1. Next Permutation Medium

A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

For example, for arr = [1,2,3], the following are considered permutations of arr: [1,2,3], [1,3,2], [3,1,2], [2,3,1]. The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

For example, the next permutation of arr = [1,2,3] is [1,3,2]. Similarly, the next permutation of arr = [2,3,1] is [3,1,2]. While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement. Given an array of integers nums, find the next permutation of nums.

The replacement must be in place and use only constant extra memory.

Example 1:

Input: nums = [1,2,3] Output: [1,3,2] Example 2:

Input: nums = [3,2,1] Output: [1,2,3] Example 3:

Input: nums = [1,1,5] Output: [1,5,1]

Constraints:

1 <= nums.length <= 100 0 <= nums[i] <= 100

In [ ]:

```
# """
# 这道题的的难度我觉得理解题意就占了一半。题目的意思是给定一个数，然后将这些数字的位置重新排

# 关键就是刚好是什么意思？比如说原数字是 A，然后将原数字的每位重新排列产生了 B C D E，然后把

# 再比如 123，其他排列有 132，213，231，312，321，从小到大排列就是 123 132 213 231 312 3

# 题目还要求空间复杂度必须是 O (1)
# """

# """
# time complexity:
# 常数阶O(1)：没有循环
# 对数阶O(logN)：while loop, 2^x = N
# 线性阶O(n):for loop
# 线性对数阶O(nlogN): for -> while
# 平方阶O(n²): for -> for loop
# 立方阶O(n³): for-> for-> for loop
# K次方阶O(n^k) k 次 for loop
# 指数阶(2^n)

# space complexity:
# 空间复杂度比较常用的有：
# O(1): no extra space assigned
# O(n): assign new list
```

```
# O(n²):
# """
```

In [ ]:

```
from typing import List

class Solution:
    def next_permutationn(self,arr:List[int]) -> List[int]:

        if len(arr) <1:
            return

        for i in range(len(arr)-2,-1,-1):
            if arr[i] < arr[i+1]:
                for k in range(len(arr)-1, i ,-1):
                    if arr[k] > arr[i]:
                        arr[i] , arr[k] = arr[k] , arr[i]
                        arr[i+1:] = sorted(arr[i+1:])
                        break
                break
            else:
                if i == 0:
                    return arr.sort()

#O(n^2)
#S(1)
```

In [ ]:

```
arr1=[2,3,1]
arr2=[3,2,1]
arr3=[1,1,5]

s=Solution()

s.next_permutation(arr1)
print(arr1)
```

```
[3, 1, 2]
```

1. Pow(x, n) Medium

Implement pow(x, n), which calculates x raised to the power n (i.e., xn).

Example 1:

Input: x = 2.00000, n = 10 Output: 1024.00000 Example 2:

Input: x = 2.10000, n = 3 Output: 9.26100 Example 3:

Input: x = 2.00000, n = -2 Output: 0.25000 Explanation: 2-2 = 1/22 = 1/4 = 0.25

Constraints:

-100.0 < x < 100.0 -231 <= n <= 231-1 -104 <= xn <= 104

In [ ]:

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        elif n < 0:
```

```
            x = 1 / x
            n = -n

        power = x
        multiplications = 1
        while n - multiplications * 2 >= 0:
            power *= power
            multiplications *= 2

        return power * self.myPow(x, n - multiplications)

#o(logn)
#s(1)
```

In [ ]:
```
s = Solution()
s.myPow(2.0,-3)
```

Out[ ]: 0.125

1. Merge Intervals Medium

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

Example 1:

Input: intervals = [[1,3],[2,6],[8,10],[15,18]] Output: [[1,6],[8,10],[15,18]] Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6]. Example 2:

Input: intervals = [[1,4],[4,5]] Output: [[1,5]] Explanation: Intervals [1,4] and [4,5] are considered overlapping.

Constraints:

1 <= intervals.length <= 104 intervals[i].length == 2 0 <= starti <= endi <= 104

In [ ]:
```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:

        #
        intervals.sort(key=lambda x: x[0])

        merged = []
        for interval in intervals:
            # if the list of merged intervals is empty or if the current
            # interval does not overlap with the previous, simply append it.
            print(interval)
            if not merged or merged[-1][1] < interval[0]:
                merged.append(interval)
            else:
            # otherwise, there is overlap, so we merge the current and previous
            # intervals.
                merged[-1][1] = max(merged[-1][1], interval[1])

        return merged


intervals = [[1,3],[2,6],[8,10],[15,18]]
```

```
s=Solution()
s.merge(intervals)
```

```
[1, 3]
[2, 6]
[8, 10]
[15, 18]
```

Out[ ]: `[[1, 6], [8, 10], [15, 18]]`

1. Simplify Path Medium

Given a string path, which is an absolute path (starting with a slash '/') to a file or directory in a Unix-style file system, convert it to the simplified canonical path.

In a Unix-style file system, a period '.' refers to the current directory, a double period '..' refers to the directory up a level, and any multiple consecutive slashes (i.e. '//') are treated as a single slash '/'. For this problem, any other format of periods such as '...' are treated as file/directory names.

The canonical path should have the following format:

The path starts with a single slash '/'. Any two directories are separated by a single slash '/'. The path does not end with a trailing '/'. The path only contains the directories on the path from the root directory to the target file or directory (i.e., no period '.' or double period '..') Return the simplified canonical path.

Example 1:

Input: path = "/home/" Output: "/home" Explanation: Note that there is no trailing slash after the last directory name. Example 2:

Input: path = "/../" Output: "/" Explanation: Going one level up from the root directory is a no-op, as the root level is the highest level you can go. Example 3:

Input: path = "/home//foo/" Output: "/home/foo" Explanation: In the canonical path, multiple consecutive slashes are replaced by a single one.

Constraints:

1 <= path.length <= 3000 path consists of English letters, digits, period '.', slash '/' or '_'. path is a valid absolute Unix path.

In [ ]:

```
class Solution:
    def simplifyPath(self, path:str) ->str:
        stack = []
        for portion in path.split("/"):
            if portion == "..":
                if stack:
                    stack.pop()
            elif portion == "." or not portion:
                continue
            else:
                stack.append(portion)
        final_str = "/" + "/".join(stack)
```

```
        return final_str
```

In [ ]:

```
str_examples ="""
Example 1:

Input: path = "/home/"
Output: "/home"

Example 2:

Input: path = "/../"
Output: "/"

Example 3:

