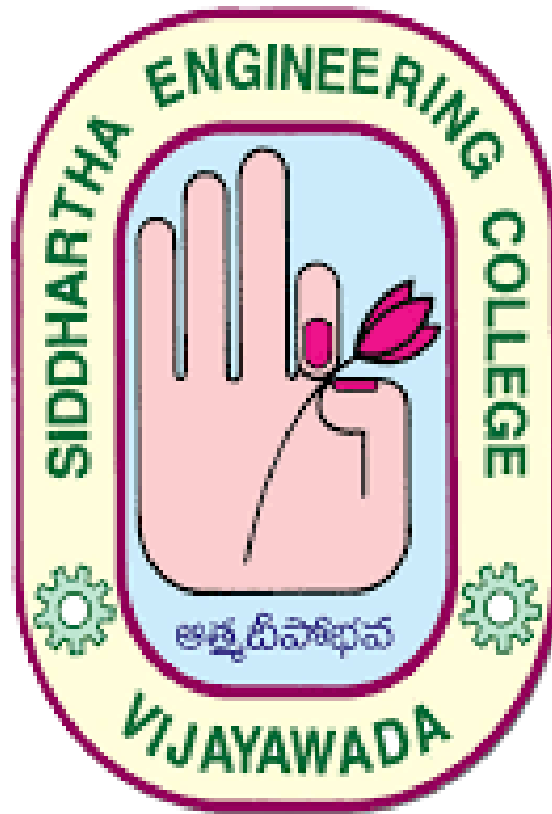


Velagapudi Ramakrishna Siddhartha Engineering College

Kanuru, 520001



ADVANCE PROGRAMMING LAB - III

Code : 20IT6353

WEEK – 1

Aim : Implement and validate the Stack Sequences.

Program :

class Solution:

```
def validateStackSequences(self, pushed: List[int], popped: List[int]) -> bool:
    stack=[]
    for i in pushed:
        stack.append(i)
        while stack and popped and stack[-1] == popped[0]: # stack top
            stack.pop()
            popped.pop(0)
    return not stack
```

Output :



Result : Sucessfully Executed the Program.

Aim : There are a number of plants in a garden. Each of the plants has been treated with some amount of pesticide. After each day, if any plant has more pesticide than the plant on its left, being weaker than the left one, it dies.

You are given the initial values of the pesticide in each of the plants. Determine the number of days after which no plant dies, i.e. the time after which there is no plant with more pesticide content than the plant to its left.

Program :

```
import math
import os
import random
import re
import sys

def poisonousPlants(p):
    stack = []
    max_days = 0
    for i in range(len(p)-1, -1, -1):
        kills = 0
        while (len(stack) > 0) and stack[-1][0] > p[i]:
            kills = max(kills + 1, stack.pop()[1])
        max_days = max(max_days, kills)
        stack.append((p[i], kills))
    return max_days

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')
```

```

n = int(input().strip())
p = list(map(int, input().rstrip().split()))
result = poisonousPlants(p)
fptr.write(str(result) + '\n')
fptr.close()

```

Output :

Sample Input

```

7
6 5 8 4 7 10 9

```

Sample Output

```

2

```

Explanation

Initially all plants are alive.

Plants = {(6,1), (5,2), (8,3), (4,4), (7,5), (10,6), (9,7)}

Plants[k] = (i,j) => jth plant has pesticide amount = i.

After the 1st day, 4 plants remain as plants 3, 5, and 6 die.

Plants = {(6,1), (5,2), (4,4), (9,7)}

After the 2nd day, 3 plants survive as plant 7 dies.

Plants = {(6,1), (5,2), (4,4)}

Plants stop dying after the 2nd day.

[Upload Code as File](#)
☐ Test against custom input
 [Run Code](#)
[Submit Code](#)

Congratulations!

You have passed the sample test cases. Click the submit button to run your code against all the test cases.

Sample Test case 0

Input (stdin)
 [Download](#)

1	7
2	6 5 8 4 7 10 9

Sample Test case 1

Your Output (stdout)
 [Download](#)

1	2
---	---

Sample Test case 2

Expected Output
 [Download](#)

1	2
---	---

Result : Sucessfully Executed the Program

Week – 2

Aim : implement Stacks using Queues.

Program :

```
from collections import deque

class MyStack:

    def __init__(self):
        self.queue1 = deque()
        self.queue2 = deque()

    def push(self, x: int) -> None:
        self.queue1.append(x)

    def pop(self) -> int:
        while len(self.queue1) > 1:
            self.queue2.append(self.queue1.popleft())
        item = self.queue1.popleft()
        self.queue1, self.queue2 = self.queue2, self.queue1
        return item

    def top(self) -> int:
        while len(self.queue1) > 1:
            self.queue2.append(self.queue1.popleft())
        item = self.queue1.popleft()
        self.queue2.append(item)
        self.queue1, self.queue2 = self.queue2, self.queue1
        return item
```

```
def empty(self) -> bool:  
    return len(self.queue1) == 0
```

Output :

Accepted Runtime: 41 ms



• Case 1

Input

```
["MyStack","push","push","top","pop","empty"]
```

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

Output

```
[null,null,null,2,2,false]
```

Expected

```
[null,null,null,2,2,false]
```

Result : Sucessfully Executed The Program.

