#Assignment 1

#Recursive

def recur\_fibo(n):

if n<=1:

return n

else:

return(recur\_fibo(n-1) + recur\_fibo(n-2))

nterms = 10

# check if the number of terms is valid

if nterms <= 0:

print("Plese enter a positive integer")

else:

print("Fibonacci sequence:")

for i in range(nterms):

print(recur\_fibo(i))

#non-recursive

def fib(n):

if n == 1:

return [1]

if n == 2:

return [1, 1]

fibs = [1, 1]

for \_ in range(2, n):

fibs.append(fibs[-1] + fibs[-2])

return fibs

print(fib(3))

#Assignment 2

# Huffman Coding in python

string = 'BCAADDDCCACACAC'

# Creating tree nodes

class NodeTree(object):

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

self.left = left

self.right = right

def children(self):

return (self.left, self.right)

def nodes(self):

return (self.left, self.right)

def \_\_str\_\_(self):

return '%s\_%s' % (self.left, self.right)

# Main function implementing huffman coding

def huffman\_code\_tree(node, left=True, binString=''):

if type(node) is str:

return {node: binString}

(l, r) = node.children()

d = dict()

d.update(huffman\_code\_tree(l, True, binString + '0'))

d.update(huffman\_code\_tree(r, False, binString + '1'))

return d

# Calculating frequency

freq = {}

for c in string:

if c in freq:

freq[c] += 1

else:

freq[c] = 1

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

nodes = freq

while len(nodes) > 1:

(key1, c1) = nodes[-1]

(key2, c2) = nodes[-2]

nodes = nodes[:-2]

node = NodeTree(key1, key2)

nodes.append((node, c1 + c2))

nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman\_code\_tree(nodes[0][0])

print(' Char | Huffman code ')

print('----------------------')

for (char, frequency) in freq:

print(' %-4r |%12s' % (char, huffmanCode[char]))

#Assignment 3

# 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)

#Assignment 4

# A naive recursive implementation

# of 0-1 Knapsack Problem

# Returns the maximum value that

# can be put in a knapsack of

# capacity W

def knapSack(W, wt, val, n):

# Base Case

if n == 0 or W == 0:

return 0

# If weight of the nth item is

# more than Knapsack of capacity W,

# then this item cannot be included

# in the optimal solution

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

return knapSack(W, wt, val, n-1)

# return the maximum of two cases:

# (1) nth item included

# (2) not included

else:

return max(

val[n-1] + knapSack(

W-wt[n-1], wt, val, n-1),

knapSack(W, wt, val, n-1))

# end of function knapSack

#Driver Code

val= [60, 100, 120]

wt = [10, 20, 30]

W=50

n=len(val)

print(knapSack(W, wt, val, n))

#Assignment 5

# Python3 program to solve N Queen

# Problem using backtracking

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

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

print()

def isSafe(board, row, col):

for i in range(col):

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

return False

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

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

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

return False

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):

if col >= N:

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

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

return True

board[i][col] = 0

return False

def solveNQ():

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

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0] ]

if solveNQUtil(board, 0) == False:

print ("Solution does not exist")

return False

printSolution(board)

return True

solveNQ()