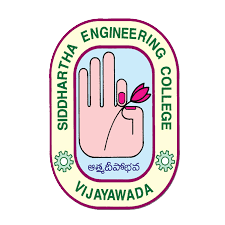
**Velagapudi Ramakrishna Siddhartha**

**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

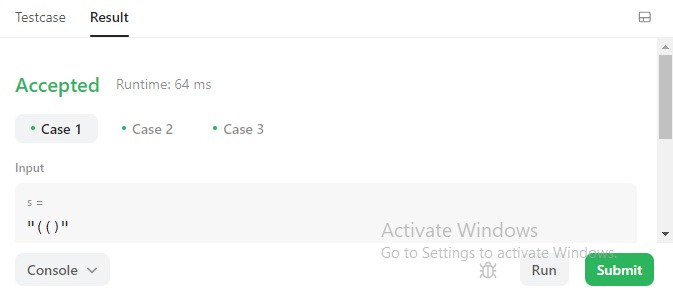
stck.append(i)

else:

max\_length=max(max\_length, i-stck[-1]) # update the length of the valid substring

return max\_length

**Output :**

****

**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);

}

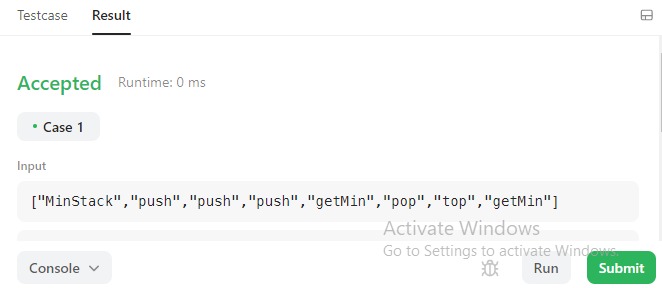
public int getMin() {

return pqueue.peek();

}

}

**Output :**

****

**Result :** Succcessfully 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

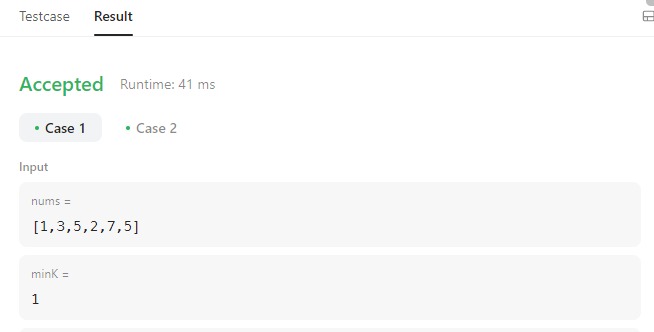
if n == minK: a = i

if n == maxK: b = i

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

return res

**Output :**

****

**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 <= 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):

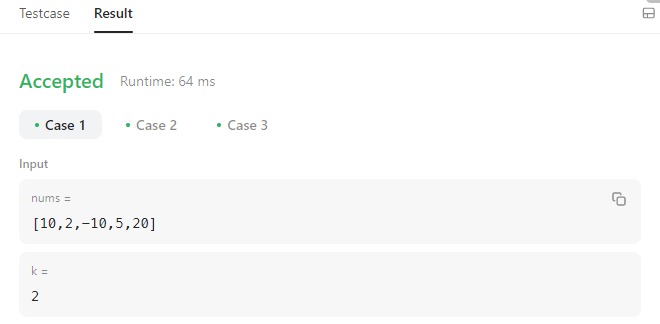
monoq.enqueue(x + max(0, monoq.max()))

if i >= K:

ans = max(ans, monoq.dequeue())

return max(ans, monoq.dequeue())

**Output :**

****

**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\_indx = 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\_indx = i

if min\_indx != -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 mim 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

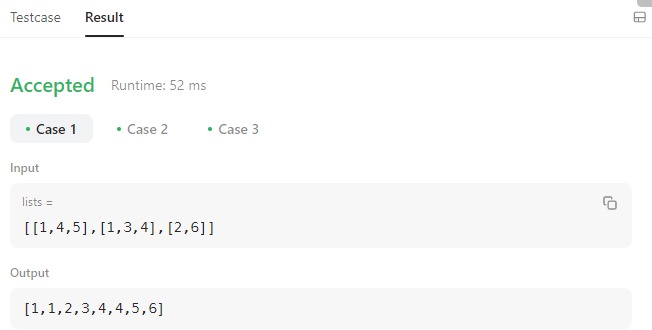
c = ListNode(val=x)

t.next = c

t = t.next

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

**Output :**

****

**Result :** Sucessfully Executed The Program.

**Aim :** Given the head of a singly linked list and two integers left and right where left <= 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

