1 Iterative approach

nterm=int(input("Enter Number : "))

n1,n2=0,1

count=0

if (nterm<=0):

print("Enter Positive")

elif (nterm==1):

print("Series Of ",nterm," :")

print(n1)

else:

print("Series")

while count<nterm:

print(n1)

nth=n1+n2

n1=n2

n2=nth

count+=1

Recursive approach

def fibonacci(n):

if (n<=1):

return n

else:

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

n=int(input("Enter : "))

for i in range (n):

print(fibonacci(i))

2 Huffman Encoding using greedy approach

class node:

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

self.freq = freq

self.symbol = symbol

self.left = left

self.right = right

self.huff = ''

def printNodes(node, val=''):

newVal = val + str(node.huff)

if(node.left):

printNodes(node.left, newVal)

if(node.right):

printNodes(node.right, newVal)

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

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

# characters for huffman tree

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

freq = [ 4, 7, 12, 14, 17, 43, 54]

"""

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

freq = [ 5, 9, 12, 13, 8, 14]

"""

nodes = []

for x in range(len(chars)):

nodes.append(node(freq[x], chars[x]))

while len(nodes) > 1:

nodes = sorted(nodes, key=lambda x: x.freq)

left = nodes[0]

right = nodes[1]

left.huff = 0

right.huff = 1

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

nodes.remove(left)

nodes.remove(right)

nodes.append(newNode)

printNodes(nodes[0])

3 Fractional Knapsack Problem using greedy approach

def fractional\_knapsack(value, weight, capacity):

index = list(range(len(value)))

ratio = [v/w for v, w in zip(value, weight)]

index.sort(key=lambda i: ratio[i], reverse=True)

max\_value = 0

fractions = [0]\*len(value)

for i in index:

if weight[i] <= capacity:

fractions[i] = 1

max\_value += value[i]

capacity -= weight[i]

else:

fractions[i] = capacity/weight[i]

max\_value += value[i]\*capacity/weight[i]

break

return max\_value, fractions

n = int(input('Enter number of items: '))

value = input('Enter the values of the {} item(s) in order: '.format(n)).split()

value = [int(v) for v in value]

weight = input('Enter the positive weights of the {} item(s) in order:'.format(n)).split()

weight = [int(w) for w in weight]

capacity = int(input('Enter maximum weight: '))

max\_value,fractions=fractional\_knapsack(value, weight, capacity)

print('The maximum value of items that can be carried:', max\_value)

print('The fractions in which the items should be taken:', fractions)

4 Program for Queen Matrix 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()