DAA--1 Program for Fibonacci numbers

#function to implement Iterative approach

def IterativeFibo(n):

f1=0

f2=1

for i in range(n):

if i< 2:

print(i, end=' ')

else:

f3= f1+f2

f1=f2

f2=f3

print(f3,end = ' ')

#function to implement recursive approach

def RecursiveFibo(n):

if (n==0 or n==1):

return n

else:

return ( RecursiveFibo(n-1)+ RecursiveFibo(n-2))

def main():

n=10

print("ITERATIVE FIBONACCI: ")

IterativeFibo(n)

print("\n RECURSIVE FIBONACCI: ")

for i in range(n):

print(RecursiveFibo(i),end = ' ')

if \_\_name\_\_== '\_\_main\_\_':

main()

DAA--2 Huffman Encoding

# A Huffman Tree Node

import heapq

class node:

def \_\_init\_\_(self, freq, symbol, left=None, right=None):

# frequency of symbol

self.freq = freq

# symbol name (character)

self.symbol = symbol

# node left of current node

self.left = left

# node right of current node

self.right = right

# tree direction (0/1)

self.huff = ''

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

return self.freq < nxt.freq

# utility function to print huffman

# codes for all symbols in the newly

# created Huffman tree

def printNodes(node, val=''):

# huffman code for current node

newVal = val + str(node.huff)

# if node is not an edge node

# then traverse inside it

if(node.left):

printNodes(node.left, newVal)

if(node.right):

printNodes(node.right, newVal)

# if node is edge node then

# display its huffman code

if(not node.left and not node.right):

print(f"{node.symbol} -> {newVal}")

# characters for huffman tree

chars = ['a', 'b', 'c', 'd', 'e', 'f']

# frequency of characters

freq = [ 5, 9, 12, 13, 16, 45]

# list containing unused nodes

nodes = []

# converting characters and frequencies

# into huffman tree nodes

for x in range(len(chars)):

heapq.heappush(nodes, node(freq[x], chars[x]))

while len(nodes) > 1:

# sort all the nodes in ascending order

# based on their frequency

left = heapq.heappop(nodes)

right = heapq.heappop(nodes)

# assign directional value to these nodes

left.huff = 0

right.huff = 1

# combine the 2 smallest nodes to create

# new node as their parent

newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)

heapq.heappush(nodes, newNode)

# Huffman Tree is ready!

printNodes(nodes[0])

DAA---3 Fractional Knapsack Problem

# Structure for an item which stores weight and

# corresponding value of Item

class Item:

def \_\_init\_\_(self, value, weight):

self.value = value

self.weight = weight

# Main greedy function to solve problem

def fractionalKnapsack(W, arr):

# Sorting Item on basis of ratio

arr.sort(key=lambda x: (x.value/x.weight), reverse=True)

# Result(value in Knapsack)

finalvalue = 0.0

# Looping through all Items

for item in arr:

# If adding Item won't overflow,

# add it completely

if item.weight <= W:

W -= item.weight

finalvalue += item.value

# If we can't add current Item,

# add fractional part of it

else:

finalvalue += item.value \* W / item.weight

break

# Returning final value

return finalvalue

# Driver Code

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

W = 50

arr = [Item(60, 10), Item(100, 20), Item(120, 30)]

# Function call

max\_val = fractionalKnapsack(W, arr)

print(max\_val)

DAA--4 program to solve a o-1 knapsack problem using dynamic

or branch and bound strategy

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

k = [[0 for x in range (w+1)] for x in range (n+1)]

for i in range(n+1):

for w in range(w+1):

if i == 0 or w == 0:

k[i][w] = 0

elif wt[i-1] <=w:

k[i][w] = max(val[i-1] + k[i-1][w-wt[i-1]], k[i-1][w])

else:

k[i][w]= k[i-1][w]

return k[n][w]

def InputList():

lst = []

n = int(input("Enter number of Element:"))

for i in range(0, n):

ele = int(input())

lst.append(ele)

return lst

#Driver code

#val = [60, 100, 120]

val= InputList()

#wt = [10, 20, 30]

wt = InputList()

#w=50

w=int(input("Enter the capacity:"))

n = len(val)

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

DAA--5 N Queen Problem

# Python3 program to solve N Queen

# Problem using backtracking

global N

N = int(input())

def printSolution(board):

for i in range(N):

for j in range(N):

print(board[i][j], end = " ")

print()

# A utility function to check if a queen can

# be placed on board[row][col]. Note that this

# function is called when "col" queens are

# already placed in columns from 0 to col -1.

# So we need to check only left side for

# attacking queens

def isSafe(board, row, col):

# Check this row on left side

for i in range(col):

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

return False

# Check upper diagonal on left side

for i, j in zip(range(row, -1, -1),

range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check lower diagonal on left side

for i, j in zip(range(row, N, 1),

range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solveNQUtil(board, col):

# base case: If all queens are placed

# then return true

if col >= N:

return True

# Consider this column and try placing

# this queen in all rows one by one

for i in range(N):

if isSafe(board, i, col):

# Place this queen in board[i][col]

board[i][col] = 1

# recur to place rest of the queens

if solveNQUtil(board, col + 1) == True:

return True

# If placing queen in board[i][col

# doesn't lead to a solution, then

# queen from board[i][col]

board[i][col] = 0

# if the queen can not be placed in any row in

# this column col then return false

return False

# This function solves the N Queen problem using

# Backtracking. It mainly uses solveNQUtil() to

# solve the problem. It returns false if queens

# cannot be placed, otherwise return true and

# placement of queens in the form of 1s.

# note that there may be more than one

# solutions, this function prints one of the

# feasible solutions.

def solveNQ():

'''board = [ [0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0] ]'''

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

if solveNQUtil(board, 0) == False:

print ("Solution does not exist")

return False

printSolution(board)

return True

# Driver Code

solveNQ()