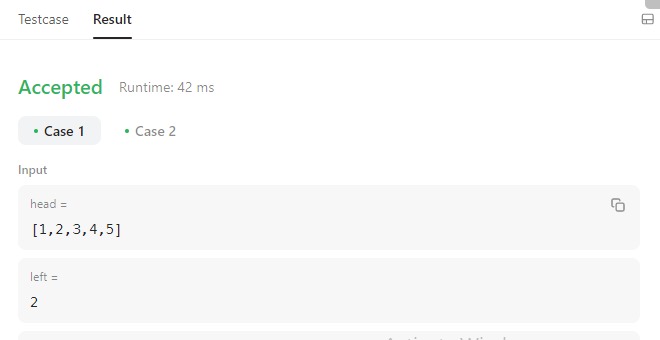
else:

left\_boundary.next = reverse\_head

reverse\_tail.next = right\_boundary

return head

**Output :**



**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\_tail().count() > 0 else ""

**Output :**

****

**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.

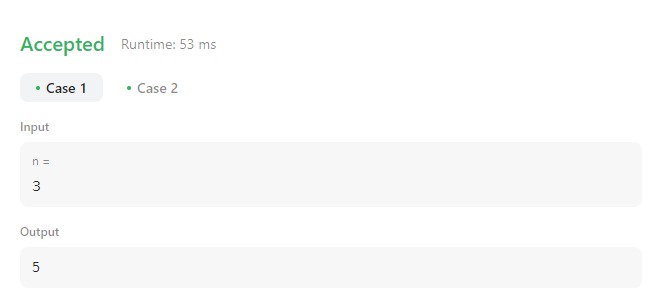
**Program :**

class Solution:

def numTrees(self, n):

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

**Output :**

****

**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)

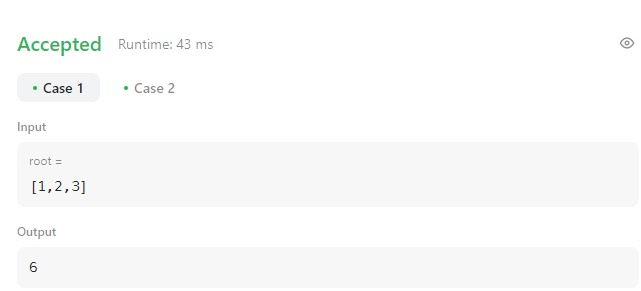
res[0] = max(res[0], node.val + t1 + t2)

return max(t1, t2) + node.val

max\_path(root)

return res[0]

**Output :**

****

**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);

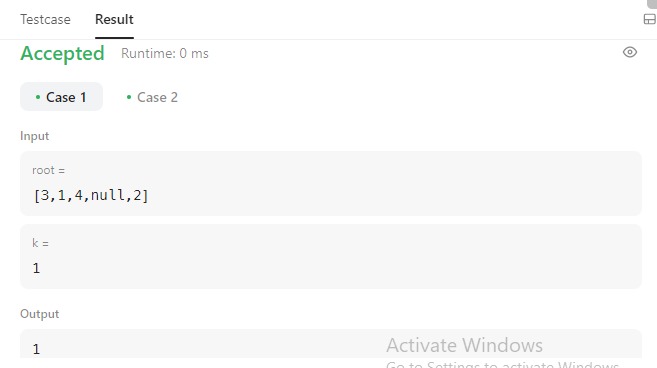
inorder(root.right,list);

}

}

}

**Output :**

****

**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 :**

****

**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 :**

****

**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 x n matrix isConnected where isConnected[i][j] = 1 if the ith city and the jth 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:

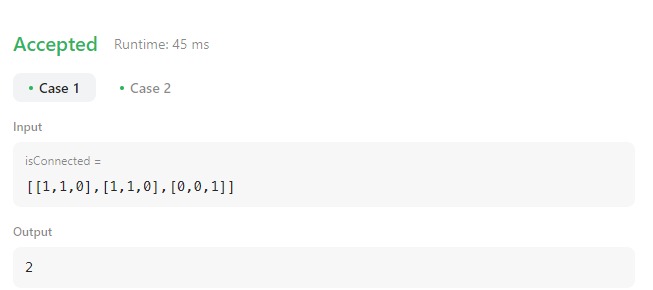
ans += 1

seen.add(i)

dfs(i)

return ans

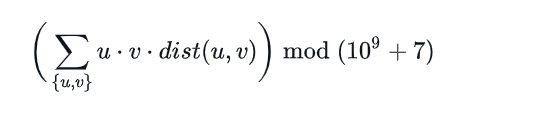
**Output :**

****

**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:



**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

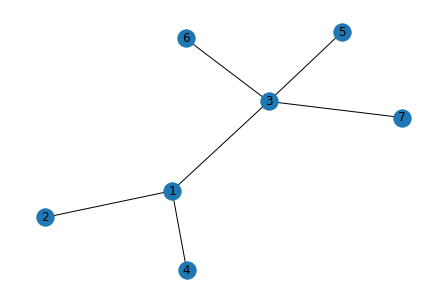
Enter a pair : 5

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 od 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

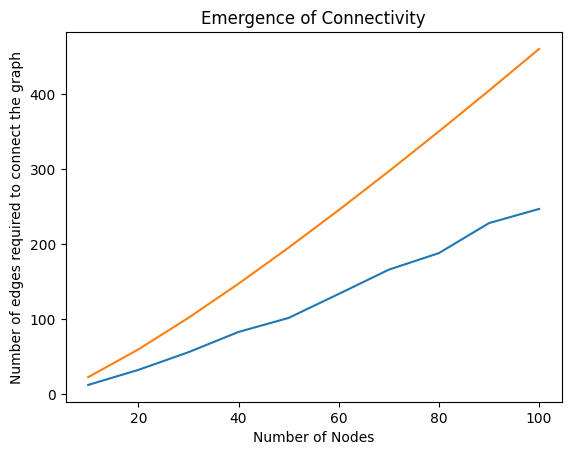
new edge added : [(2, 9)]

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.