**DAA**

**Practical no. 1**

# Non-recursive Fib

a = int(input('Enter first num : '))

b = int(input('Enter second num : '))

n = int(input('Enter num of terms : '))

print(a,b,end=' ')

while(n-2):

c = a + b

a = b

b = c

print(c,end=' ')

n = n-1

#Recursive-Fib

def fib(n):

if n<=1:

return 1

else:

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

n = int(input("Enter the no. of terms : "))

for i in range(n):

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

**practical 2**

import heapq

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

return self.freq < nxt.freq

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}")

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

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

nodes = []

for x in range(len(chars)):

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

while len(nodes) > 1:

left = heapq.heappop(nodes)

right = heapq.heappop(nodes)

left.huff = 0

right.huff = 1

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

heapq.heappush(nodes, newNode)

printNodes(nodes[0])

**Practical 3**

class Item:

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

self.profit = profit

self.weight = weight

def fractionalKnapsack(W, arr):

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

finalvalue = 0.0

for item in arr:

if item.weight <= W:

W -= item.weight

finalvalue += item.profit

else:

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

break

return finalvalue

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

W = 50

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

max\_val = fractionalKnapsack(W, arr)

print(max\_val)

**practical 4**

def knapsack\_dynamic\_programming(weights, values, capacity):

n = len(weights)

dp = [0] \* (capacity + 1)

for i in range(n):

for w in range(capacity, 0, -1):

if weights[i] <= w:

dp[w] = max(dp[w], dp[w - weights[i]] + values[i])

return dp[capacity]

# Example usage:

weights = [2, 2, 3, 4, 5]

values = [3, 4, 5, 8, 10]

capacity = 9

max\_value = knapsack\_dynamic\_programming(weights, values, capacity)

print("Maximum value:", max\_value)

**practical 5**

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

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

print("Q",end=" ")

else:

print(".",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

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

solveNQ()

**practical 6**

def quick(sequence):

length = len(sequence)

if length<=1:

return sequence

else:

pivot = sequence.pop()

item\_greater = []

item\_lower = []

for item in sequence:

if item>pivot:

item\_greater.append(item)

else:

item\_lower.append(item)

return quick(item\_lower) + [pivot] + quick(item\_greater)

print(quick([33,45,33,67,1,767,87,4,33,2,0]))