Input: path = "/home//foo/"
Output: "/home/foo"
"""

def get_test_cases(str_examples):
    import re
    params =re.split('Example.+?:',str_examples.replace('\n','').replace(' ',''))[1:
    cases_arg, cases_ans = [], []
    # print(params)
    for param in params:
        # print(param)
        kawargs = {}
        args_,ans = param.split('Output:')
        args_ = args_.replace('Input:','').strip().split(',')
        # ans.replace('"','')
        cases_ans.append(ans.replace('"',''))
        # print(args_)
        for arg in args_:
            key,val = arg.split("=")
            # print(key)
            # print(val)
    #          # val.replace('"','')
            kawargs[key] = val.replace('"','')
        cases_arg.append(kawargs)
    return zip(cases_arg, cases_ans)

def auto_test(test_cases):
    f = getattr(Solution(), dir(Solution())[-1])
    try:
        for kwargs, answer in test_cases:
            print(kwargs,answer)
            assert f(**kwargs) == answer
        return 'Accepted'
    except Exception as e:
        print(e)
        return 'Wrong Answer'

test_case = get_test_cases(str_examples)
auto_test(test_case)
```

```
{'path': '/home/'} /home
{'path': '/../'} /
{'path': '/home//foo/'} /home/foo
```

Out[ ]:

```
'Accepted'
```

1. Copy List with Random Pointer Medium

A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a deep copy of the list. The deep copy should consist of exactly n brand new nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

Return the head of the copied linked list.

The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random_index] where:

val: an integer representing Node.val random_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node. Your code will only be given the head of the original linked list.

Example 1:

Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]] Output: [[7,null],[13,0],[11,4],[10,2],[1,0]] Example 2:

Input: head = [[1,1],[2,1]] Output: [[1,1],[2,1]] Example 3:

Input: head = [[3,null],[3,0],[3,null]] Output: [[3,null],[3,0],[3,null]]

Constraints:

0 <= n <= 1000 -104 <= Node.val <= 104 Node.random is null or is pointing to some node in the linked list.

In [ ]:

```
from typing import Optional

class Node:
    def __init__(self, x: int, next: 'Node' = None, random: 'Node' = None):
        self.val = int(x)
        self.next = next
        self.random = random


class Solution:
    def __init__(self):
        self.visit = {None: None}

    def copyRandomList(self, head: 'Optional[Node]') -> 'Optional[Node]':
        if head in self.visit:
            return self.visit[head]
        node = Node(head.val, None, None)
        self.visit[head] = node
        node.next = self.copyRandomList(head.next)
        node.random = self.copyRandomList(head.random)
```

```
        return node




str_Examples ="""
Example 1:

Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
Example 2:

Input: head = [[1,1],[2,1]]
Output: [[1,1],[2,1]]
Example 3:

Input: head = [[3,null],[3,0],[3,null]]
Output: [[3,null],[3,0],[3,null]]

"""
```

In [ ]:

```
# class LinkedList:
#     def __init__(self):
#         self.head = None

#     def print_in_order(self):
#         '''
#         print linked list element starting from the head, walking
#         until the end of the list
#         '''
#         curr_node = self.head
#         print(curr_node.val)
#         while curr_node.next is not None:
#             print(curr_node.next.val)
#             curr_node = curr_node.next


# class LinkedListNode:
#     def __init__(self, val=None, next=None):
#         '''
#         singly linked list individual node
#         '''
#         self.val = val
#         self.next = next

# class SinglyLinkedList(LinkedList):
#     '''
#     singly linked list
#     '''

#     def populate_from_set(self, set_to_use: set):
#         '''
#         iteratively populate a singly linked list from a set of values
#         '''
#         if len(set_to_use) == 0:
#             raise ValueError('Cannot start a singly linked list from an empty set.

#         # then iterate through to the end of the set, linking each node to the nex
#         node_prev = None
#         for set_i in range(len(set_to_use)):

#             # set the current node
```

```
#             node_curr = LinkedListNode(val = set_to_use[set_i])

#             # set the head of the SLL on the first pass through the set
#             if set_i == 0:
#                 self.head = node_curr
#             # otherwise we link the previous node to the current node
#             else:
#                 node_prev.next = node_curr

#             # then set the previous node to the current node for the next iteratio
#             node_prev = node_curr

# class DoublyLinkedListNode(LinkedListNode):
#     def __init__(self, val=None, prev=None, next=None):
#         self.val = val
#         self.prev=prev
#         self.next=next

# class DoublyLinkedList(LinkedList):

#     def __init__(self, tail=None):
#         self.tail=tail

#     def populate_from_set(self, set_to_use: set):
#         '''
#         iteratively populate the doubly linked list from a set of values
#         '''
#         if len(set_to_use) == 0:
#             raise ValueError('Cannot populate a doubly linked list from an empty s

#         prev_node = None
#         for set_i in range(len(set_to_use)):
#             curr_node = DoublyLinkedListNode(val=set_to_use[set_i])

#             # if we are on the first element, we assign the head
#             if set_i == 0:
#                 self.head = curr_node
#             # otherwise we assign a next value to the previous and a
#             # previous to the current
#             else:
#                 prev_node.next = curr_node
#                 curr_node.prev = prev_node

#             # if we are on the last set element, we assign a tail value
#             if set_i == len(set_to_use)-1:
#                 self.tail = curr_node

#             prev_node = curr_node

#     def print_in_reverse_order(self):
#         '''
#         print all linked list elements starting from the tail
#         and walking along until you hit the head
#         '''
#         curr_node = self.tail
#         print(curr_node.val)
#         while curr_node.prev is not None:
#             print(curr_node.prev.val)
#             curr_node = curr_node.prev

# def main():

#     days = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun')
#     print('SINGLE:')
```

```
#     sll = SinglyLinkedList()
#     sll.populate_from_set(days)
#     sll.print_in_order()

#     print('\n\nDOUBLE:')
#     dll = DoublyLinkedList()
#     dll.populate_from_set(days)
#     print('\n\nforward:')
#     dll.print_in_order()
#     print('\n\nreverse:')
#     dll.print_in_reverse_order()

# if __name__=='__main__':
#     main()
```

1. Find Peak Element Medium

A peak element is an element that is strictly greater than its neighbors.

Given an integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that nums[-1] = nums[n] = -∞.

You must write an algorithm that runs in O(log n) time.

Example 1:

Input: nums = [1,2,3,1] Output: 2 Explanation: 3 is a peak element and your function should return the index number 2. Example 2:

Input: nums = [1,2,1,3,5,6,4] Output: 5 Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

In [ ]:

