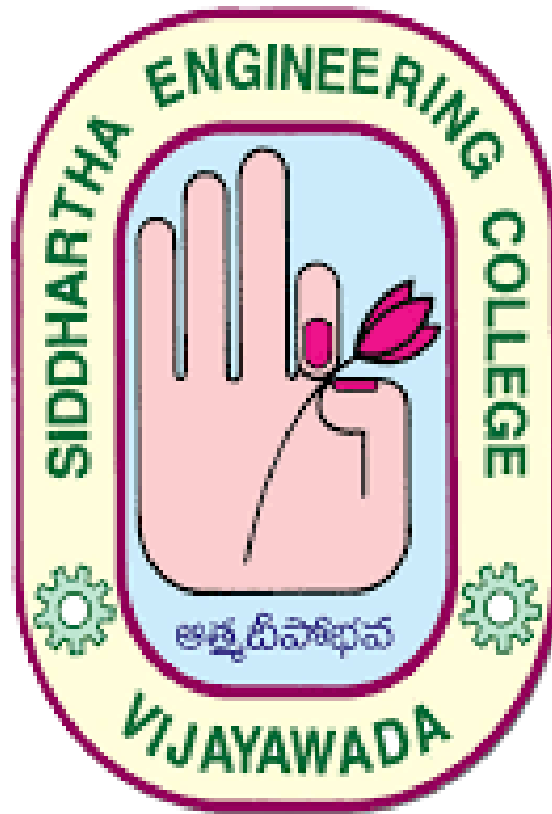


Velagapudi Ramakrishna Siddhartha Engineering College

Kanuru, 520001



ADVANCE PROGRAMMING LAB - III

Code : 20IT6353

WEEK – 1

Aim :

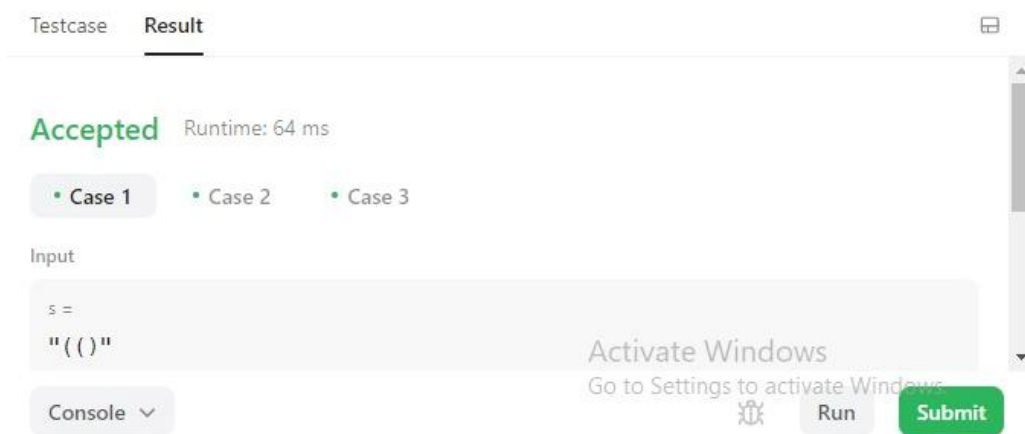
Given a string containing just the characters '(' and ')', return the length of the longest valid (well-formed) parentheses substring.

Program :

class Solution:

```
def longestValidParentheses(self, s: str) -> int:
    max_length = 0
    stck=[-1] # initialize with a start index
    for i in range(len(s)):
        if s[i] == '(':
            stck.append(i)
        else:
            stck.pop()
            if not stck: # if popped -1, add a new start index
                stck.append(i)
            else:
                max_length=max(max_length, i-stck[-1]) # update the length of the
valid substring
    return max_length
```

Output :



The screenshot shows a web-based code execution interface. At the top, there are two tabs: 'Testcase' and 'Result', with 'Result' being the active tab. Below the tabs, the status 'Accepted' is displayed in green, followed by 'Runtime: 64 ms'. Underneath, there are three buttons labeled 'Case 1', 'Case 2', and 'Case 3', with 'Case 1' being selected. Below these buttons, the 'Input' section shows the code 's = "("' and 's = "()"'. At the bottom, there is a 'Console' dropdown menu, a 'Run' button, and a green 'Submit' button. An 'Activate Windows' watermark is visible in the background.

