Fibonacci series

**Recursion**

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 Recursion

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(7))

**Huffman**

# 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])

**knapsack fractional knapsack greedy**

items = [('A',60,10),('B',100,20),('C',120,30), ('D', 1000, 10)]

max\_wt = 50

def fractional\_knapsack(items, max\_wt): ratios = []

for i in items:

ratios.append((i[0], i[1], i[2], i[1]/i[2])) ratios.sort(key=lambda tup: tup[3], reverse = True) ans = {}

optimal\_profit = 0 for item in ratios:

if(item[2] < max\_wt): max\_wt -= item[2] ans[item[0]]=1

optimal\_profit += item[1] else:

fraction = max\_wt/item[2] ans[item[0]] = fraction max\_wt = 0

optimal\_profit += fraction\*item[1] break

return {"ans":ans, "optimal\_profit":optimal\_profit}

print(fractional\_knapsack(items, max\_wt))

**knspsack dynamic programming**

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

K = [[0 for x in range(W + 1)] for x in range(n + 1)] # Build table K[][] in bottom up manner

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]

# Driver program to test above function val = [60, 100, 120]

wt = [10, 20, 30]

W = 50

n = len(val)

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

**N-Quuen**

class QueenChessBoard:

    def \_\_init\_\_(self, size):

# board has dimensions size x size

        self.size = size

        # columns[r] is a number c if a queen is placed at row r and column c.

        # columns[r] is out of range if no queen is place in row r.

        # Thus after all queens are placed, they will be at positions

        # (columns[0], 0), (columns[1], 1), ... (columns[size - 1], size - 1)

        self.columns = []

    def place\_in\_next\_row(self, column):

        self.columns.append(column)

    def remove\_in\_current\_row(self):

        return self.columns.pop()

    def is\_this\_column\_safe\_in\_next\_row(self, column):

# index of next row

        row = len(self.columns)

        # check column

        for queen\_column in self.columns:

            if column == queen\_column:

                return False

        # check diagonal

        for queen\_row, queen\_column in enumerate(self.columns):

            if queen\_column - queen\_row == column - row:

                return False

        # check other diagonal

        for queen\_row, queen\_column in enumerate(self.columns):

            if ((self.size - queen\_column) - queen\_row== (self.size - column) - row):

                return False

        return True

    def display(self):

        for row in range(self.size):

            for column in range(self.size):

                if column == self.columns[row]:

                    print('Q', end=' ')

                else:

                    print('.', end=' ')

            print()

    def solve\_queen(size):

        board = QueenChessBoard(size)

        number\_of\_solutions = 0

        row = 0

        column = 0

        # iterate over rows of board

        while True:

        # place queen in next row

            while column < size:

                if board.is\_this\_column\_safe\_in\_next\_row(column):

                    board.place\_in\_next\_row(column)

                    row += 1

                    column = 0

                    break

                else:

                    column += 1

            if (column == size or row == size):

                if row == size:

                    board.display()

                    print()

                    number\_of\_solutions += 1

                    board.remove\_in\_current\_row()

                    row -= 1

        # now backtrack

                try:

                    prev\_column = board.remove\_in\_current\_row()

                except IndexError:

                 row -= 1

        # start checking at column = (1 + value of column in previous row)

                column = 1 + prev\_column

            print('Number of solutions:', number\_of\_solutions)

    n = int(input('Enter n: '))

    solve\_queen(n)