```
class Solution:
    # def findPeakElement(self,nums:List[int]) ->int:
    #     for i in range(0,len(nums)-1,1):
    #         if nums[i] > nums[i+1]:
    #             return i

    #     return len(nums)-1
#这个只会return第一个peak，根据题目return任意peak，所以满足。
#[3,2,4,1]


    def findPeakElement(self,nums:List[int]) ->int:
        l = 0
        r = len(nums)-1
        while l < r:
            mid = int((l + r)/2)
            if nums[mid] > nums[mid+1]:
                r = mid
            else:
                l = mid + 1

        return l
#binary search,找到中间位置的元素，然后查看向后一位是递增还是递减
#递增，就证明peak在后面这部分，所以 l = mid+1
#递减，就证明peak在前面这部分，所以 r = mid
```

```
str_examples ="""
Example 1:

Input: nums = [1,2,3,1]
Output: 2

Example 2:

Input: nums = [1,2,1,3,5,6,4]
Output: 5

"""
```

In [ ]:
```
nums=[1,2,3,1]
s=Solution()
s.findPeakElement(nums)
```

Out[ ]: 2

1. Binary Search Tree Iterator Medium

Implement the BSTIterator class that represents an iterator over the in-order traversal of a binary search tree (BST):

BSTIterator(TreeNode root) Initializes an object of the BSTIterator class. The root of the BST is given as part of the constructor. The pointer should be initialized to a non-existent number smaller than any element in the BST. boolean hasNext() Returns true if there exists a number in the traversal to the right of the pointer, otherwise returns false. int next() Moves the pointer to the right, then returns the number at the pointer. Notice that by initializing the pointer to a non-existent smallest number, the first call to next() will return the smallest element in the BST.

You may assume that next() calls will always be valid. That is, there will be at least a next number in the in-order traversal when next() is called.

Example 1:

Input ["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "next", "hasNext"] [[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], [], [], []] Output [null, 3, 7, true, 9, true, 15, true, 20, false]

Explanation BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]); bSTIterator.next(); // return 3 bSTIterator.next(); // return 7 bSTIterator.hasNext(); // return True bSTIterator.next(); // return 9 bSTIterator.hasNext(); // return True bSTIterator.next(); // return 15 bSTIterator.hasNext(); // return True bSTIterator.next(); // return 20 bSTIterator.hasNext(); // return False

Constraints:

The number of nodes in the tree is in the range [1, 105]. 0 <= Node.val <= 106 At most 105 calls will be made to hasNext, and next.

Follow up:

Could you implement next() and hasNext() to run in average O(1) time and use O(h) memory, where h is the height of the tree?

In [ ]:
```
from typing import Optional


class TreeNode:
    def __init__(self,val=0,left=None,right=None):
        self.val=val
        self.left=left
        self.right=right


class BSTIterator:
    def __init__(self, root:Optional[TreeNode]):
        self.stack = []
        while root:
            self.stack.append(root)
            root = root.left

    def next(self) -> int:
        temp = self.stack.pop()
        x = temp.right
        while x:
            self.stack.append(x)
            x = x.left
        return temp.val

    def hasNext(self) -> bool:
        return len(self.stack) > 0


def TreeBuilder(val):
    def create(it):
        value = next(it)
        return None if value is None else TreeNode(value)

    if not val:
        return None

    it = iter(val)
    root = TreeNode(next(it))
    nextlevel = [root]

    try:
        while nextlevel:
            level = nextlevel
            nextlevel = []
            for node in level:
                if node:
                    node.left = create(it)
                    node.right = create(it)
                    nextlevel +=[node.left,node.right]
    except StopIteration:
        return root
    raise ValueError("Invalid Error")
```

In [ ]:
```
root = [7,3,15,None,None,9,20]
obj = BSTIterator(TreeBuilder(root))

"""
```

```
Input
["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "ne
[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], [], [], []]
Output
[null, 3, 7, true, 9, true, 15, true, 20, false]

"""
```

Out[ ]:
```
'\nInput\n["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "has
Next", "next", "hasNext"]\n[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [],
[], [], []]\nOutput\n[null, 3, 7, true, 9, true, 15, true, 20, false]\n'
```

1. Binary Tree Right Side View Medium

Given the root of 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 1:

Input: root = [1,2,3,null,5,null,4] Output: [1,3,4] Example 2:

Input: root = [1,null,3] Output: [1,3] Example 3:

Input: root = [] Output: []

Constraints:

The number of nodes in the tree is in the range [0, 100]. -100 <= Node.val <= 100

In [ ]:
```
from typing import Optional
from collections import deque


class TreeNode:
    def __init__(self,val=0,left=None,right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution:
    def rightSideView(self,root:Optional[TreeNode]) -> List[int]:
        if root is None:
            return []

        next_level = deque([root,])
        rightside = []

        while next_level:
            # prepare for the next level
            curr_level = next_level
            next_level = deque()

            while curr_level:
                node = curr_level.popleft()

                # add child nodes of the current level
                # in the queue for the next level
                if node.left:
                    next_level.append(node.left)
```

```
                if node.right:
                    next_level.append(node.right)

            # The current level is finished.
            # Its last element is the rightmost one.
            rightside.append(node.val)

        return rightside


def TreeBuilder(val):
    def create(it):
        value = next(it)
        return None if value is None else TreeNode(value)

    if not val:
        return None

    it = iter(val)
    root = TreeNode(next(it))
    nextlevel = [root]

    try:
        while nextlevel:
            level = nextlevel
            nextlevel = []
            for node in level:
                if node:
                    node.left = create(it)
                    node.right = create(it)
                    nextlevel +=[node.left,node.right]
    except StopIteration:
        return root
    raise ValueError("Invalid Input")
```

In [ ]:
```
root=[1,2,3,None,5,None,4]
s=Solution()
s.rightSideView(TreeBuilder(root))
```

Out[ ]: [1, 3, 4]

1. Sum Root to Leaf Numbers

You are given the root of a binary tree containing digits from 0 to 9 only.

Each root-to-leaf path in the tree represents a number.

For example, the root-to-leaf path 1 -> 2 -> 3 represents the number 123. Return the total sum of all root-to-leaf numbers. Test cases are generated so that the answer will fit in a 32-bit integer.

A leaf node is a node with no children.

Example 1:

Input: root = [1,2,3] Output: 25 Explanation: The root-to-leaf path 1->2 represents the number 12. The root-to-leaf path 1->3 represents the number 13. Therefore, sum = 12 + 13 = 25.
Example 2:

Input: root = [4,9,0,5,1] Output: 1026 Explanation: The root-to-leaf path 4->9->5 represents the number 495. The root-to-leaf path 4->9->1 represents the number 491. The root-to-leaf path 4->0 represents the number 40. Therefore, sum = 495 + 491 + 40 = 1026.

Constraints:

The number of nodes in the tree is in the range [1, 1000]. 0 <= Node.val <= 9 The depth of the tree will not exceed 10.

In [ ]:

```
from typing import Optional

class TreeNode:
    def __init__(self,val=0,left=None,right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution:
    def sumNumbers(self, root: Optional[TreeNode]) -> int:
        def find(root, n):
            if root is None:
                return 0
            cur = n * 10 + root.val
            if root.left is None and root.right is None:
                return cur
            return find(root.left, cur) + find(root.right, cur)

        return find(root, 0)


def TreeBuilder(val):
    def create(it):
        value = next(it)
        return None if value is None else TreeNode(value)

    if not val:
        return None

    it = iter(val)
    root = TreeNode(next(it))
    nextlevel = [root]

    try:
        while nextlevel:
            level = nextlevel
            nextlevel = []
            for node in level:
                if node:
                    node.left = create(it)
                    node.right = create(it)
                    nextlevel +=[node.left,node.right]
    except StopIteration:
        return root
    raise ValueError("Invalid Input")
```

1. Kth Largest Element in an Array

Given an integer array nums and an integer k, return the kth largest element in the array.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

Example 1:

Input: nums = [3,2,1,5,6,4], k = 2 Output: 5 Example 2:

Input: nums = [3,2,3,1,2,4,5,5,6], k = 4 Output: 4

Constraints:

1 <= k <= nums.length <= 104 -104 <= nums[i] <= 104

In [ ]:

```
from typing import List

# class Solution:
#     def findKthLargest(self, nums: List[int], k: int) -> int:
#         return sorted(nums)[-k]

#heapq 堆

# import heapq

# class Solution:
#     def findKthLargest(self, nums: List[int], k: int) -> int:
#         return heapq.nlargest(k, nums)[-1]

import random

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        def partition(start, end):
            ran = random.randint(start, end)
            pivot = end
            nums[pivot], nums[ran] = nums[ran], nums[pivot]

            border = start
            for cur in range(start, end):
                if nums[cur] >= nums[pivot]:
                    nums[cur], nums[border] = nums[border], nums[cur]
                    border += 1

            nums[border], nums[pivot] = nums[pivot], nums[border]
            return border

        def quick_select(start, end, k_largest):
            res = None
            while start <= end:
                p = partition(start, end)
                if p == k_largest:
                    res = nums[k_largest]
                    break
                elif p > k_largest:
                    end = p - 1
                else:
                    start = p + 1
            return res

        if k > len(nums):
            return None

        return quick_select(0, len(nums)-1, k-1)
```

In [ ]:
```
nums=[3,2,3,1,2,4,5,5,6]
k = 4

s=Solution()
s.findKthLargest(nums,k)
```

Out[ ]: 4

1. Basic Calculator II Medium

Given a string s which represents an expression, evaluate this expression and return its value.

The integer division should truncate toward zero.

You may assume that the given expression is always valid. All intermediate results will be in the range of [-231, 231 - 1].

Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

In [ ]:
```
class Solution:
    def calculate(self, s: str) -> int:
        stack, cur, op = [], 0, '+'
        for c in s + '+':
            if c == " ":
                continue
            #这里判断是不是两位数或者以上
            elif c.isdigit():
                cur = (cur * 10) + int(c)
            else:
                if op == '-':
                    stack.append(-cur)
                elif op == '+':
                    stack.append(cur)
                elif op == '*':
                    stack.append(stack.pop() * cur)
                elif op == '/':
                    stack.append(int(stack.pop() / cur))
                cur, op = 0, c
        return sum(stack)

s1="3+2*2"
s2=" 3/2 "
s3=" 3+5 / 2 "

s = Solution()
print(s.calculate(s1))
print(s.calculate(s2))
print(s.calculate(s3))
```

```
7
1
5
```

In [ ]:
```
class Solution:
    def calculate(self, s: str) -> int:
        inner, outer, result, opt = 0, 0, 0, '+'
        for c in s + '+':
```

```
            if c == ' ': continue
            if c.isdigit():
                inner = 10 * inner + int(c)
                continue
            if opt == '+':
                result += outer
                outer = inner
            elif opt == '-':
                result += outer
                outer = -inner
            elif opt == '*':
                outer = outer * inner
            elif opt == '/':
                outer = int(outer / inner)
            inner, opt = 0, c
        return result + outer

s1="3+2*2"
s2=" 3/2 "
s3=" 3+5 / 2 "

s = Solution()
print(s.calculate(s1))
print(s.calculate(s2))
print(s.calculate(s3))
```

```
7
1
5
```

1. Lowest Common Ancestor of a Binary Tree Medium

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 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).”

Example 1:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1 Output: 3 Explanation: The LCA of nodes 5 and 1 is 3. Example 2:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4 Output: 5 Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition. Example 3:

Input: root = [1,2], p = 1, q = 2 Output: 1

In [ ]:

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -
        if root == None or root == p or root == q:
            return root

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

        if Left_LCA == None:
            return Right_LCA

        if Right_LCA == None:
```

```
            return Left_LCA

        return root
```

1. Group Shifted Strings

We can shift a string by shifting each of its letters to its successive letter.

For example, "abc" can be shifted to be "bcd". We can keep shifting the string to form a sequence.

For example, we can keep shifting "abc" to form the sequence: "abc" -> "bcd" -> ... -> "xyz". Given an array of strings strings, group all strings[i] that belong to the same shifting sequence. You may return the answer in any order.