Result : Successfully Executed the Program.

Aim : Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

MinStack() initializes the stack object.

void push(int val) pushes the element val onto the stack.

void pop() removes the element on the top of the stack.

int top() gets the top element of the stack.

int getMin() retrieves the minimum element in the stack.

You must implement a solution with $O(1)$ time complexity for each function.

Program :

```
class MinStack {
    List<Integer> list;
    PriorityQueue<Integer> pqueue;
    public MinStack() {
        list = new ArrayList<>();
        pqueue = new PriorityQueue<>();
    }

    public void push(int val) {
        list.add(val);
        pqueue.offer(val);
    }

    public void pop() {
        Integer n = list.get(list.size() - 1);
```

```
list.remove(list.size() - 1);  
pqueue.remove(n);  
}  
  
public int top() {  
    return list.get(list.size() - 1);  
}  
  
public int getMin() {  
    return pqueue.peek();  
}  
}
```

Output :

The screenshot shows a code execution interface with two tabs: 'Testcase' and 'Result'. The 'Result' tab is active, displaying a green 'Accepted' status with a runtime of 0 ms. Below this, a 'Case 1' label is shown. The 'Input' section contains a text box with the sequence: ["MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin"]. At the bottom, there is a 'Console' dropdown menu, a 'Run' button, and a green 'Submit' button. A semi-transparent 'Activate Windows' watermark is visible in the background.

Result : Successfully Executed the Program

Week – 2

Aim : You are given an integer array nums and two integers minK and maxK.

A fixed-bound subarray of nums is a subarray that satisfies the following conditions:

The minimum value in the subarray is equal to minK.

The maximum value in the subarray is equal to maxK.

Return the number of fixed-bound subarrays.

A subarray is a contiguous part of an array.

Program :

class Solution:

```
def countSubarrays(self, nums: List[int], minK: int, maxK: int) -> int:
```

```
    x,a,b = -1,-2,-2
```

```
    res = 0
```

```
    for i,n in enumerate(nums):
```

```
        if n>maxK or n<minK: x = i
```

```
        if n == minK: a = i
```

```
        if n == maxK: b = i
```


```
        res += max(0, min(a,b) - x)
```

```
    return res
```

Output :

Testcase

Result



Accepted Runtime: 41 ms

• Case 1

• Case 2

Input

nums =
[1,3,5,2,7,5]

minK =
1

Result : Successfully Executed The Program.

Aim : Given an integer array nums and an integer k, return the maximum sum of a non-empty subsequence of that array such that for every two consecutive integers in the subsequence, nums[i] and nums[j], where $i < j$, the condition $j - i \leq k$ is satisfied.

A subsequence of an array is obtained by deleting some number of elements (can be zero) from the array, leaving the remaining elements in their original order.

Program :

```
class Monoqueue(collections.deque):
```

```
    def enqueue(self, val):
```

```
        count = 1
```

```
        while self and self[-1][0] < val:
```

```
            count += self.pop()[1]
```

```
        self.append([val, count])
```

```
    def dequeue(self):
```

```
        ans = self.max()
```

```
        self[0][1] -= 1
```

```
        if self[0][1] <= 0:
```

```
            self.popleft()
```

```
        return ans
```

```
    def max(self):
```

```
        return self[0][0] if self else 0
```



```
class Solution(object):  
    def constrainedSubsetSum(self, A, K):  
        monoq = Monoqueue()  
        ans = max(A)  
        for i, x in enumerate(A):  
            monoq.enqueue(x + max(0, monoq.max()))  
            if i >= K:  
                ans = max(ans, monoq.dequeue())  
        return max(ans, monoq.dequeue())
```

Output :

Testcase

Result

Accepted Runtime: 64 ms

• Case 1

• Case 2

• Case 3

Input

nums =
[10,2,-10,5,20]

k =
2

Result : Sucessfully Executed The Program.