Aim : Implement Circular Queue using OOPS Concepts.

Program :

```
class MyCircularQueue:
    def __init__(self, k: int):
        self.queue = [None] * k
        self.head = 0
        self.tail = 0
        self.size = 0
        self.capacity = k
    def enQueue(self, value: int) -> bool:
        if self.isFull():
            return False
        self.queue[self.tail] = value
        self.tail = (self.tail + 1) % self.capacity
        self.size += 1
        return True
    def deQueue(self) -> bool:
        if self.isEmpty():
            return False
        self.head = (self.head + 1) % self.capacity
        self.size -= 1
        return True
    def Front(self) -> int:
        if self.isEmpty():
```

```
        return -1

        return self.queue[self.head]

def Rear(self) -> int:

    if self.isEmpty():

        return -1

    return self.queue[(self.tail - 1 + self.capacity) % self.capacity]

def isEmpty(self) -> bool:

    return self.size == 0

def isFull(self) -> bool:

    return self.size == self.capacity
```

Output :

Accepted Runtime: 56 ms

• Case 1

Input

["MyCircularQueue", "enqueue", "enqueue", "enqueue", "enqueue", "Rear", "isFull", "deQueue", "enqueue", "Rear"]

[[3], [1], [2], [3], [4], [], [], [], [4], []]

Output

[null, true, true, true, false, 3, true, true, true, 4]

Expected

[null, true, true, true, false, 3, true, true, true, 4]

Result : Sucessfully Executed The Program.

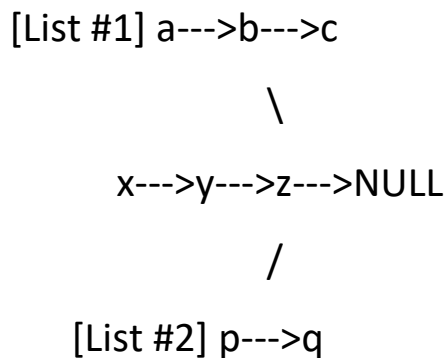
WEEK – 3

Aim : Given pointers to the head nodes of linked lists that merge together at some point, find the node where the two lists merge. The merge point is where both lists point to the same node, i.e. they reference the same memory location. It is guaranteed that the two head nodes will be different, and neither will be NULL. If the lists share a common node, return that node's data value.

Note: After the merge point, both lists will share the same node pointers.

Example :

In the diagram below, the two lists converge at Node x:



Program :

```

import math
import os
import random
import re
import sys

class SinglyLinkedListNode:
    def __init__(self, node_data):
        self.data = node_data
        self.next = None
  
```

```
class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
    def insert_node(self, node_data):
        node = SinglyLinkedListNode(node_data)
        if not self.head:
            self.head = node
        else:
            self.tail.next = node
        self.tail = node
    def print_singly_linked_list(self, node, sep, fptr):
        while node:
            fptr.write(str(node.data))
            node = node.next
            if node:
                fptr.write(sep)
    def findMergeNode(head1, head2):
        li=[]
        while head1 :
            li.append(head1)
            head1=head1.next
        temp=head2
        while head2:
            if head2 in li:
```

```
        return head2.data
    head2=head2.next
if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')
    tests = int(input())
    for tests_itr in range(tests):
        index = int(input())
        llist1_count = int(input())
        llist1 = SinglyLinkedList()
        for _ in range(llist1_count):
            llist1_item = int(input())
            llist1.insert_node(llist1_item)
        llist2_count = int(input())
        llist2 = SinglyLinkedList()
        for _ in range(llist2_count):
            llist2_item = int(input())
            llist2.insert_node(llist2_item)
        ptr1 = llist1.head;
        ptr2 = llist2.head;
        for i in range(llist1_count):
            if i < index:
                ptr1 = ptr1.next
        for i in range(llist2_count):
            if i != llist2_count-1:
                ptr2 = ptr2.next
```

```

ptr2.next = ptr1
result = findMergeNode(llist1.head, llist2.head)
fptr.write(str(result) + '\n')
fptr.close()

```

Output :

Test Case 0

```

1
 \
  2--->3--->NULL
 /
1

```

Test Case 1

```

1--->2
 \
  3--->Null
 /
1

```

Sample Output

```

2
3

```

Explanation

Test Case 0: As demonstrated in the diagram above, the merge node's data field contains the integer 2.

Line: 44 Col: 24

[Upload Code as File](#)
☐ Test against custom input

[Run Code](#)
[Submit Code](#)

Congratulations!

You have passed the sample test cases. Click the submit button to run your code against all the test cases.