Example 1:

Input: strings = ["abc","bcd","acef","xyz","az","ba","a","z"] Output: [["acef"],["a","z"], ["abc","bcd","xyz"],["az","ba"]] Example 2:

Input: strings = ["a"] Output: [["a"]]

In [ ]:
```
import collections

class Solution:
    def groupStrings(self, strings: List[str]) -> List[List[str]]:

        def shift_letter(letter: str, shift: int):
            return chr((ord(letter) - shift) % 26 + ord('a'))

        # Create a hash value
        def get_hash(string: str):
            # Calculate the number of shifts to make the first character to be 'a'
            shift = ord(string[0])
            return ''.join(shift_letter(letter, shift) for letter in string)

        # Create a hash_value (hashKey) for each string and append the string
        # to the list of hash values i.e. mapHashToList["abc"] = ["abc", "bcd"]
        groups = collections.defaultdict(list)
        for string in strings:
            hash_key = get_hash(string)
            groups[hash_key].append(string)

        # Return a list of all of the grouped strings
        return list(groups.values())
```

In [ ]:
```
input = ["abc","bcd","acef","xyz","az","ba","a","z"]

s=Solution()
s.groupStrings(input)
```

Out[ ]:
```
[['abc', 'bcd', 'xyz'], ['acef'], ['az', 'ba'], ['a', 'z']]
```

In [ ]:
```
def groupStrings(strings: List[str]) -> List[List[str]]:
        hashmap = {}
        for s in strings:
```

```
                key = ()
                for i in range(len(s) - 1):
                        circular_difference = 26 + ord(s[i+1]) - ord(s[i])
                        key += (circular_difference % 26,)
                hashmap[key] = hashmap.get(key, []) + [s]
        return list(hashmap.values())
```

In [ ]:
```
input = ["abc","bcd","acef","xyz","az","ba","a","z"]
groupStrings(input)
```

Out[ ]:
```
[['abc', 'bcd', 'xyz'], ['acef'], ['az', 'ba'], ['a', 'z']]
```

1. Binary Tree Vertical Order Traversal

Given the root of a binary tree, return the vertical order traversal of its nodes' values. (i.e., from top to bottom, column by column).

If two nodes are in the same row and column, the order should be from left to right.

Example 1:

Input: root = [3,9,20,null,null,15,7] Output: [[9],[3,15],[20],[7]] Example 2:

Input: root = [3,9,8,4,0,1,7] Output: [[4],[9],[3,0,1],[8],[7]] Example 3:

Input: root = [3,9,8,4,0,1,7,null,null,null,2,5] Output: [[4],[9,5],[3,0,1],[8,2],[7]]

In [ ]:
```
from collections import defaultdict

# TreeNode

class Solution:
    def verticalOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []

        columnTable = defaultdict(list)
        min_column = max_column = 0
        queue = deque([(root, 0)])

        while queue:
            node, column = queue.popleft()

            if node is not None:
                columnTable[column].append(node.val)
                min_column = min(min_column, column)
                max_column = max(max_column, column)

                queue.append((node.left, column - 1))
                queue.append((node.right, column + 1))

        return [columnTable[x] for x in range(min_column, max_column + 1)]

#TreeBuilder
```

1. Top K Frequent Elements

Given an integer array nums and an integer k, return the k most frequent elements. You may return the answer in any order.

Example 1:

Input: nums = [1,1,1,2,2,3], k = 2 Output: [1,2] Example 2:

Input: nums = [1], k = 1 Output: [1]

In [ ]:
```
from collections import Counter
import heapq
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # O(1) time
        if k == len(nums):
            return nums

        # 1. build hash map : character and how often it appears
        # O(N) time
        count = Counter(nums)
        # 2-3. build heap of top k frequent elements and
        # convert it into an output array
        # O(N log k) time
        return heapq.nlargest(k, count.keys(), key=count.get)
```

In [ ]:
```
from collections import Counter
class Solution:
    def topKFrequent(self, nums, k):
        bucket = [[] for _ in range(len(nums) + 1)]
        Count = Counter(nums).items()
        for num, freq in Count: bucket[freq].append(num)
        flat_list = [item for sublist in bucket for item in sublist]
        return flat_list[::-1][:k]
```

In [ ]:
```
from collections import Counter
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        count = Counter(nums)
        unique = list(count.keys())

        def partition(left, right, pivot_index) -> int:
            pivot_frequency = count[unique[pivot_index]]
            # 1. move pivot to end
            unique[pivot_index], unique[right] = unique[right], unique[pivot_index]

            # 2. move all less frequent elements to the left
            store_index = left
            for i in range(left, right):
                if count[unique[i]] < pivot_frequency:
                    unique[store_index], unique[i] = unique[i], unique[store_index]
                    store_index += 1

            # 3. move pivot to its final place
            unique[right], unique[store_index] = unique[store_index], unique[right]

            return store_index

        def quickselect(left, right, k_smallest) -> None:
            """
```

```
            Sort a list within left..right till kth less frequent element
            takes its place.
            """
            # base case: the list contains only one element
            if left == right:
                return

            # select a random pivot_index
            pivot_index = random.randint(left, right)

            # find the pivot position in a sorted list
            pivot_index = partition(left, right, pivot_index)

            # if the pivot is in its final sorted position
            if k_smallest == pivot_index:
                 return
            # go left
            elif k_smallest < pivot_index:
                quickselect(left, pivot_index - 1, k_smallest)
            # go right
            else:
                quickselect(pivot_index + 1, right, k_smallest)

        n = len(unique)
        # kth top frequent element is (n - k)th less frequent.
        # Do a partial sort: from less frequent to the most frequent, till
        # (n - k)th less frequent element takes its place (n - k) in a sorted array.
        # All element on the left are less frequent.
        # All the elements on the right are more frequent.
        quickselect(0, n - 1, n - k)
        # Return top k frequent elements
        return unique[n - k:]
```

1. Nested List Weight Sum Medium

You are given a nested list of integers nestedList. Each element is either an integer or a list whose elements may also be integers or other lists.

The depth of an integer is the number of lists that it is inside of. For example, the nested list [1,[2,2],[[3],2],1] has each integer's value set to its depth.

Return the sum of each integer in nestedList multiplied by its depth.

Example 1:

Input: nestedList = [[1,1],2,[1,1]] Output: 10 Explanation: Four 1's at depth 2, one 2 at depth 1. 1*2* + *12* + *21* + *12* + 1*2 = 10. Example 2:

Input: nestedList = [1,[4,[6]]] Output: 27 Explanation: One 1 at depth 1, one 4 at depth 2, and one 6 at depth 3. 1*1* + *42* + 6*3 = 27. Example 3:

Input: nestedList = [0] Output: 0

In [ ]:

```
#DFS

class Solution:
    def depthSum(self, nestedList: List[NestedInteger]) -> int:

        def dfs(nested_list, depth):
```

```
            total = 0
            for nested in nested_list:
                if nested.isInteger():
                    total += nested.getInteger() * depth
                else:
                    total += dfs(nested.getList(), depth + 1)
            return total

        return dfs(nestedList, 1)
```

In [ ]:

```
#BFS

class Solution:
    def depthSum(self, nestedList: List[NestedInteger]) -> int:
        queue = deque(nestedList)

        depth = 1
        total = 0

        while len(queue) > 0:
            for i in range(len(queue)):
                nested = queue.pop()
                if nested.isInteger():
                    total += nested.getInteger() * depth
                else:
                    queue.extendleft(nested.getList())
            depth += 1

        return total
```

1. Random Pick Index Medium

Given an integer array nums with possible duplicates, randomly output the index of a given target number. You can assume that the given target number must exist in the array.