WEEK – 3

Aim : You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Program :

Definition for singly-linked list.

class ListNode:

def __init__(self, val=0, next=None):

self.val = val

self.next = next

class Solution:

def mergeKLists(self, lists: List[ListNode]) -> ListNode:

l = ListNode() # the new list that we want to return

t = l # taking a temporary copy of the new list as we need to move to next pointers to store data.

get the minimum front value of all linked lists in the input list.

def get_min():

min_val, min_idx = float('inf'), -1

for i in range(len(lists)):

if lists[i] != None and lists[i].val < min_val:

min_val = lists[i].val

min_idx = i

if min_idx != -1:

when a min value is found,

```

# increment the linked list

# so that we don't consider the same min value the next time

# and also the next value of linked list comes at the front

lists[min_indx] = lists[min_indx].next

return min_val

while(1):

    x = get_min() # get the min value to add to new list

    if (x == float('inf')):

        # if min value is not obtained that means all the linked lists are
        traversed so break

        break

    c = ListNode(val=x)

    t.next = c

    t = t.next

return l.next # as we made l to be just a head for our actual linked list

```

Output :

| Testcase | Result |
|---|--------|
| <div>Accepted Runtime: 52 ms</div> <div> Case 1 Case 2 Case 3 </div> <div> Input lists = [[1,4,5],[1,3,4],[2,6]] </div> <div> Output [1,1,2,3,4,4,5,6] </div> | |

Result : Sucessfully Executed The Program.

Aim : Given the head of a singly linked list and two integers left and right where $\text{left} \leq \text{right}$, reverse the nodes of the list from position left to position right, and return the reversed list.

Program :

Definition for singly-linked list.

class ListNode:

def __init__(self, val=0, next=None):

self.val = val

self.next = next

class Solution:

def reverseBetween(self, head: Optional[ListNode], left: int, right: int) -> Optional[ListNode]:

def reverse(head, left, right):

if (head and not head.next) or not head:

return head, head

prev = None

curr = tail = head

nxt = head.next

while(left != right + 1):

curr.next = prev

prev = curr

curr = nxt

if nxt:

nxt = nxt.next

left += 1

return prev, tail

```

left_boundary = right_boundary = ListNode(next=head)
count_r = count_l = 0
while(count_r!=right+1):
    right_boundary=right_boundary.next
    if (count_l != left-1):
        left_boundary=left_boundary.next
        count_l+=1
    count_r+=1
reverse_head,reverse_tail = reverse(left_boundary.next,left,right)
if count_l == 0:
    head = reverse_head
else:
    left_boundary.next = reverse_head
reverse_tail.next = right_boundary
return head

```

Output :

| Testcase | Result |
|---|--------|
| <div>Accepted Runtime: 42 ms</div> <div> <div>Case 1</div> <div>Case 2</div> </div> <div>Input</div> <div> <div>head =</div> <div>[1,2,3,4,5]</div> <div>left =</div> <div>2</div> </div> | |

Result : Sucessfully Executed The Program .

WEEK – 4

Aim : Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.

Implement the LRUCache class:

LRUCache(int capacity) Initialize the LRU cache with positive size capacity.

int get(int key) Return the value of the key if the key exists, otherwise return -1.

void put(int key, int value) Update the value of the key if the key exists.

Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key.

The functions get and put must each run in $O(1)$ average time complexity.

Program :

```
class LRUCache:
```

```
    def __init__(self, capacity: int):
```

```
        self.capacity = capacity
```

```
        self.values = OrderedDict()
```

```
    def get(self, key: int) -> int:
```

```
        if key not in self.values:
```

```
            return -1
```

```
        else:
```

```
            self.values[key] = self.values.pop(key)
```

```
            return self.values[key]
```

```
    def put(self, key: int, value: int) -> None:
```

```
        if key not in self.values:
```

```
            if len(self.values) == self.capacity:
```

```
self.values.popitem(last=False)
else:
    self.values.pop(key)
self.values[key] = value
```

Output :



Result : Successfully Executed The Program.