✓ Sample Test case 0

✓ Sample Test case 1

Input (stdin)

Download

1	1
2	1
3	3
4	1
5	2
6	3
7	1
8	1

Your Output (stdout)

1	2
---	---

Result : Sucessfully Executed The Program.

Aim : Given pointers to the heads of two sorted linked lists, merge them into a single, sorted linked list. Either head pointer may be null meaning that the corresponding list is empty.

Example

headA refers to $1 \rightarrow 3 \rightarrow 7 \rightarrow NULL$

headB refers to $1 \rightarrow 2 \rightarrow NULL$

The new list is $1 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow NULL$

Program :

```
import math
import os
import random
import re
import sys

class SinglyLinkedListNode:
    def __init__(self, node_data):
        self.data = node_data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert_node(self, node_data):
        node = SinglyLinkedListNode(node_data)
```

```
    if not self.head:
        self.head = node
    else:
        self.tail.next = node
    self.tail = node

def print_singly_linked_list(node, sep, fptr):
    while node:
        fptr.write(str(node.data))
        node = node.next
        if node:
            fptr.write(sep)

def mergeLists(head1, head2):
    li=[]
    temp1=head1
    temp2=head2
    while temp1 is not None:
        li.append(temp1.data)
        temp1=temp1.next
    while temp2 is not None:
        li.append(temp2.data)
        temp2=temp2.next
    li.sort()
    ll=SinglyLinkedList()
    for i in range(0,len(li)):
        ll.insert_node(li[i])
```

```
return ll.head
```

```
if __name__ == '__main__':
```

```
    fptr = open(os.environ['OUTPUT_PATH'], 'w')
```

```
    tests = int(input())
```

```
    for tests_itr in range(tests):
```

```
        llist1_count = int(input())
```

```
        llist1 = SinglyLinkedList()
```

```
        for _ in range(llist1_count):
```

```
            llist1_item = int(input())
```

```
            llist1.insert_node(llist1_item)
```

```
        llist2_count = int(input())
```

```
        llist2 = SinglyLinkedList()
```

```
        for _ in range(llist2_count):
```

```
            llist2_item = int(input())
```

```
            llist2.insert_node(llist2_item)
```

```
        llist3 = mergeLists(llist1.head, llist2.head)
```

```
        print_singly_linked_list(llist3, ' ', fptr)
```

```
        fptr.write('\n')
```

```
fptr.close()
```

Output :

- $1 \leq t \leq 10$
- $1 \leq n, m \leq 1000$
- $1 \leq list[i] \leq 1000$, where $list[i]$ is the i^{th} element of the list.

Sample Input

```
1
3
1
2
3
2
3
4
```

Sample Output

```
1 2 3 3 4
```

Explanation

The first linked list is: $1 \rightarrow 3 \rightarrow 7 \rightarrow NULL$

The second linked list is: $3 \rightarrow 4 \rightarrow NULL$

Hence, the merged linked list is: $1 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow 4 \rightarrow NULL$

Line: 57 Col: 30

☐ Test against custom input

Congratulations!

You have passed the sample test cases. Click the submit button to run your code against all the test cases.

☒ Sample Test case 0
 ☒ Sample Test case 1

Input (stdin)

1	1
2	3
3	1
4	2
5	3
6	2
7	3
8	4

Your Output (stdout)

1	1 2 3 3 4
---	-----------

Result : Sucessfully Executed The Program .

WEEK – 4

Aim : Given the pointer to the head node of a doubly linked list, reverse the order of the nodes in place. That is, change the *next* and *prev* pointers of the nodes so that the direction of the list is reversed. Return a reference to the head node of the reversed list.

Note: The head node might be NULL to indicate that the list is empty.

Function Description :

Complete the *reverse* function in the editor below.

reverse has the following parameter(s):

- *DoublyLinkedListNode head*: a reference to the head of a *DoublyLinkedList*

Returns

- *DoublyLinkedListNode*: a reference to the head of the reversed list

Program :

```
import math
import os
import random
import re
import sys

class DoublyLinkedListNode:
    def __init__(self, node_data):
        self.data = node_data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
```

```
self.head = None

self.tail = None

def insert_node(self, node_data):

    node = DoublyLinkedListNode(node_data)

    if not self.head:

        self.head = node

    else:

        self.tail.next = node

        node.prev = self.tail

    self.tail = node

def print_doubly_linked_list(node, sep, fptr):

    while node:

        fptr.write(str(node.data))

        node = node.next

        if node:

            fptr.write(sep)

def reverse(llist):

    # Write your code here

    head = llist

    while llist:

        curr = llist.prev

        llist.prev = llist.next

        llist.next = curr

        head = llist

        llist = llist.prev
```

```

return head

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')

    t = int(input())

    for t_itr in range(t):
        llist_count = int(input())

        llist = DoublyLinkedList()

        for _ in range(llist_count):
            llist_item = int(input())

            llist.insert_node(llist_item)

        llist1 = reverse(llist.head)

        print_doubly_linked_list(llist1, ' ', fptr)

        fptr.write('\n')

    fptr.close()

```

Output :

Note: The head node might be NULL to indicate that the list is empty.

Function Description

Complete the reverse function in the editor below.

reverse has the following parameter(s):

- DoublyLinkedListNode head: a reference to the head of a DoublyLinkedList

Returns

- DoublyLinkedListNode: a reference to the head of the reversed list

Input Format

The first line contains an integer t , the number of test cases.

Each test case is of the following format:

- The first line contains an integer n , the number of elements in the linked list.
- The next n lines contain an integer each denoting an element of the linked list.

Constraints

- $1 \leq t \leq 10$
- $0 \leq n \leq 1000$

Congratulations!

You have passed the sample test cases. Click the submit button to run your code against all the test cases.

Sample Test case 0

1 1
2 4
3 1
4 2
5 3
6 4

Sample Test case 1

Sample Test case 2

Your Output (stdout)

1	4 3 2 1
---	---------

Expected Output

1	4 3 2 1
---	---------

[Download](#)

Result : Sucessfully Executed The Program.

Aim : You have a browser of one tab where you start on the homepage and you can visit another url, get back in the history number of steps or move forward in the history number of steps.

Implement the BrowserHistory class:

BrowserHistory(string homepage) Initializes the object with the homepage of the browser.

void visit(string url) Visits url from the current page. It clears up all the forward history.

string back(int steps) Move steps back in history. If you can only return x steps in the history and steps > x, you will return only x steps. Return the current url after moving back in history at most steps.

string forward(int steps) Move steps forward in history. If you can only forward x steps in the history and steps > x, you will forward only x steps. Return the current url after forwarding in history at most steps.

Program :

```
class BrowserHistory:
```

```
    def __init__(self, homepage: str):
```

```
        self.history = [homepage]
```

```
        self.current = 0
```

```
    def visit(self, url: str) -> None:
```

```
        self.history = self.history[:self.current+1] + [url]
```

```
        self.current += 1
```

```
    def back(self, steps: int) -> str:
```

```
        self.current = max(0, self.current - steps)
```

```
        return self.history[self.current]
```

```
def forward(self, steps: int) -> str:
    self.current = min(len(self.history)-1, self.current + steps)
    return self.history[self.current]
```

Your BrowserHistory object will be instantiated and called as such:

```
# obj = BrowserHistory(homepage)
```

```
# obj.visit(url)
```

```
# param_2 = obj.back(steps)
```

```
# param_3 = obj.forward(steps)
```

Output :



The screenshot shows a test case result on a coding platform. The 'Result' tab is selected, showing 'Accepted' in green text with a runtime of 39 ms. Below this, 'Case 1' is listed. The 'Input' section contains two lines of code: a sequence of operations and a list of URLs with step counts. The 'Output' section shows the resulting sequence of URLs. The 'Expected' section is partially visible at the bottom.

Testcase **Result**

Accepted Runtime: 39 ms

- Case 1

Input

```
["BrowserHistory","visit","visit","visit","back","back","forward","visit","forward","back","back"]
```

```
[["leetcode.com"],["google.com"],["facebook.com"],["youtube.com"],[1],[1],[1],["linkedin.com"],[2],[2],[7]]
```

Output

```
[null,null,null,null,"facebook.com","google.com","facebook.com",null,"linkedin.com","google.com","leetcode.com"]
```

Expected

Result : Sucessfully Executed The Program .

WEEK – 5

Aim : Implement Symmetric Tree

Program :

```
class Solution:
```

```
    def isSymmetric(self, root):
```

```
        if root is None:
```

```
            return True
```

```
        else:
```

```
            return self.isMirror(root.left, root.right)
```

```
    def isMirror(self, left, right):
```

```
        if left is None and right is None:
```

```
            return True
```

```
        if left is None or right is None:
```

```
            return False
```

```
        if left.val == right.val:
```

```
            outPair = self.isMirror(left.left, right.right)
```

