**Assignment : 01**

**Title : Write a program to calculate Fibonacci numbers and find its step count**

**Roll no: CO427**

**Program :**

1. **Non-Recursive**

def fibonacci(n):

"""Return the nth Fibonacci number.""" if n == 0:

return 0

# r[i] will contain the ith Fibonacci number r = [-1] \* (n + 1)

r[0] = 0

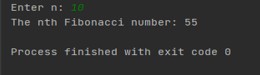
r[1] = 1

for i in range(2, n + 1): r[i] = r[i - 1] + r[i - 2]

return r[n]

n = int(input('Enter n: ')) ans = fibonacci(n)

print('The nth Fibonacci number:', ans)

**OUTPUT**

1. **Recursive**

# 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")

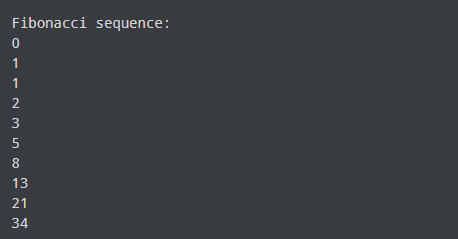
else:

print("Fibonacci sequence:")

for i in range(nterms):

print(recur\_fibo(i))

**OUTPUT**



**Assignment : 02**

**Title : Write program to implement Huffman Encoding using a greed strategy.**

**Roll no: CO427**

**Program :**

import heapq class node:

def init (self,freq,symbol,left=None,right=None): self.freq=freq

self.symbol=symbo lself.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("{} -> {}".format(node.symbol,newval))

if name ==" main ": chars = ['a', 'b', 'c', 'd', 'e', 'f'] freq = [ 5, 9, 12, 13, 16, 45]

nodes=[]

for i in range(len(chars)): heapq.heappush(nodes, node(freq[i],chars[i]))

while len(nodes)>1: left=heapq.heappop(nodes) right=heapq.heappop(nodes)left.huff

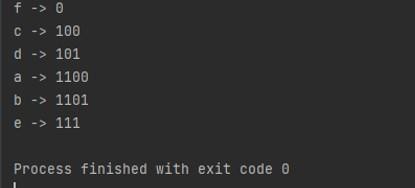
= 0

right.huff = 1

newnode = node(left.freq + right.freq , left.symbol + right.symbol , left , right) heapq.heappush(nodes, newnode)

printnodes(nodes[0])

OUTPUT :



**Assignment : 03**

**Title : Write a program to solve a fractional Knapsack problem using a greedy method.**

**Roll no: CO427**

**Program :**

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]

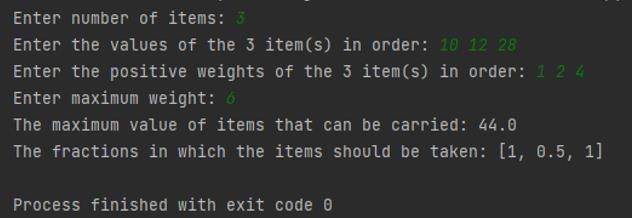
weight = input('Enter the positive weights of the {} item(s) in order: '

.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)

OUTPUT:



# Assignment:04

**Title : Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch andbound strategy.**

**Rollno:CO427**

**Program :**

def knapsack(value, weight, capacity): n = len(value) - 1

m = [[-1] \* (capacity + 1) for \_ in range(n + 1)] for w in range(capacity + 1):

m[0][w] = 0

for i in range(1, n + 1):

for w in range(capacity + 1):

if weight[i] > w:

m[i][w] = m[i - 1][w]else:

m[i][w] = max(m[i - 1][w - weight[i]] + value[i], m[i - 1][w])

return m[n][capacity]

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]

value.insert(0, None) # so that the value of the ith item is at value[i] weight = input('Enter the positive weights of the {} item(s) in order: '

.format(n)).split() weight = [int(w) for w in weight]

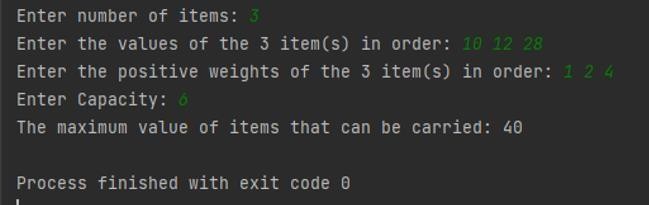
weight.insert(0, None) # so that the weight of the ith item is at weight[i]capacity

= int(input('Enter Capacity: '))

ans = knapsack(value, weight, capacity)

print('The maximum value of items that can be carried:', ans)

OUTPUT



# Assignment:05

**Title : Design n Queens Matrix having First Queen placed using backtracking to place remaining Queens to generate the final n Queens Matrix.**

**Rollno:CO427**

**Program :**

N = 8 # Size of the chessboard def is\_safe(board, row, col):

# Check if there is a Queen in the same column for i in range(row):

if board[i][col] == 1: return False

# Check upper left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)): if board[i][j] == 1:

return False

# Check upper right diagonal

for i, j in zip(range(row, -1, -1), range(col, N)): if board[i][j] == 1:

return False return True

def solve\_n\_queens(board, row): if row >= N:

return True

for col in range(N):

if is\_safe(board, row, col): board[row][col] = 1 # Place the Queen

if solve\_n\_queens(board, row + 1): # Recur to place rest of the Queens return True

board[row][col] = 0 # If placing Queen doesn't lead to a solution, backtrack return False

# Initialize the chessboard with the first Queen already placed chessboard = [[0 for \_ in range(N)] for \_ in range(N)] chessboard[0][0] = 1

# Solve the 8-Queens problem using backtracking if solve\_n\_queens(chessboard, 1):

# Print the solution for row in chessboard:

print(' '.join(['Q' if x else '.' for x in row]))

else:

print("No solution exists.")

**OUTPUT**