Aim : Design a data structure to store the strings' count with the ability to return the strings with minimum and maximum counts.

Implement the AllOne class:

AllOne() Initializes the object of the data structure.

inc(String key) Increments the count of the string key by 1. If key does not exist in the data structure, insert it with count 1.

dec(String key) Decrements the count of the string key by 1. If the count of key is 0 after the decrement, remove it from the data structure. It is guaranteed that key exists in the data structure before the decrement.

getMaxKey() Returns one of the keys with the maximal count. If no element exists, return an empty string "".

getMinKey() Returns one of the keys with the minimum count. If no element exists, return an empty string "".

Note that each function must run in $O(1)$ average time complexity.

Program :

```
from collections import defaultdict
```

```
class Node(object):
```

```
    def __init__(self):
```

```
        self.key_set = set([])
```

```
        self.prev, self.nxt = None, None
```

```
    def add_key(self, key):
```

```
        self.key_set.add(key)
```

```
    def remove_key(self, key):
```

```
        self.key_set.remove(key)
```

```
    def get_any_key(self):
```

```
        if self.key_set:
```

```
            result = self.key_set.pop()
```



```
        self.add_key(result)
        return result
    else:
        return None
def count(self):
    return len(self.key_set)
def is_empty(self):
    return len(self.key_set) == 0
class DoubleLinkedList(object):
    def __init__(self):
        self.head_node, self.tail_node = Node(), Node()
        self.head_node.nxt, self.tail_node.prev = self.tail_node, self.head_node
        return
    def insert_after(self, x):
        node, temp = Node(), x.nxt
        x.nxt, node.prev = node, x
        node.nxt, temp.prev = temp, node
        return node
    def insert_before(self, x):
        return self.insert_after(x.prev)
    def remove(self, x):
        prev_node = x.prev
        prev_node.nxt, x.nxt.prev = x.nxt, prev_node
        return
```

```
def get_head(self):
    return self.head_node.nxt

def get_tail(self):
    return self.tail_node.prev

def get_sentinel_head(self):
    return self.head_node

def get_sentinel_tail(self):
    return self.tail_node

class AllOne(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """

        self.dll, self.key_counter = DoubleLinkedList(), defaultdict(int)
        self.node_freq = {0:self.dll.get_sentinel_head()}

    def _rmv_key_pf_node(self, pf, key):
        node = self.node_freq[pf]
        node.remove_key(key)
        if node.is_empty():
            self.dll.remove(node)
            self.node_freq.pop(pf)
        return

    def inc(self, key):
        """
        Inserts a new key <Key> with value 1. Or increments an existing key by 1.
        """
```

```
:type key: str
```

```
:rtype: void
```

```
"""
```

```
self.key_counter[key] += 1
```

```
cf, pf = self.key_counter[key], self.key_counter[key]-1
```

```
if cf not in self.node_freq:
```

```
    # No need to test if pf = 0 since frequency zero points to sentinel node
```

```
    self.node_freq[cf] = self.dll.insert_after(self.node_freq[pf])
```

```
self.node_freq[cf].add_key(key)
```

```
if pf > 0:
```

```
    self._rmv_key_pf_node(pf, key)
```

```
def dec(self, key):
```

```
    """
```

Decrements an existing key by 1. If Key's value is 1, remove it from the data structure.

```
:type key: str
```

```
:rtype: void
```

```
"""
```

```
if key in self.key_counter:
```

```
    self.key_counter[key] -= 1
```

```
    cf, pf = self.key_counter[key], self.key_counter[key]+1
```

```
    if self.key_counter[key] == 0:
```

```
        self.key_counter.pop(key)
```

```
    if cf != 0:
```

```
        if cf not in self.node_freq:
```

```

        self.node_freq[cf] = self.dll.insert_before(self.node_freq[pf])
        self.node_freq[cf].add_key(key)
        self._rmv_key_pf_node(pf, key)
def getMaxKey(self):
    """
    Returns one of the keys with maximal value.
    :rtype: str
    """
    return self.dll.get_tail().get_any_key() if self.dll.get_tail().count() > 0 else ""
def getMinKey(self):
    """
    Returns one of the keys with Minimal value.
    :rtype: str
    """
    return self.dll.get_head().get_any_key() if self.dll.get_head().count() > 0 else ""