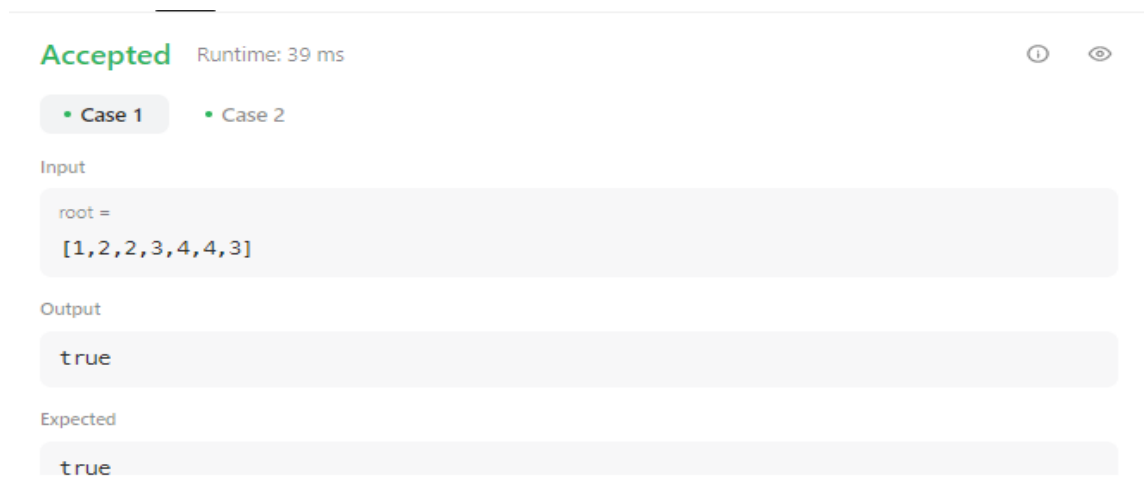
```
            inPair = self.isMirror(left.right, right.left)
```

```
            return outPair and inPair
```

```
        else:
```

```
            return False
```

Output :



Result : Sucessfully Executed The Program.

Aim : Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A valid 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.

Program :

class Solution:

```
def isValidBST(self, root: Optional[TreeNode]) -> bool:
```

```
    low, high = float("-inf"), float("inf")
```

```
    def isValidbst(root, low, high):
```

```
        if not root:
```

```
            return True
```

```
        if root.val <= high and root.val >= low:
```

```
            left_sub_tree = isValidbst(root.left, low, root.val - 1)
```

```
            right_sub_tree = isValidbst(root.right, root.val + 1, high)
```

```
            return left_sub_tree and right_sub_tree
```

```
        else:
```

```
            return False
```

```
    return isValidbst(root, low, high)
```


Output :

Testcase

Result

Accepted Runtime: 38 ms

• Case 1

• Case 2

Input

root =
[2,1,3]

Output


true

Expected

true

♥ Contribute a testcase

Console ▾

 Run **Submit**

Result : Sucessfully Executed The Program.

WEEK – 6

Aim : Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

Program :

Definition for a binary tree node.

class TreeNode:

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

self.val = val

self.left = left

self.right = right

class Solution:

def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:

total_nums = len(nums)

if not total_nums:

return None

mid_node = total_nums // 2

return TreeNode(

nums[mid_node],

self.sortedArrayToBST(nums[:mid_node]),

self.sortedArrayToBST(nums[mid_node + 1 :])

)

Output :

Testcase

Result

Accepted Runtime: 43 ms

• Case 1

• Case 2

Input

nums =
[-10,-3,0,5,9]

Output

[0,-3,9,-10,null,5]

Expected

[0,-3,9,-10,null,5]

Console ▾



Run

Submit

Result : Sucessfully Executed the Program.

Aim : You are given two binary trees root1 and root2.

Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

Return the merged tree.

Note: The merging process must start from the root nodes of both trees.

Program :

#Definition for a binary tree node.

```
class TreeNode:
```

```
    def __init__(self, val=0, left=None, right=None):
```

```
        self.val = val
```

```
        self.left = left
```

```
        self.right = right
```

```
class Solution:
```

```
    def mergeTrees(self, root1: TreeNode, root2: TreeNode) -> TreeNode:
```

```
        if not root1:
```

```
            return root2
```

```
        if not root2:
```

```
            return root1
```

```
        # merge the nodes that are overlapped
```

```
        root1.val += root2.val
```

```
        root1.left = self.mergeTrees(root1.left, root2.left)
```

```
root1.right = self.mergeTrees(root1.right, root2.right)
return root1
```

Output :

The screenshot displays a coding platform interface with a 'Testcase' tab and a 'Result' tab. The 'Result' tab is active, showing a green 'Accepted' status with a runtime of 50 ms. Below this, there are two tabs for 'Case 1' and 'Case 2'. The 'Input' section shows two trees: root1 = [1, 3, 2, 5] and root2 = [2, 1, 3, null, 4, null, 7]. The 'Output' section shows the result [3, 4, 5, 5, 4, null, 7]. The 'Expected' section also shows [3, 4, 5, 5, 4, null, 7]. At the bottom, there is a 'Console' dropdown, a bug icon, and 'Run' and 'Submit' buttons.

Testcase Result

Accepted Runtime: 50 ms

• Case 1 • Case 2

Input

root1 =
[1, 3, 2, 5]

root2 =
[2, 1, 3, null, 4, null, 7]

Output

[3, 4, 5, 5, 4, null, 7]

Expected

[3, 4, 5, 5, 4, null, 7]

Console ▾

Run Submit

Result : Sucessfully Executed The Program.

WEEK – 7

Aim : You are given an array of variable pairs equations and an array of real numbers values, where $\text{equations}[i] = [A_i, B_i]$ and $\text{values}[i]$ represent the equation $A_i / B_i = \text{values}[i]$. Each A_i or B_i is a string that represents a single variable.

You are also given some queries, where $\text{queries}[j] = [C_j, D_j]$ represents the j th query where you must find the answer for $C_j / D_j = ?$.

Return the answers to all queries. If a single answer cannot be determined, return -1.0.

Note: The input is always valid. You may assume that evaluating the queries will not result in division by zero and that there is no contradiction.

Program :

class Solution:

```
def calcEquation(self, equations: List[List[str]], values: List[float], queries:
List[List[str]]) -> List[float]:
```

```
    # Build the graph
```

```
    graph = defaultdict(dict)
```

```
    for i in range(len(equations)):
```

```
        u, v = equations[i]
```

```
        graph[u][v] = values[i]
```

```
        graph[v][u] = 1 / values[i]
```

```
    # Helper function to perform DFS and find the path value
```

```
    def dfs(start, end, visited):
```

```
        # If we have already visited this node or it doesn't exist in the graph,
        return -1.0
```

```
        if start in visited or start not in graph:
```

```
        return -1.0

    # If we have reached the end node, return the path value
    if start == end:
        return 1.0

    # Mark the current node as visited
    visited.add(start)

    # Traverse the neighbors and find the path value recursively
    for neighbor, value in graph[start].items():
        path_value = dfs(neighbor, end, visited)

        # If we have found a valid path, return the product of the current value
        # and path value
        if path_value != -1.0:
            return value * path_value

    # If we haven't found a valid path, return -1.0
    return -1.0

# Calculate the answer for each query
result = []
for query in queries:
    start, end = query

    # Perform DFS to find the path value
    result.append(dfs(start, end, set()))

return result
```

Output :

The screenshot displays a web-based code execution environment. At the top, a tab labeled 'Result' is active, showing a green 'Accepted' status with a runtime of 53 ms. Below this, three tabs for 'Case 1', 'Case 2', and 'Case 3' are visible, with 'Case 1' selected. The 'Input' section contains three variables: 'equations' with a 2x2 array of strings, 'values' with a 1x2 array of floats, and 'queries' with a 1x5 array of string pairs. The 'Output' section shows a 1x5 array of floats, which matches the 'Expected' output shown below it. At the bottom, there is a 'Console' dropdown, a 'Run' button, and a green 'Submit' button.

Testcase **Result**

Accepted Runtime: 53 ms

• Case 1 • Case 2 • Case 3

Input

equations =
[["a", "b"], ["b", "c"]]

values =
[2.0, 3.0]

queries =
[["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"]]

Output

[6.00000, 0.50000, -1.00000, 1.00000, -1.00000]

Expected

[6.00000, 0.50000, -1.00000, 1.00000, -1.00000]

Console ▾ Run **Submit**

Result : Sucessfully Executed The program.

Aim : You are given a network of n nodes, labeled from 1 to n . You are also given times, a list of travel times as directed edges $times[i] = (u_i, v_i, w_i)$, where u_i is the source node, v_i is the target node, and w_i is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k . Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

Program :

class Solution:

```
def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
```

```
    adj_list = defaultdict(list)
```

```
    for x,y,w in times:
```

```
        adj_list[x].append((w, y))
```

```
    visited=set()
```

```
    heap = [(0, k)]
```

```
    while heap:
```

```
        travel_time, node = heapq.heappop(heap)
```

```
        visited.add(node)
```

```
        if len(visited)==n:
```

```
            return travel_time
```

```
        for time, adjacent_node in adj_list[node]:
```

```
            if adjacent_node not in visited:
```

```
                heapq.heappush(heap, (travel_time+time, adjacent_node))
```

```
    return -1
```

Output :

The screenshot displays the 'Result' tab of a coding platform. At the top, it shows 'Accepted' in green text and 'Runtime: 40 ms'. Below this, there are three tabs: 'Case 1', 'Case 2', and 'Case 3', with 'Case 1' being the active one. The 'Input' section contains three text boxes: 'times =' followed by '[[2, 1, 1], [2, 3, 1], [3, 4, 1]]', 'n =' followed by '4', and 'k =' followed by '2'. The 'Output' section shows a single text box with the value '2'. The 'Expected' section also shows a single text box with the value '2'. At the bottom of the interface, there is a 'Console' dropdown menu, a 'Contribute a testcase' link with a heart icon, a 'Run' button, and a green 'Submit' button.

Testcase Result

Accepted Runtime: 40 ms

• Case 1 • Case 2 • Case 3

Input

times =
[[2, 1, 1], [2, 3, 1], [3, 4, 1]]

n =
4

k =
2

Output

2

Expected

2

Contribute a testcase

Console ▾ Run Submit

Result : Sucessfully Executed The Program.

WEEK – 8

Aim : Kitty has a tree, T , consisting of n nodes where each node is uniquely labeled from 1 to n . Her friend Alex gave her q sets, where each set contains k distinct nodes. Kitty needs to calculate the following expression on each set:

$$\left(\sum_{\{u,v\}} u \cdot v \cdot \text{dist}(u, v) \right) \bmod (10^9 + 7)$$

Program :

```
import networkx as nx
from itertools import combinations
import matplotlib.pyplot as plt

def dist(u,v):
    #print("Distance Nodes : " , int(sp[u][v]))
    duv = sp[u][v]
    return duv

def product_tuple(x):
    prodt = 1
    for i in range(len(x)):
        prodt *= x[i]
    return prodt

def kitty_formula(combi):
    res = 1
    for k in range(len(combi)):
        dtup = combi[k]
        temp = product_tuple(dtup) * dist(dtup[0], dtup[1])
```

```
    res = res + temp

print("\n\nFinal Result : ", res-1)

# inputs taking
nodes, queries = map(int, input("Enter Nodes And Queries : ").split())
#print(nodes, queries)

edges = []

for i in range(nodes-1):
    edge = list(map(int, input("Enter intial and final nodes : ").split()))
    #print("edge entered is : ", edge)
    edges.append(edge)

print("All Edge List : ", edges)

for i in range(queries):
    lq = int(input("Enter length of the query set : "))
    q1 = list(map(int, input("Enter a pair : ").split()[:lq]))
    #print("q1 : ", q1)
    if len(q1) == 1:
        # calculation Start
        print("Final result : ", 0)
    else:
        # calculation Start
        combi = list(combinations(q1, 2))
        G = nx.Graph()
        G.add_nodes_from([h for h in range(1, nodes+1)])
        G.add_edges_from(edges)
        sp = dict(nx.all_pairs_shortest_path_length(G))
```

```
kitty_formula(combi)
nx.draw(G, with_labels=True)
plt.show()
```

Output :

Enter Nodes And Queries : 7 3

Enter intial and final nodes : 1 2

Enter intial and final nodes : 1 3

Enter intial and final nodes : 1 4

Enter intial and final nodes : 3 5

Enter intial and final nodes : 3 6

Enter intial and final nodes : 3 7

All Edge List : [[1, 2], [1, 3], [1, 4], [3, 5], [3, 6], [3, 7]]

Enter length of the query set : 2

Enter a pair : 2 4

Final Result : 16

Enter length of the query set : 1

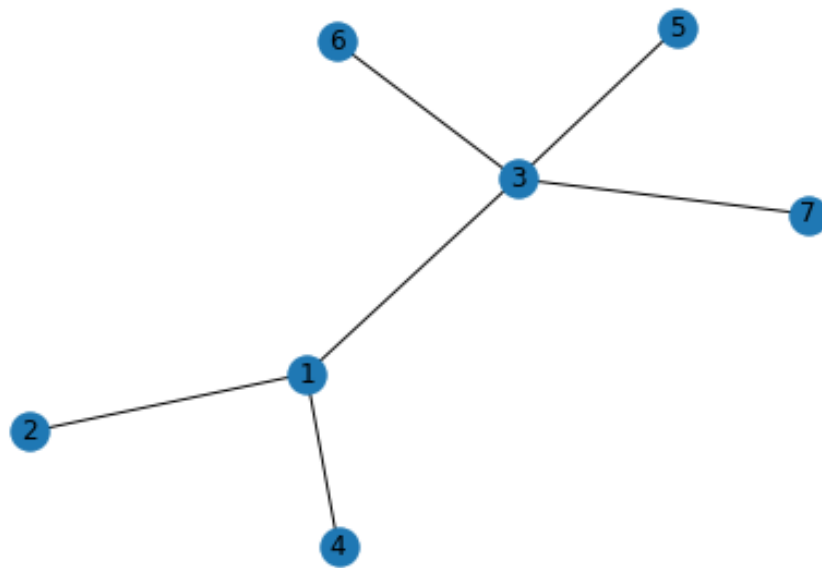
Enter a pair : 5

Final result : 0

Enter length of the query set : 3

Enter a pair : 2 4 5

Final Result : 106



Result : Sucessfully Executed The Program.

Aim : Implement Emergence Of Connectivity Problem Using Networkx module.

Program :

```
import networkx as nx
import matplotlib.pyplot as plt
import random
import numpy as np

# Add n number of nodes in the graph and return it.
def add_nodes(n):
    G = nx.Graph()
    G.add_nodes_from(range(n))
    return G

# add one random edge
def add_random_edge(G):
    v1 = random.choice(list(G.nodes()))
    v2 = random.choice(list(G.nodes()))
    if v1 != v2:
        G.add_edge(v1,v2)
    return G

# it add random edges in graph until it becomes connected
def add_till_connectivity(G):
    while nx.is_connected(G) == False:
        G = add_random_edge(G)
    return G

# creates an instance of entire process. it takes as input number of nodes and
```

```
# returns the number of edges for connectivity.
```

```
def create_instance(n):
```

```
    G = add_nodes(n)
```

```
    G = add_till_connectivity(G)
```

```
    return G.number_of_edges()
```

```
# Average it over 100 instances
```

```
def create_avg_instance(n):
```

```
    list1 = []
```

```
    for i in range(0,100):
```

```
        list1.append(create_instance(n))
```

```
    return np.average(list1)
```

```
# plot the desired for different number of edges
```

```
def plot_connectivity():
```

```
    x = []
```

```
    y = []
```

```
    i = 10 # it tells no of nodes
```

```
    while i <= 100:
```

```
        x.append(i)
```

```
        y.append(create_avg_instance(i))
```

```
        i = i + 10
```

```
    plt.xlabel("Number of Nodes")
```

```
    plt.ylabel("Number of edges required to connect the graph")
```

```
    plt.title("Emergence of Connectivity")
```

```
    plt.plot(x,y)
```



```
x1 = []
y1 = []
i1 = 10
while i1 <= 100:
    x1.append(i1)
    y1.append(i1*np.log(i1))
    # y1.append(i1*float(np.log(i1))/2)
    i1 = i1 + 10
plt.plot(x1, y1)
plt.show()
```

```
g = add_nodes(10)
print("No of nodes : ", g.number_of_nodes())
print("Connected or not : ", nx.is_connected(g))
g1 = add_random_edge(g)
print("new edge added : ", g1.edges())
g2 = add_till_connectivity(g1)
print("Total edges in a g2 : ", g2.edges())
print("Total no of edges : ", g2.number_of_edges())
print("Connected or not : ", nx.is_connected(g2))
d = create_instance(10)
print("No of edges required for connectivity : ", d)
plot_connectivity()
```

Output :

No of nodes : 10

Connected or not : False

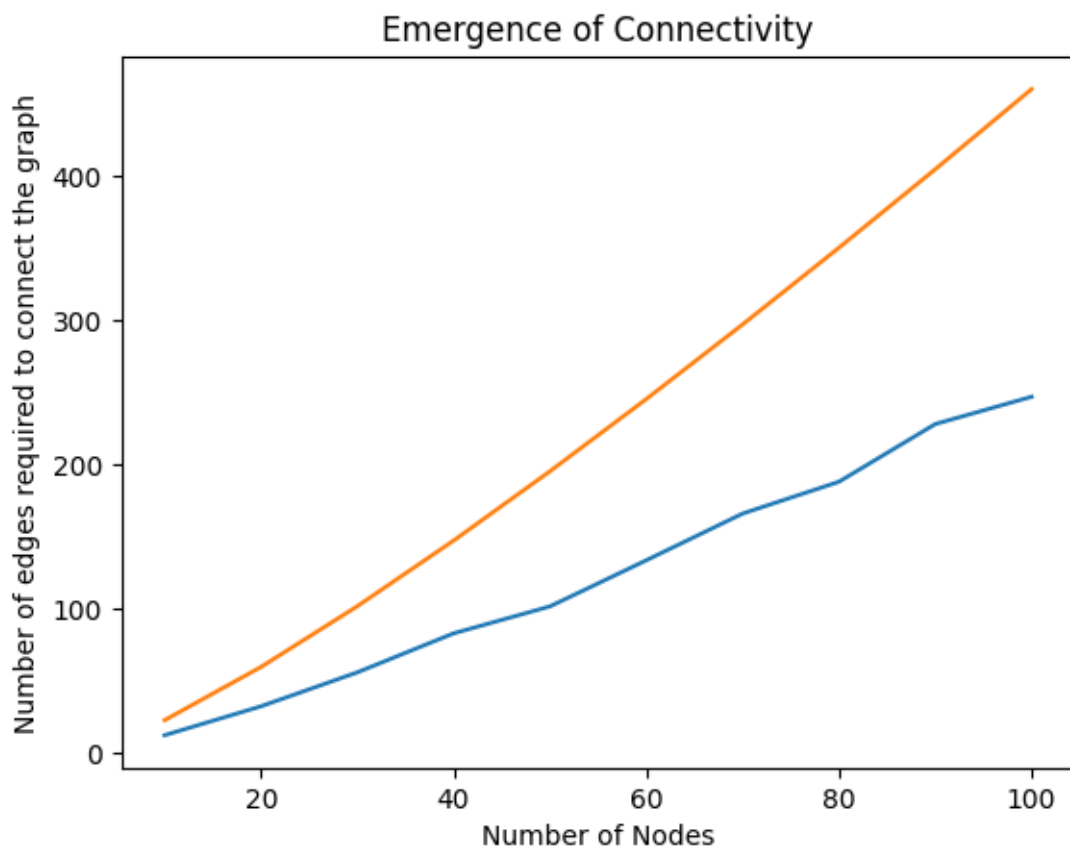
new edge added : [(2, 9)]

Total edges in a g2 : [(0, 8), (0, 6), (1, 9), (1, 7), (2, 9), (3, 9), (3, 4), (3, 7), (3, 5), (5, 6)]

Total no of edges : 10

Connected or not : True

No of edges required for connectivity : 13



Result : Sucessfully Executed The Program.