**Fibonacci series**

# Program to display the Fibonacci sequence up to n-th term

nterms = int(input("How many terms? "))

# first two terms

n1, n2 = 0, 1

count = 0

# check if the number of terms is valid

if nterms <= 0:

print("Please enter a positive integer")

# if there is only one term, return n1

elif nterms == 1:

print("Fibonacci sequence upto",nterms,":")

print(n1)

# generate fibonacci sequence

else:

print("Fibonacci sequence:")

while count < nterms:

print(n1)

nth = n1 + n2

# update values

n1 = n2

n2 = nth

count += 1

**Fibonacci series using recursion**

# Python program to display the Fibonacci sequence

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))

**# 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])

**Fractional Knapsack Problem**

# Structure for an item which stores weight and

# corresponding value of Item

class Item:

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

self.profit = profit

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.profit/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.profit

# If we can't add current Item,

# add fractional part of it

else:

finalvalue += item.profit \* 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)

**0/1 knapsack using Branch and Bound**

from queue import Queue

from typing import List

class KnapsackNode:

def \_\_init\_\_(self, items: List[int], value: int, weight: int):

self.items = items

self.value = value

self.weight = weight

class Item:

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

self.value = value

self.weight = weight

self.ratio = value / weight

class Knapsack:

def \_\_init\_\_(self, maxWeight: int, items: List[Item]):

self.maxWeight = maxWeight

self.items = items

def solve(self) -> int:

self.items.sort(key=lambda x: x.ratio, reverse=True)

bestValue = 0

queue = [KnapsackNode([], 0, 0)]

while queue:

node = queue.pop(0)

i = len(node.items)

if i == len(self.items):

bestValue = max(bestValue, node.value)

else:

item = self.items[i]

withItem = KnapsackNode(

node.items + [i],

node.value + item.value,

node.weight + item.weight

)

if self.isPromising(withItem, self.maxWeight, bestValue):

queue.append(withItem)

withoutItem = KnapsackNode(

node.items,

node.value,

node.weight

)

if self.isPromising(withoutItem, self.maxWeight, bestValue):

queue.append(withoutItem)

return bestValue

def isPromising(self, node: KnapsackNode, maxWeight: int, bestValue: int) -> bool:

return node.weight <= maxWeight and node.value + self.getBound(node) > bestValue

def getBound(self, node: KnapsackNode) -> float:

remainingWeight = self.maxWeight - node.weight

bound = node.value

for i in range(len(node.items), len(self.items)):

item = self.items[i]

if remainingWeight >= item.weight:

bound += item.value

remainingWeight -= item.weight

else:

bound += remainingWeight \* item.ratio

break

return bound

**0/1 Knapsack Problem using Dyanamic Programming**

# This is the memoization approach of

# 0 / 1 Knapsack in Python in simple

# we can say recursion + memoization = DP

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

# base conditions

if n == 0 or W == 0:

return 0

if t[n][W] != -1:

return t[n][W]

# choice diagram code

if wt[n-1] <= W:

t[n][W] = max(

val[n-1] + knapsack(

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

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

return t[n][W]

elif wt[n-1] > W:

t[n][W] = knapsack(wt, val, W, n-1)

return t[n][W]

# Driver code

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

profit = [60, 100, 120]

weight = [10, 20, 30]

W = 50

n = len(profit)

# We initialize the matrix with -1 at first.

t = [[-1 for i in range(W + 1)] for j in range(n + 1)]

print(knapsack(weight, profit, W, n))

# This code is contributed by Prosun Kumar Sarkar

**N queen problem**

print("Enter the number of queens")

N = int(input())

board = [[0]\*N for \_ in range(N)]

def is\_attack(i, j):

    for k in range(0,N):

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

            return True

    for k in range(0, N):

        for l in range(0,N):

            if (k+l==i+j) or (k-l==i-j):

                if board[k][l]==1:

                    return True

    return False

def N\_Queen(n):

    if n==0:

        return True

    for i in range(0,N):

        for j in range(0,N):

            if (not(is\_attack(i,j))) and (board[i][j] != 1):

                board[i][j] = 1

                if N\_Queen(n-1) == True:

                    return True

                board[i][j] = 0

    return False

N\_Queen(N)

for i in board:

    print(i)