```

Output :

Accepted Runtime: 54 ms

• Case 1

Input

```
["AllOne", "inc", "inc", "getMaxKey", "getMinKey", "inc", "getMaxKey", "getMinKey"]
```

```
[[], ["hello"], ["hello"], [], [], ["leet"], [], []]
```

Output

```
[null, null, null, "hello", "hello", null, "hello", "leet"]
```

Activate Windows

Result : Sucessfully Executed The Program .

WEEK – 5

Aim : Given an integer n , return the number of structurally unique BST's (binary search trees) which has exactly n nodes of unique values from 1 to n .

Program :

class Solution:

def numTrees(self, n):

return factorial(2*n)//factorial(n)//factorial(n)//(n+1)

Output :

Accepted Runtime: 53 ms

• Case 1 • Case 2

Input

n =
3

Output

5

Result : Successfully Executed The Program.

Aim : A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. Note that the path does not need to pass through the root.

The path sum of a path is the sum of the node's values in the path.

Given the root of a binary tree, return the maximum path sum of any non-empty path.

Program :

class Solution:

```
def maxPathSum(self, root: Optional[TreeNode]) -> int:
```

```
    res = [root.val]
```

```
def max_path(node):
```

```
    if not node:
```

```
        return 0
```

```
    t1 = max(max_path(node.left), 0)
```

```
    t2 = max(max_path(node.right), 0)
```

```
    res[0] = max(res[0], node.val + t1 + t2)
```

```
    return max(t1, t2) + node.val
```

```
max_path(root)
```

```
return res[0]
```

Output :

Accepted

Runtime: 43 ms



• Case 1

• Case 2

Input

```
root =  
[1,2,3]
```

Output

```
6
```

Result : Sucessfully Executed The Program.

WEEK – 6

Aim : Given the root of a binary search tree, and an integer k, return the kth smallest value (1-indexed) of all the values of the nodes in the tree.

Program :

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public int kthSmallest(TreeNode root, int k) {
        ArrayList<Integer> list = new ArrayList<>();
        inorder(root,list);

        if(k>list.size())
            return -1;

        return list.get(k-1);
    }
    public void inorder(TreeNode root, ArrayList<Integer> list)
    {

        if(root!=null)
        {
```



```
inorder(root.left,list);  
list.add(root.val);  
inorder(root.right,list);  
}  
}  
}
```

Output :

The screenshot shows a code execution result interface. At the top, there are tabs for 'Testcase' and 'Result', with 'Result' being the active tab. Below the tabs, the status 'Accepted' is displayed in green, followed by 'Runtime: 0 ms'. There are two tabs for test cases: 'Case 1' (selected) and 'Case 2'. Under the 'Input' section, there are two input fields: 'root =' with the value '[3,1,4,null,2]' and 'k =' with the value '1'. Under the 'Output' section, there is one output field with the value '1'. A watermark 'Activate Windows' is visible in the bottom right corner of the screenshot.

| Testcase | Result |
|----------|---------------------------|
| Case 1 | Accepted Runtime: 0 ms |

Input

root =
[3,1,4,null,2]

k =
1

Output

1

Result : Sucessfully Executed the Program.

Aim : Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:

Search for a node to remove.

If the node is found, delete the node

Program :

class Solution:

```
def deleteNode(self, root: Optional[TreeNode], key: int) ->
```

```
Optional[TreeNode]:
```

```
    if root == None:
```

```
        return root
```

```
    if key < root.val:
```

```
        root.left = self.deleteNode(root.left, key)
```

```
    elif key > root.val:
```

```
        root.right = self.deleteNode(root.right, key)
```

```
    else:
```

```
        if root.left == None and root.right == None:
```

```
            return None
```

```
        if root.left == None:
```

```
            temp = root.right
```

```
            root = None
```

```
            return temp
```

```
        if root.right == None:
```

```
            temp = root.left
```

```
            root = None
```

```
        return temp
    temp = self.minNode(root.right)
    root.val = temp.val
    root.right = self.deleteNode(root.right,temp.val)
    return root
def minNode(self,node):
    curr = node
    while curr.left != None:
        curr = curr.left
    return curr
```

Output :



The screenshot shows a web-based code execution environment. At the top, there are two tabs: 'Testcase' and 'Result', with 'Result' being the active tab. Below the tabs, the status 'Accepted' is displayed in green, followed by 'Runtime: 42 ms'. There are three test cases listed: 'Case 1', 'Case 2', and 'Case 3', all of which are marked as passed with green dots. Below the test cases, the 'Input' section shows two variables: 'root =' with the value '[5,3,6,2,4,null,7]' and 'key =' with the value '3'. The 'Output' section shows the result '[5,4,6,2,null,null,7]'. A watermark 'Activate Windows' is visible in the bottom right corner of the output area.

Testcase Result

Accepted Runtime: 42 ms

• Case 1 • Case 2 • Case 3

Input

root =
[5,3,6,2,4,null,7]

key =
3

Output

[5,4,6,2,null,null,7] Activate Windows

Result : Successfully Executed The Program.

WEEK – 7

Aim : Given a reference of a node in a connected undirected graph.

Return a deep copy (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbours.

```
class Node {  
    public int val;  
    public List<Node> neighbors;  
}
```

Program :

class Solution:

```
def cloneGraph(self, node: 'Node') -> 'Node':
```

```
    if not node: return node
```

```
    q, clones = deque([node]), {node.val: Node(node.val, [])}
```

```
    while q:
```

```
        cur = q.popleft()
```

```
        cur_clone = clones[cur.val]
```

```
        for ngbr in cur.neighbors:
```

```
            if ngbr.val not in clones:
```

```
                clones[ngbr.val] = Node(ngbr.val, [])
```

```
                q.append(ngbr)
```

```
        cur_clone.neighbors.append(clones[ngbr.val])
```

```
return clones[node.val]
```

Output :

Accepted Runtime: 42 ms

• Case 1 • Case 2 • Case 3

Input

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

Output

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

Result : Sucessfully Executed The program.

Aim : There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b , and city b is connected directly with city c , then city a is connected indirectly with city c .

A province is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an $n \times n$ matrix `isConnected` where `isConnected[i][j] = 1` if the i th city and the j th city are directly connected, and `isConnected[i][j] = 0` otherwise.

Return the total number of provinces

Program :

```
from collections import defaultdict
```

```
class Solution:
```

```
    def findCircleNum(self, isConnected: List[List[int]]) -> int:
```

```
        def dfs(node):
```

```
            for neighbor in range(len(isConnected[node])):
```

```
                if isConnected[node][neighbor] and neighbor not in seen:
```

```
                    seen.add(neighbor)
```

```
                    dfs(neighbor)
```

```
        n = len(isConnected)
```

```
        seen = set()
```

```
        ans = 0
```

```
        for i in range(n):
```

```
            if i not in seen:
```

```
                ans += 1
```

```
                seen.add(i)
```

```
                dfs(i)
```

return ans

Output :

Accepted Runtime: 45 ms

• Case 1

• Case 2

Input