Implement the Solution class:

Solution(int[] nums) Initializes the object with the array nums. int pick(int target) Picks a random index i from nums where nums[i] == target. If there are multiple valid i's, then each index should have an equal probability of returning.

Example 1:

Input ["Solution", "pick", "pick", "pick"] [[[1, 2, 3, 3, 3]], [3], [1], [3]] Output [null, 4, 0, 2]

Explanation Solution solution = new Solution([1, 2, 3, 3, 3]); solution.pick(3); // It should return either index 2, 3, or 4 randomly. Each index should have equal probability of returning. solution.pick(1); // It should return 0. Since in the array only nums[0] is equal to 1. solution.pick(3); // It should return either index 2, 3, or 4 randomly. Each index should have equal probability of returning.

In [ ]:

```
from typing import List

class Solution(object):
    def __init__(self,nums:List[int]):
        #create a dict
```

```
        self.dict = collections.defaultdict(list)
        #{num:[i]}, {1: [0], 2: [1], 3: [2], 4: [3], 5: [4], 6: [5], 7: [6]}
        for i , num in enumerate(nums):
            self.dict[num].append(i)

    def pick(self,target):
        return random.choice(self.dict[target])
```

In [ ]:

```
class Solution:
    def __init__(self,nums:List[int]):
        self.dict = {}
        for i in range(len(nums)):
            if nums[i] not in self.dict:
                self.dict[nums[i]] = [i]
            else:
                self.dict[nums[i]].append(i)

    def pick(self,target):
        key = self.dict[target]
        idx = random.randrange(len(key))

        return key[idx]
```

1. Diagonal Traverse Medium

Given an m x n matrix mat, return an array of all the elements of the array in a diagonal order.

Example 1:

Input: mat = [[1,2,3],[4,5,6],[7,8,9]] Output: [1,2,4,7,5,3,6,8,9] Example 2:

Input: mat = [[1,2],[3,4]] Output: [1,2,3,4]

In [ ]:

```
class Solution:
    def findDiagonalOrder(self,matrix:List[List[int]]) -> List[int]:
        if not matrix or not matrix[0]:
            return []

        N , M = len(matrix) , len(matrix[0])
        row , col = 0 , 0
        direction = 1 # going upward
        res = []

        while row < N and col < M:
            res.append(matrix[row][col])

            # the current direction.[i, j] -> [i - 1, j + 1] if
            # going up and [i, j] -> [i + 1][j - 1] if going down.
            if direction == 1:
                new_row = row -1
                new_col = col +1
            else:
                new_row = row +1
                new_col = col -1

            if new_row < 0 or new_row == N or new_col < 0 or new_col == M:
                if direction:
                    if col == M-1: row += 1
                    if col < M-1: col +=1
                else:
```

```
                    if row == N-1: col +=1
                    if row < N-1: row +=1

                direction = 1 - direction
            else:
                row = new_row
                col = new_col

        return res
```

1. Continuous Subarray Sum Medium

Given an integer array nums and an integer k, return true if nums has a continuous subarray of size at least two whose elements sum up to a multiple of k, or false otherwise.

An integer x is a multiple of k if there exists an integer n such that x = n * k. 0 is always a multiple of k.

Example 1:

Input: nums = [23,2,4,6,7], k = 6 Output: true Explanation: [2, 4] is a continuous subarray of size 2 whose elements sum up to 6. Example 2:

Input: nums = [23,2,6,4,7], k = 6 Output: true Explanation: [23, 2, 6, 4, 7] is an continuous subarray of size 5 whose elements sum up to 42. 42 is a multiple of 6 because 42 = 7 * 6 and 7 is an integer. Example 3:

Input: nums = [23,2,6,4,7], k = 13 Output: false

In [ ]:

```
class Solution():
    def checkSubarraySum(self, nums: List[int], k: int) -> bool:
        #create a dict, key=0,val=-1
        dic = {0:-1}
        summ = 0
        #since enumerate i is index, n is val
        for i, n in enumerate(nums):
            if k != 0:
                summ = (summ + n) % k
            else:
                summ += n

            if summ not in dic:
                dic[summ] = i
            else:
                if i - dic[summ] >= 2:
                    return True
        return False
```

1. Subarray Sum Equals K Medium

Given an array of integers nums and an integer k, return the total number of subarrays whose sum equals to k.

A subarray is a contiguous non-empty sequence of elements within an array.

Example 1:

Input: nums = [1,1,1], k = 2 Output: 2 Example 2:

Input: nums = [1,2,3], k = 3 Output: 2

In [ ]:

```
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        ans, n = 0, len(nums)
        preSum = 0
        dic = {}
        #{0:1}
        dic[0] = 1
        #i represents val in list
        for i in nums:
            preSum += i
            #find the key value in a dict
            if preSum-k in dic:
                ans+=dic[preSum-k]
                #dic.get(key return key value, if key not exist return value)
            dic[preSum] = dic.get(preSum,0) + 1
        return ans

#[1,1,1]   k =2
#ans =0 , n =0, presum =0, dic = {0:1}
#i =1 , presum = 1, presum - k = 1 -2 = -1, not in dict, dic[1] = 1 , dic = {0:1,1:1
#i = 1 , presum = 2, presum - k = 2-2 =0, in dict, ans = dic[0] = 1 , dic = {0:1,1:1
#i = 1 , presum = 3, presum - k = 3-2 =1,  in dict, ans = 2, dic = {0:1, 1:1, 2:1, 3
# ans = 2

#[1,2,3]   k =3
#ans =0 , n =0, presum =0, dic = {0:1}
#i =1 , presum = 1, presum - k = 1 -3 = -2, not in dict, dic[1] = 1 , dic = {0:1,1:1
#i = 2 , presum = 3, presum - k = 3-3 =0, in dict, ans = dic[0] = 1 , dic = {0:1,1:1
#i = 3 , presum = 6, presum - k = 6-3 =3,  in dict, ans = 1+dic[3]=2, dic = {0:1, 1:
# ans = 2
```

1. Exclusive Time of Functions Medium

On a single-threaded CPU, we execute a program containing n functions. Each function has a unique ID between 0 and n-1.

Function calls are stored in a call stack: when a function call starts, its ID is pushed onto the stack, and when a function call ends, its ID is popped off the stack. The function whose ID is at the top of the stack is the current function being executed. Each time a function starts or ends, we write a log with the ID, whether it started or ended, and the timestamp.

