
        self.finish_time = 0
        self.color = "white"

    def add_neighbor(self, nbr, weight=0):
        self.adjacent[nbr] = weight
```

```
    def  get_neighbors(self):
        return self.adjacent.keys()

    def get_id(self):
        return self.id

    def get_weight(self, nbr):
        return self.adjacent[nbr]

    def get_indegree(self):
        return self.indegree

    def set_indegree(self, indegree):
        self.indegree = indegree

    def get_outdegree(self):
        return self.outdegree

    def set_outdegree(self, outdegree):
        self.outdegree = outdegree

    def get_predecessor(self):
        return self.predecessor

    def set_predecessor(self, pred):
        self.predecessor = pred

    def get_visit_time(self):
        return self.visit_time

    def set_visit_time(self, visit_time):
        self.visit_time = visit_time

    def get_finish_time(self):
        return self.finish_time

    def set_finish_time(self, finish_time):
        self.finish_time = finish_time

    def get_color(self):
```

```
        return self.color

    def set_color(self, color):
        self.color = color

    def___str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x
 in self.adjacent])




class Graph:

    def___init__(self):
        self.vertex_dict = {}
        self.no_vertices = 0
        self.no_edges = 0

    def add_vertex(self, vert_key):
        new_vertex_obj = Vertex(vert_key)
        self.vertex_dict[vert_key] = new_vertex_obj
        self.no_vertices += 1

    def get_vertex(self, vert_key):
        if vert_key in self.vertex_dict:
            return self.vertex_dict[vert_key]
        else:
            return None

    def add_edge(self, fro, to, weight=1):
        if fro not in self.vertex_dict:
            self.add_vertex(fro)
            from_vertex = self.get_vertex(fro)
        else:
            from_vertex = self.vertex_dict[fro]

        if to not in self.vertex_dict:
            self.add_vertex(to)
            to_vertex = self.get_vertex(to)
```

```
        else:
            to_vertex = self.vertex_dict[to]

        from_vertex.add_neighbor(to_vertex, weight)
        from_vertex.set_outdegree(from_vertex.get_outdegree() +
1)
        to_vertex.set_indegree(to_vertex.get_indegree() + 1)
        self.no_edges += 1

    def get_edges(self):
        edges = []
        for u in self.vertex_dict:
            for v in self.vertex_dict[u].get_neighbors():
                u_id = u
                #print(v)
                v_id = v.get_id()
                edges.append((u_id, v_id, self.vertex_dict[u].ge
t_weight(v)))
        return edges

    def get_vertices(self):
        return self.vertex_dict


class DFS:

    def___init__(self, graph):
        self.graph = graph
        self.has_cycle = False

    def dfs(self):
        for vertex in self.graph.get_vertices():
            if self.graph.vertex_dict[vertex].get_color() == "wh
ite":
                self.dfs_visit(self.graph.vertex_dict[vertex])

    def dfs_visit(self, node):
        node.set_color("gray")
        for vert in node.get_neighbors():
            if vert.get_color() == "gray":
```

```
                self.has_cycle = True
            if vert.get_color() == "white":
                vert.set_color("gray")
                self.dfs_visit(vert)
        node.set_color("black")


class Solution(object):
    def canFinish(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: bool
        """
        if not prerequisites:
            return True
        else:
            g = Graph()

            for edge in prerequisites:
                g.add_edge(edge[0], edge[1])

            dfs_obj = DFS(g)
            dfs_obj.dfs()
            if dfs_obj.has_cycle == True:
                return False
            else:
                return True


if __name__ == "__main__":
    soln1 = Solution()
    print(soln1.canFinish(2, [[1,0]]))

    soln2 = Solution()
    print(soln2.canFinish(2, [[1,0],[0,1]]))
```

Graph Valid Tree

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:

Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], return false.

Hint:

Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], what should your return? Is this case a valid tree? 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.”

URL: https://leetcode.com/problems/graph-valid-tree/