```
isConnected =  
[[1,1,0],[1,1,0],[0,0,1]]
```

Output

```
2
```

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

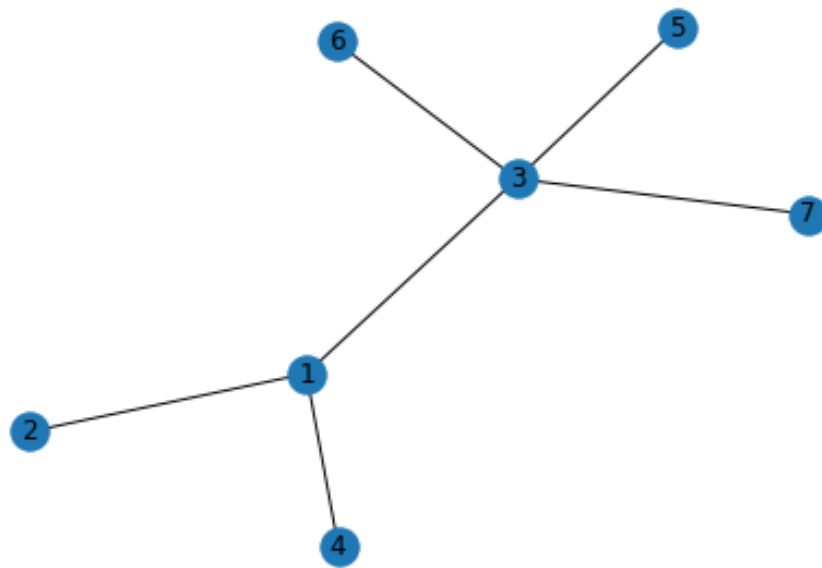
Enter a pair : 5

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

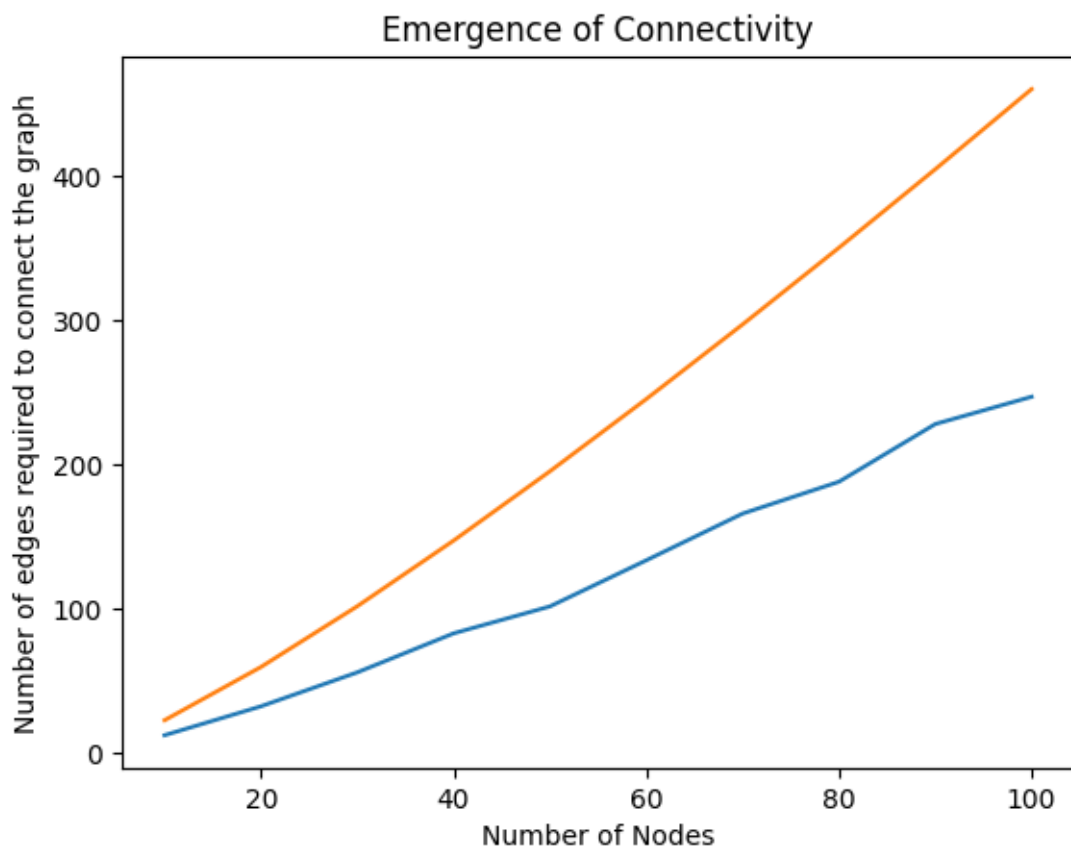
new edge added : [(2, 9)]

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.