You are given a list logs, where logs[i] represents the ith log message formatted as a string "{function_id}:{"start" | "end"}:{timestamp}". For example, "0:start:3" means a function call with function ID 0 started at the beginning of timestamp 3, and "1:end:2" means a function call with function ID 1 ended at the end of timestamp 2. Note that a function can be called multiple times, possibly recursively.

A function's exclusive time is the sum of execution times for all function calls in the program. For example, if a function is called twice, one call executing for 2 time units and another call executing for 1 time unit, the exclusive time is 2 + 1 = 3.

Return the exclusive time of each function in an array, where the value at the ith index represents the exclusive time for the function with ID i.

Example 1:

Input: n = 2, logs = ["0:start:0","1:start:2","1:end:5","0:end:6"] Output: [3,4] Explanation: Function 0 starts at the beginning of time 0, then it executes 2 for units of time and reaches the end of time 1. Function 1 starts at the beginning of time 2, executes for 4 units of time, and ends at the end of time 5. Function 0 resumes execution at the beginning of time 6 and executes for 1 unit of time. So function 0 spends 2 + 1 = 3 units of total time executing, and function 1 spends 4 units of total time executing. Example 2:

Input: n = 1, logs = ["0:start:0","0:start:2","0:end:5","0:start:6","0:end:6","0:end:7"] Output: [8] Explanation: Function 0 starts at the beginning of time 0, executes for 2 units of time, and recursively calls itself. Function 0 (recursive call) starts at the beginning of time 2 and executes for 4 units of time. Function 0 (initial call) resumes execution then immediately calls itself again. Function 0 (2nd recursive call) starts at the beginning of time 6 and executes for 1 unit of time. Function 0 (initial call) resumes execution at the beginning of time 7 and executes for 1 unit of time. So function 0 spends 2 + 4 + 1 + 1 = 8 units of total time executing. Example 3:

Input: n = 2, logs = ["0:start:0","0:start:2","0:end:5","1:start:6","1:end:6","0:end:7"] Output: [7,1] Explanation: Function 0 starts at the beginning of time 0, executes for 2 units of time, and recursively calls itself. Function 0 (recursive call) starts at the beginning of time 2 and executes for 4 units of time. Function 0 (initial call) resumes execution then immediately calls function 1. Function 1 starts at the beginning of time 6, executes 1 unit of time, and ends at the end of time 6. Function 0 resumes execution at the beginning of time 6 and executes for 2 units of time. So function 0 spends 2 + 4 + 1 = 7 units of total time executing, and function 1 spends 1 unit of total time executing.

In [ ]:

```python
from typing import List


class Solution:
    def exclusiveTime(self, n: int, logs: List[str]) -> List[int]:
        res = [0] * n
        stack = list()
        last = 0
        for s in logs:
            t = s.split(":")
            # t[0]表示函数id, t[1], t[2]表示当前是start还是end字符串, t[2]表示对应的开始
            tid, marks, ts = int(t[0])  , t[1], int(t[2])
            # 判断当前的标识符
            if marks == "start":
                # 判断当前栈中是否存在元素
                if stack: res[stack[-1]] += ts - last
                stack.append(tid)
                last = ts
            else:
                res[stack[-1]] += ts - last + 1
                stack.pop()
                last = ts + 1
        return res
```

1. Maximum Swap Medium

You are given an integer num. You can swap two digits at most once to get the maximum valued number.

Return the maximum valued number you can get.

Example 1:

Input: num = 2736 Output: 7236 Explanation: Swap the number 2 and the number 7. Example 2:

Input: num = 9973 Output: 9973 Explanation: No swap.

In [ ]:
```
class Solution:
    def maximumSwap(self, num: int) -> int:
        s = list(str(num))
        n = len(s)
        for i in range(n-1):                                    # find index where s[i]
            if s[i] < s[i+1]: break
        else: return num                                        # if nothing find, retur
        max_idx, max_val = i+1, s[i+1]                          # keep going right, find
        for j in range(i+1, n):
            if max_val <= s[j]: max_idx, max_val = j, s[j]
        left_idx = i                                            # going right from i, fi
        for j in range(i, -1, -1):
            if s[j] < max_val: left_idx = j
        s[max_idx], s[left_idx] = s[left_idx], s[max_idx]   # swap maximum after i a
        return int(''.join(s))                                  # re-create the integer
```

1. Accounts Merge Medium

Given a list of accounts where each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.

Example 1:

Input: accounts = [["John","johnsmith@mail.com","john_newyork@mail.com"], ["John","johnsmith@mail.com","john00@mail.com"],["Mary","mary@mail.com"], ["John","johnnybravo@mail.com"]] Output: [["John","john00@mail.com","john_newyork@mail.com","johnsmith@mail.com"], ["Mary","mary@mail.com"],["John","johnnybravo@mail.com"]] Explanation: The first and second John's are the same person as they have the common email "johnsmith@mail.com". The third

John and Mary are different people as none of their email addresses are used by other accounts. We could return these lists in any order, for example the answer [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'], ['John', 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com']] would still be accepted. Example 2:

Input: accounts = [["Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"], ["Kevin","Kevin3@m.co","Kevin5@m.co","Kevin0@m.co"], ["Ethan","Ethan5@m.co","Ethan4@m.co","Ethan0@m.co"], ["Hanzo","Hanzo3@m.co","Hanzo1@m.co","Hanzo0@m.co"], ["Fern","Fern5@m.co","Fern1@m.co","Fern0@m.co"]] Output: [["Ethan","Ethan0@m.co","Ethan4@m.co","Ethan5@m.co"], ["Gabe","Gabe0@m.co","Gabe1@m.co","Gabe3@m.co"], ["Hanzo","Hanzo0@m.co","Hanzo1@m.co","Hanzo3@m.co"], ["Kevin","Kevin0@m.co","Kevin3@m.co","Kevin5@m.co"], ["Fern","Fern0@m.co","Fern1@m.co","Fern5@m.co"]]

In [ ]:

```python
class Solution:
    def accountsMerge(self, accounts):
        hashmap = {}
        graph = collections.defaultdict(set)
        for account in accounts:
            name = account[0]
            for email in account[1:]:
                hashmap[email] = name
                graph[account[1]].add(email)
                graph[email].add(account[1])
        visited = set()
        res = []
        for node in hashmap:
            if node not in visited:
                tmplist = []
                self.dfs(graph, node, visited, tmplist)
                res.append([hashmap[node]] + sorted(tmplist))
        return res

    def dfs(self, graph, node, visited, tmplist):
        visited.add(node)
        tmplist.append(node)
        for nei in graph[node]:
            if nei not in visited:
                self.dfs(graph, nei, visited, tmplist)
```

1. Convert Binary Search Tree to Sorted Doubly Linked List Medium

Convert a Binary Search Tree to a sorted Circular Doubly-Linked List in place.

