# ----------------------------Rec & non-Rec--------------------------------------

import time

# Recursive algorithm

def factorial\_recursive(n):

if n == 0:

return 1

return n \* factorial\_recursive(n-1)

# Non-recursive algorithm

def factorial\_non\_recursive(n):

fact = 1

for i in range(1, n+1):

fact \*= i

return fact

# Test

start\_time = time.time()

print(factorial\_recursive(20)) # 2432902008176640000

print("Recursive algorithm --- %s seconds ---" % (time.time() - start\_time))

start\_time = time.time()

print(factorial\_non\_recursive(20)) # 2432902008176640000

print("Non-recursive algorithm --- %s seconds ---" % (time.time() - start\_time))

# ---------------------------HEAP--------------------------------------------------

def heapify(arr, n, i):

largest = i

l = 2 \* i + 1

r = 2 \* i + 2

if l < n and arr[i] < arr[l]:

largest = l

if r < n and arr[largest] < arr[r]:

largest = r

if largest != i:

arr[i],arr[largest] = arr[largest],arr[i]

heapify(arr, n, largest)

def heapSort(arr):

n = len(arr)

for i in range(n, -1, -1):

heapify(arr, n, i)

for i in range(n-1, 0, -1):

arr[i], arr[0] = arr[0], arr[i]

heapify(arr, i, 0)

arr = [4,8,2,7,19,22,3,5,9]

heapSort(arr)

print("Sorted array is:",arr)

#------------------------ COIN CHANGIND ------------------------------------------

def count(coins, n, sum):

table = [0 for k in range(sum+1)]

table[0] = 1

for i in range(0, n):

for j in range(coins[i], sum+1):

table[j] += table[j-coins[i]]

return table[sum]

coins = [1, 2, 4,5]

n = len(coins)

sum = 8

x = count(coins, n, sum)

print(x)

# ----------------------------WARSHALL-------------------------------------------------

def transitive\_closure(graph):

V = len(graph)

reach = [[False for i in range(V)] for j in range(V)]

for i in range(V):

for j in graph[i]:

reach[i][j] = True

for k in range(V):

for i in range(V):

for j in range(V):

reach[i][j] = reach[i][j] or (reach[i][k] and reach[k][j])

return reach

graph = [[1, 2, 3], [0, 2, 3], [0, 1, 3], [0, 1, 2]]

print(transitive\_closure(graph))

# ------------------------------------Knapsack-------------------------------------------

def knapsack(w,wt,val,n):

if n==0 or w==0:

return 0

if (wt[n-1]>w):

return knapsack (w,wt,val,n-1)

else:

return max (val[n-1]+knapsack(w-wt[n-1],wt,val,n-1),knapsack(w,wt,val,n-1))

val=[18,16,6]

wt=[1,2,2]

w=4

n=len(val)

print(knapsack(w,wt,val,n))

# ------------------------------Assignment problem--------------------------------

from itertools import permutations

from collections import defaultdict

cost\_matrix = [[10, 11, 5, 7], [6, 8, 9, 5], [5, 7, 10, 8], [7, 6, 8, 12]]

permutations\_list = list(permutations(range(len(cost\_matrix))))

def calculate\_cost(assignment):

total\_cost = 0

for i, job in enumerate(assignment):

total\_cost += cost\_matrix[i][job]

return total\_cost

min\_cost = defaultdict(int)

for i in range(len(cost\_matrix[0])):

min\_cost[i] = float('inf')

for permutation in permutations\_list:

cost = calculate\_cost(permutation)

for i, job in enumerate(permutation):

if cost < min\_cost[job]:

min\_cost[job] = cost

best\_assignment = None

best\_cost = float('inf')

for permutation in permutations\_list:

valid = True

for i, job in enumerate(permutation):

if cost\_matrix[i][job] > min\_cost[job]:

valid = False

break

if valid and calculate\_cost(permutation) < best\_cost:

best\_assignment = permutation

best\_cost = calculate\_cost(permutation)

print("Best assignment:", best\_assignment)

print("Best cost:", best\_cost)

# ------------------------------------------------Hoffman------------------------------------

import heapq

class Node:

def \_\_init\_\_(self, char, prob):

self.char = char

self.prob = prob

self.left = None

self.right = None

def \_\_lt\_\_(self, other):

return self.prob < other.prob

def build\_huffman\_tree(characters, probabilities):

nodes = [Node(char, prob) for char, prob in zip(characters, probabilities)]

heapq.heapify(nodes)

while len(nodes) > 1:

left = heapq.heappop(nodes)

right = heapq.heappop(nodes)

parent = Node(None, left.prob + right.prob)

parent.left = left

parent.right = right

heapq.heappush(nodes, parent)

return nodes[0]

def get\_codes(node, code, codes):

if node is None:

return

if node.char is not None:

codes[node.char] = code

return

get\_codes(node.left, code + "0", codes)

get\_codes(node.right, code + "1", codes)

def main():

characters = ['A','B','C','D','E','-']

probabilities = [0.1,0.5,0.35,0.5,0.4,0.2]

root = build\_huffman\_tree(characters, probabilities)

codes = {}

get\_codes(root, "", codes)

print(codes)

if \_\_name\_\_ == "\_\_main\_\_":

main()

# ----------------------------------------------SIMPLEX---------------------------------------

from scipy.optimize import linprog

c = [-63, -90] # objective function coefficients

A = [[110, 40], [60, 110]] # constraint coefficients

b = [20200, 30000] # constraint limits

x0\_bounds = (0, None) # non-negativity constraints

x1\_bounds = (0, None)

res = linprog(c, A\_ub=A, b\_ub=b, bounds=[x0\_bounds, x1\_bounds], method='simplex')

print("Optimal number of acres of corn: ", res.x[0])

print("Optimal number of acres of soybeans: ", res.x[1])

print("Optimal profit: ", -res.fun)

# ----------------------------------------------N-Queen--------------------------------------------

def is\_safe(board, row, col, n):

for x in range(col):

if board[row][x] == 1:

return False

for x, y in zip(range(row, -1, -1), range(col, -1, -1)):

if board[x][y] == 1:

return False

for x, y in zip(range(row, n, 1), range(col, -1, -1)):

if board[x][y] == 1:

return False

return True

def solve(board, col, n):

if col == n:

return True

for i in range(n):

if is\_safe(board, i, col, n):

board[i][col] = 1

if solve(board, col+1, n):

return True

board[i][col] = 0

return False

def n\_queen(n):

board = [[0 for i in range(n)] for j in range(n)]

if not solve(board, 0, n):

print("Solution does not exist")

else:

for i in range(n):

print(board[i])

n\_queen(5) # You can replace the number 5 with desired number of queens

# ----------------------------------------------TOPOLOGICAL SORTING ----------------------------

# A Python program to print topological sorting of a DAG

from collections import defaultdict

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = defaultdict(list)

def addEdge(self, v, w):

self.graph[v].append(w)

def topologicalSortUtil(self, v, visited, stack):

visited[v] = True

if v in self.graph.keys():

for neighbour in self.graph[v]:

if visited[neighbour] == False:

self.topologicalSortUtil(neighbour, visited, stack)

stack.append(v)

def topologicalSort(self):

visited = [False]\*self.V

stack = []

for i in range(self.V):

if visited[i] == False:

self.topologicalSortUtil(i, visited, stack)

print("Topological Sort of the given graph:")

while stack:

print(stack.pop(), end=' ')

n = int(input("Enter the number of edges: "))

g = Graph(n)

for i in range(n):

v1, v2 = map(int, input().split())

g.addEdge(v1, v2)

g.topologicalSort()