```
import sys
class Vertex:
    def __init__(self, node):
        self.id = node
        self.adjacent = {}
        # Set distance to infinity for all nodes
        self.distance = sys.maxsize
        # Mark all nodes unvisited
        self.visited = False
        # Mark all nodes color with white
        self.color = 'white'
        # Predecessor
        self.previous = None

    def addNeighbor(self, neighbor, weight=0):
        self.adjacent[neighbor] = weight

    def getConnections(self):
```

```
        return self.adjacent.keys()

    def getVertexID(self):
        return self.id

    def getWeight(self, neighbor):
        return self.adjacent[neighbor]

    def setDistance(self, dist):
        self.distance = dist

    def getDistance(self):
        return self.distance

    def setColor(self, color):
        self.color = color

    def getColor(self):
        return self.color

    def setPrevious(self, prev):
        self.previous = prev

    def setVisited(self):
        self.visited = True

    def __str__(self):
        return str(self.id) + ' adjacent: ' + str([x.id for x in
 self.adjacent])

class Graph:
    def __init__(self):
        self.vertDictionary = {}
        self.numVertices = 0

    def __iter__(self):
        return iter(self.vertDictionary.values())

    def  addVertex(self,  node):
        self.numVertices = self.numVertices + 1
```

```
        newVertex = Vertex(node)
        self.vertDictionary[node] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertDictionary:
            return self.vertDictionary[n]
        else:
            return None

    def addEdge(self, frm, to, cost=0):
        if frm not in self.vertDictionary:
            self.addVertex(frm)
        if to not in self.vertDictionary:
            self.addVertex(to)

        self.vertDictionary[frm].addNeighbor(self.vertDictionary
[to], cost)
        self.vertDictionary[to].addNeighbor(self.vertDictionary[
frm], cost)

    def getVertices(self):
        return self.vertDictionary.keys()

    def setPrevious(self, current):
        self.previous = current

    def getPrevious(self, current):
        return self.previous

class Solution:
    def validTree(self, n, edges):
        """
        :type n: int
        :type edges: List[List[int]]
        :rtype: bool
        """
        if n == 1 and len(edges) == 0:
            return True
        elif self.check_input(n, edges) == False:
```

```
            return False
        elif n == 0 and len(edges) > 0:
            return False
        elif n == 1 and len(edges) >= 1:
            return False
        else:
            G = Graph()
            for entries in edges:
                G.addEdge(entries[0], entries[1], 1)

            results  =  []
            for vertex in G:
                if vertex.getColor() == "white":
                    results.append(self.check_validity(vertex))

            if len(results) > 1:
                return False
            else:
                return results[0]


    def check_input(self, n, edges):
        vertices = []
        for entries in edges:
            vertices.append(entries[0])
            vertices.append(entries[1])
        if len(set(vertices)) != n:
            return False
        else:
            return True

    def check_validity(self, start):
        stack = []
        start.setColor("gray")
        stack.append(start)
        while stack != []:
            curr_node = stack.pop()
            for nbr in curr_node.getConnections():
                if nbr.getColor() == "gray":
                    return False
```

```
                if nbr.getColor() == "white":
                    nbr.setColor("gray")
                    stack.append(nbr)
            curr_node.setColor("black")
        return True
```

Course Schedule 2

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

For example:

2, [[1,0]] There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1]

4, [[1,0],[2,0],[3,1],[3,2]] There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].

URL: https://leetcode.com/problems/course-schedule-ii/

```
from queue import Queue
import sys

class Vertex:
    def __init__(self, node):
        self.id = node
        self.adjacent = {}
        # Set distance to infinity for all nodes
        self.distance = sys.maxsize
        # Mark all nodes unvisited
        self.visited = False
        # Mark all nodes color with white
        self.color = 'white'
        # Predecessor
```

```
        self.previous = None
        #indegree of the vertex
        self.indegree = 0

    def addNeighbor(self, neighbor, weight=0):
        self.adjacent[neighbor] = weight

    def getConnections(self):
        return self.adjacent.keys()

    def getVertexID(self):
        return self.id

    def getWeight(self, neighbor):
        return self.adjacent[neighbor]

    def setDistance(self, dist):
        self.distance = dist

    def getDistance(self):
        return self.distance

    def setColor(self, color):
        self.color = color

    def getColor(self):
        return self.color

    def setPrevious(self, prev):
        self.previous = prev

    def setVisited(self):
        self.visited = True

    def setIndegree(self, indegree):
        self.indegree = indegree

    def getIndegree(self):
        return self.indegree
```

```
    def___str__(self):
        return str(self.id) + ' adjacent: ' + str([x.id for x in
 self.adjacent])

class DirectedGraph:
    def___init__(self):
        self.vertDictionary = {}
        self.numVertices = 0

    def___iter__(self):
        return iter(self.vertDictionary.values())

    def  addVertex(self,  node):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(node)
        self.vertDictionary[node] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertDictionary:
            return self.vertDictionary[n]
        else:
            return None

    def  addEdge(self,  frm,  to,  cost=0):
        if frm not in self.vertDictionary:
            self.addVertex(frm)
        if to not in self.vertDictionary:
            self.addVertex(to)

        self.vertDictionary[frm].addNeighbor(self.vertDictionary
[to], cost)
        self.vertDictionary[to].setIndegree(self.vertDictionary[
to].getIndegree() + 1)

    def getVertices(self):
        return self.vertDictionary.keys()

    def setPrevious(self, current):
        self.previous = current
```

```
    def getPrevious(self, current):
        return self.previous


class Solution:
    def___init__(self):
        self.has_cycle = False

    def findOrder(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: List[int]
        """
        if prerequisites == [] and numCourses > 0:
            return [entries for entries in range(numCourses)]
        elif prerequisites == [] and numCourses == 0:
            return []
        else:
            G = DirectedGraph()
            for entries in prerequisites:
                G.addEdge(entries[1], entries[0], 1)

            return self.topsort(G)

    def topsort(self, G):
        if G.getVertices() == []:
            return []
        else:
            topological_list = []
            topological_queue = Queue()
            nodes  =  G.getVertices()
            for node in G:
                if node.getIndegree() == 0:
                    topological_queue.put(node)

            while topological_queue.empty() == False:
                curr_node = topological_queue.get()
                topological_list.append(curr_node.getVertexID())
```

```
                for nbr in curr_node.getConnections():
                    nbr.setIndegree(nbr.getIndegree() - 1)
                    if nbr.getIndegree() == 0:
                        topological_queue.put(nbr)

            if len(topological_list) != len(nodes):
                self.has_cycle = True

            return topological_list


if __name__ == "__main__":

    soln = Solution()
    print(soln.findOrder(4, [[1,0],[2,0],[3,1],[3,2]]))
    print(soln.findOrder(2, [[1,0]]))
    print(soln.findOrder(3, [[1,0]]))
```

Number of Islands

Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

11110 11010 11000 00000 Answer: 1

Example 2:

11000 11000 00100 00011 Answer: 3

URL: https://leetcode.com/problems/number-of-islands/

```
class Solution:
    # @param {boolean[][]} grid a boolean 2D matrix
    # @return {int} an integer
    def numIslands(self, grid):
        if not grid:
            return 0

        row = len(grid)
        col = len(grid[0])
        used = [[False for j in xrange(col)] for i in xrange(row
)]

        count = 0
        for i  in  xrange(row):
            for j in xrange(col):
                if grid[i][j] == '1' and not used[i][j]:
                    self.dfs(grid, used, row, col, i, j)
                    count += 1
        return count

    def dfs(self, grid, used, row, col, x, y):
        if grid[x][y] == '0' or used[x][y]:
            return
        used[x][y] = True

        if x != 0:
            self.dfs(grid, used, row, col, x - 1, y)
        if x != row - 1:
            self.dfs(grid, used, row, col, x + 1, y)
        if y != 0:
            self.dfs(grid, used, row, col, x, y - 1)
        if y != col - 1:
            self.dfs(grid, used, row, col, x, y + 1)
```