You can think of the left and right pointers as synonymous to the predecessor and successor pointers in a doubly-linked list. For a circular doubly linked list, the predecessor of the first element is the last element, and the successor of the last element is the first element.

We want to do the transformation in place. After the transformation, the left pointer of the tree node should point to its predecessor, and the right pointer should point to its successor. You should return the pointer to the smallest element of the linked list.

Example 1:

Input: root = [4,2,5,1,3]

Output: [1,2,3,4,5]

Explanation: The figure below shows the transformed BST. The solid line indicates the successor relationship, while the dashed line means the predecessor relationship.

Example 2:

Input: root = [2,1,3] Output: [1,2,3]

In [ ]:

```
"""
# Definition for a Node.
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
"""
#default DFS, inorder traversal, root - left - right


class Solution:
    def treeToDoublyList(self, root: 'Node') -> 'Node':
        def helper(node):
            """
            Performs standard inorder traversal:
            left -> node -> right
            and links all nodes into DLL
            """
            nonlocal last, first
            if node:
                # left
                helper(node.left)
                # node
                if last:
                    # link the previous node (last)
                    # with the current one (node)
                    last.right = node
                    node.left = last
                else:
                    # keep the smallest node
                    # to close DLL later on
                    first = node
                last = node
                # right
                helper(node.right)

        if not root:
            return None

        # the smallest (first) and the largest (last) nodes
        first, last = None, None
        helper(root)
        # close DLL
        last.right = first
        first.left = last
        return first
```

1. Custom Sort String Medium

You are given two strings order and s. All the words of order are unique and were sorted in some custom order previously.

Permute the characters of s so that they match the order that order was sorted. More specifically, if a character x occurs before a character y in order, then x should occur before y in the permuted string.

Return any permutation of s that satisfies this property.

Example 1:

Input: order = "cba", s = "abcd" Output: "cbad" Explanation: "a", "b", "c" appear in order, so the order of "a", "b", "c" should be "c", "b", and "a". Since "d" does not appear in order, it can be at any position in the returned string. "dcba", "cdba", "cbda" are also valid outputs. Example 2:

Input: order = "cbafg", s = "abcd" Output: "cbad"

In [ ]:
```
import collections
class Solution(object):
    def customSortString(self, S, T):
        # count[char] will be the number of occurrences of
        # 'char' in T.
        #Counter({'a': 1, 'b': 1, 'c': 3, 'd': 1})
        count = collections.Counter(T)
        ans = []

        # Write all characters that occur in S, in the order of S.
        for c in S:
            ans.append(c * count[c])
            # Set count[c] = 0 to denote that we do not need
            # to write 'c' to our answer anymore.
            count[c] = 0

        # Write all remaining characters that don't occur in S.
        # That information is specified by 'count'.
        for c in count:
            ans.append(c * count[c])

        return "".join(ans)

s="cba"
t="abcccd"

so = Solution()
so.customSortString(s,t)
```

Out[ ]:
```
'cccbad'
```

1. Insert into a Sorted Circular Linked List Medium

Given a Circular Linked List node, which is sorted in ascending order, write a function to insert a value insertVal into the list such that it remains a sorted circular list. The given node can be a

reference to any single node in the list and may not necessarily be the smallest value in the circular list.

If there are multiple suitable places for insertion, you may choose any place to insert the new value. After the insertion, the circular list should remain sorted.

If the list is empty (i.e., the given node is null), you should create a new single circular list and return the reference to that single node. Otherwise, you should return the originally given node.

Example 1:

Input: head = [3,4,1], insertVal = 2 Output: [3,4,1,2] Explanation: In the figure above, there is a sorted circular list of three elements. You are given a reference to the node with value 3, and we need to insert 2 into the list. The new node should be inserted between node 1 and node 3. After the insertion, the list should look like this, and we should still return node 3.

Example 2:

Input: head = [], insertVal = 1 Output: [1] Explanation: The list is empty (given head is null). We create a new single circular list and return the reference to that single node. Example 3:

Input: head = [1], insertVal = 0 Output: [1,0]

In [ ]:

```
class Node:
    def __init__(self, val=None, next=None):
        self.val = val
        self.next = next


class Solution:
    def insert(self, head: 'Node', insertVal: int) -> 'Node':
        node = Node(insertVal)

        if not head:
            node.next = node
            return node

        prev, curr = head, head.next

        while prev.next != head:
            # Case1: 1 <- Node(2) <- 3
            if prev.val <= node.val <= curr.val:
                break

            # Case2: 3 -> 1, 3 -> Node(4) -> 1, 3 -> Node(0) -> 1
            if prev.val > curr.val and (node.val > prev.val or node.val < curr.val):
                break

            prev, curr = prev.next, curr.next

        # Insert node.
        node.next = curr
        prev.next = node

        return head

#how to create linked list from list

class LinkedList:
```

```
    def __init__(self):
        self.head = None

    def print_in_order(self):
        '''
        print linked list element starting from the head, walking
        until the end of the list
        '''
        curr_node = self.head
        print(curr_node.val)
        while curr_node.next is not None:
            print(curr_node.next.val)
            curr_node = curr_node.next


class SinglyLinkedList(LinkedList):
    '''
    singly linked list
    '''

    def populate_from_set(self, set_to_use: set):
        '''
        iteratively populate a singly linked list from a set of values
        '''
        if len(set_to_use) == 0:
            raise ValueError('Cannot start a singly linked list from an empty set.')

        # then iterate through to the end of the set, linking each node to the next
        node_prev = None
        for set_i in range(len(set_to_use)):

            # set the current node
            node_curr = Node(val = set_to_use[set_i])

            # set the head of the SLL on the first pass through the set
            if set_i == 0:
                self.head = node_curr
            # otherwise we link the previous node to the current node
            else:
                node_prev.next = node_curr

            # then set the previous node to the current node for the next iteration
            node_prev = node_curr
```

In [ ]:

```
input = [3,4,1]
list = SinglyLinkedList.populate_from_set(LinkedList,input)
s = Solution()
s.insert(list,2)
```

Out[ ]:

```
<__main__.Node at 0x282ef99ac70>
```

In [ ]: