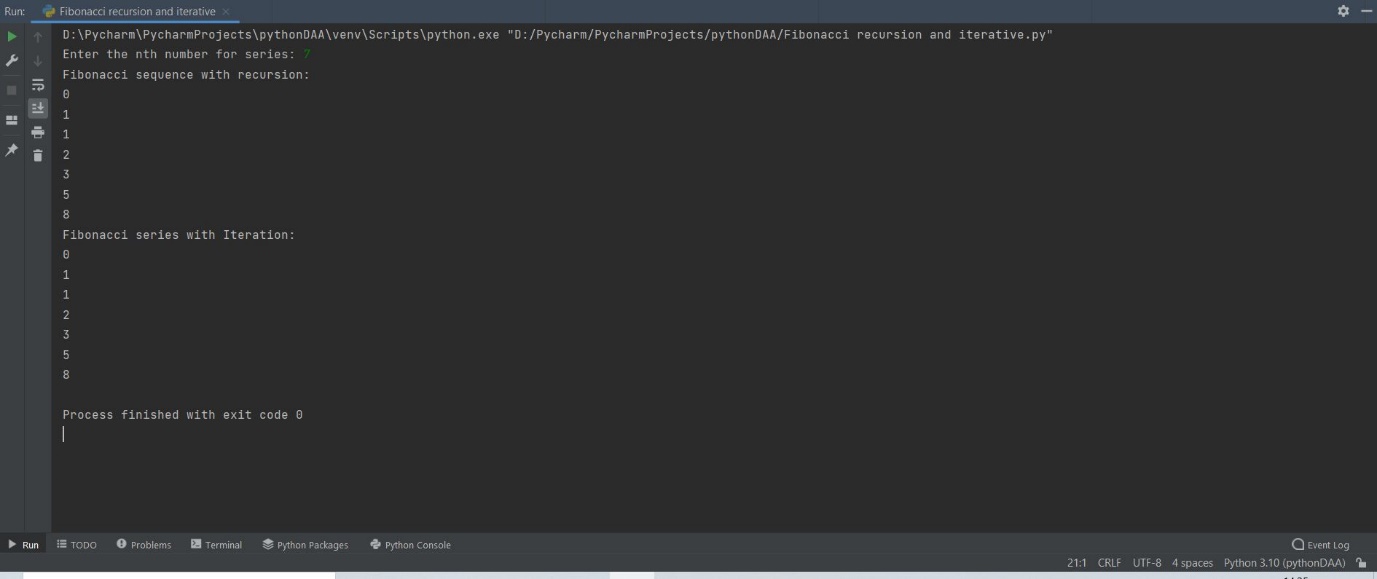
Q1.) Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

Program:

def recur(n):  
 if n <= 1:  
 return n  
  
 else:  
 return(recur(n-1) + recur(n-2))  
  
def iterative(n):  
 a = 0  
 b = 1  
  
 print(a)  
 print(b)  
  
 for i in range(2, n):  
 print(a + b)  
 a, b = b, a + b  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 num = int(input("Enter the nth number for series: "))  
  
 if num <= 0:  
 print("Please enter a positive integer")  
  
 else:  
 print("Fibonacci sequence with recursion:")  
  
 for i in range(num):  
 print(recur(i))  
  
 print("Fibonacci series with Iteration:")  
 iterative(num)

Output:

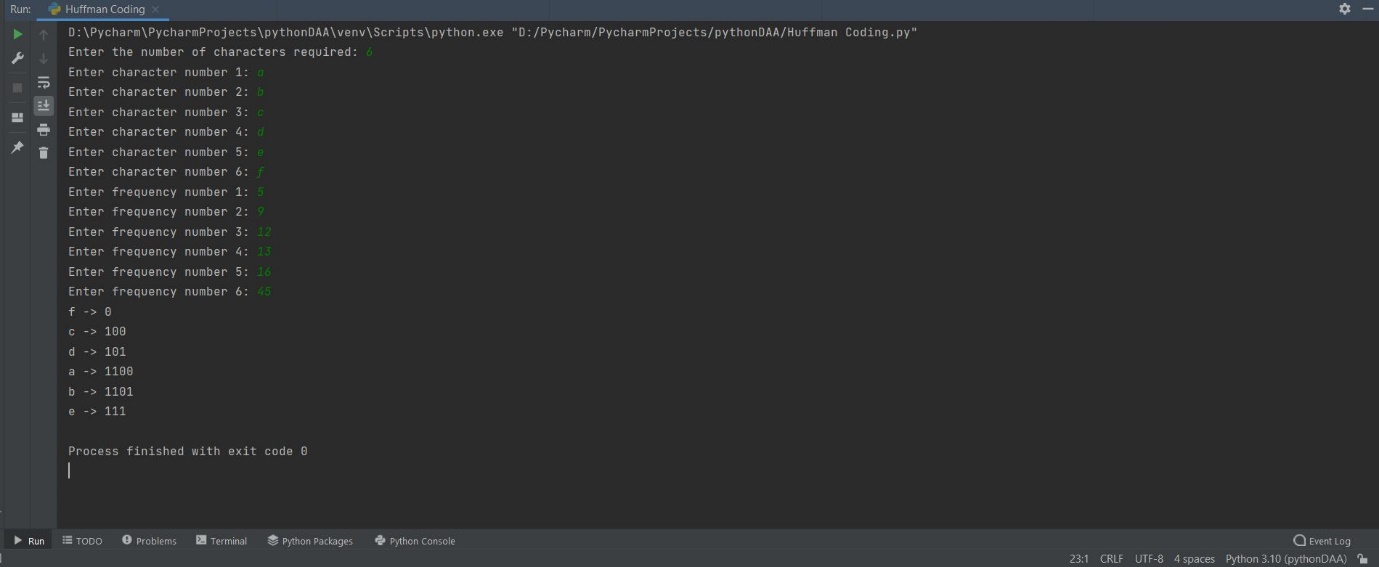


Q.2) Write a program to implement Huffman Encoding using a greedy strategy.

Program:

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 = []  
freq = []  
n = int(input("Enter the number of characters required: "))  
  
for i in range(n):  
 ch = input(f"Enter character number {i+1}: ")  
 chars.append(ch)  
  
for i in range(n):  
 num = int(input(f"Enter frequency number {i+1}: "))  
 freq.append(num)  
  
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])

Output:

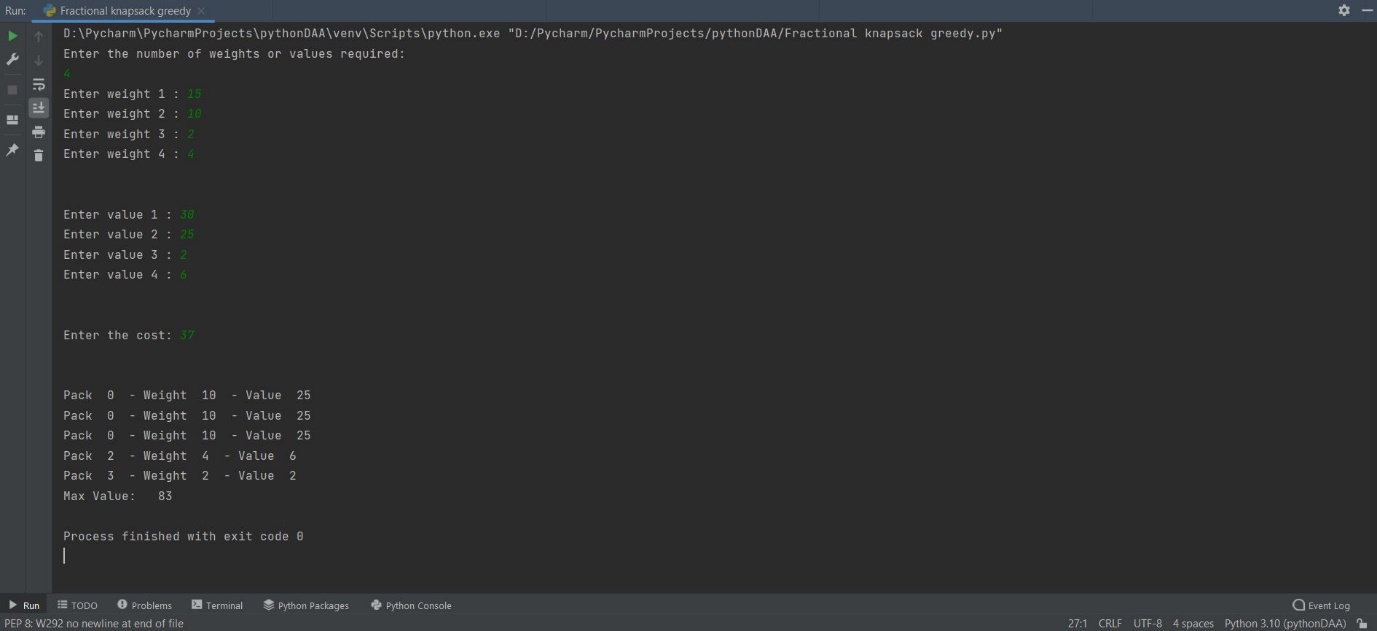


Q.3) Write a program to solve a fractional Knapsack problem using a greedy method.

Program:

class KnapsackPackage(object):  
 def \_\_init\_\_(self, weight, value):  
 self.weight = weight  
 self.value = value  
 self.cost = value / weight  
  
 def \_\_lt\_\_(self, other):  
 return self.cost < other.cost  
  
class FractionalKnapsack(object):  
 def knapsackGreProc(self, W, V, M, n):  
 packs = []  
  
 for i in range(n):  
 packs.append(KnapsackPackage(W[i], V[i]))  
 packs.sort(reverse=True)  
  
 remain = M  
 result = 0  
  
 i = 0  
 stopProc = False  
  
 while (stopProc != True):  
 if (packs[i].weight <= remain):  
 remain -= packs[i].weight;  
 result += packs[i].value;  
  
 print("Pack ", i, " - Weight ", packs[i].weight, " - Value ", packs[i].value)  
  
 if (packs[i].weight > remain):  
 i += 1  
  
 if (i == n):  
 stopProc = True  
  
 print("Max Value:\t", result)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 W = []  
 V = []  
 num = int(input("Enter the number of weights or values required: \n"))  
  
 for i in range(num):  
 n = int(input(f"Enter weight {i+1} : "))  
 W.append(n)  
 print("\n")  
  
 for i in range(num):  
 n = int(input(f"Enter value {i+1} : "))  
 V.append(n)  
 print("\n")  
  
 M = int(input("Enter the cost: "))  
 n = len(V)  
 print("\n")  
 proc = FractionalKnapsack()  
 proc.knapsackGreProc(W, V, M, n)

Output:

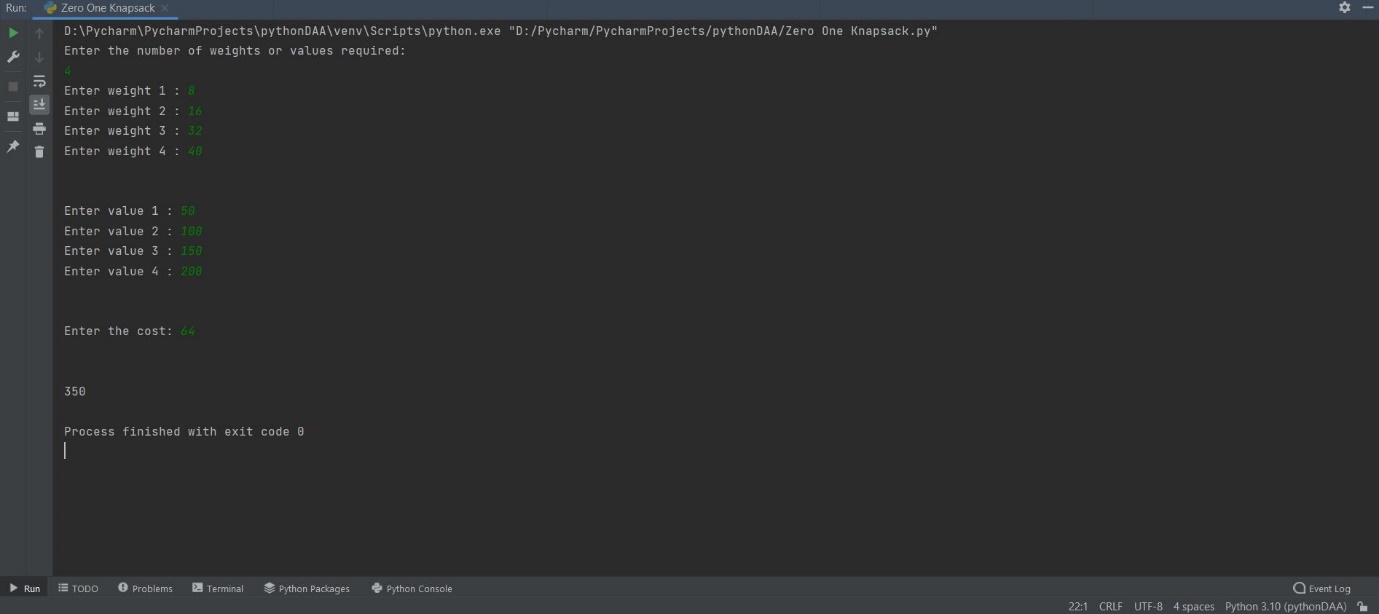


Q.4) Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

Program:

def knapSack(W, wt, val, n):  
 dp = [0 for i in range(W + 1)]  
  
 for i in range(1, n + 1):  
  
 for w in range(W, 0, -1):  
 if wt[i - 1] <= w:  
  
  
 dp[w] = max(dp[w], dp[w - wt[i - 1]] + val[i - 1])  
  
 return dp[W]  
  
  
W = []  
V = []  
num = int(input("Enter the number of weights or values required: \n"))  
  
for i in range(num):  
 n = int(input(f"Enter weight {i+1} : "))  
 W.append(n)  
print("\n")  
  
for i in range(num):  
 n = int(input(f"Enter value {i+1} : "))  
 V.append(n)  
print("\n")  
  
M = int(input("Enter the cost: "))  
n = len(V)  
print("\n")  
print(knapSack(M, W, V, n))

Output:



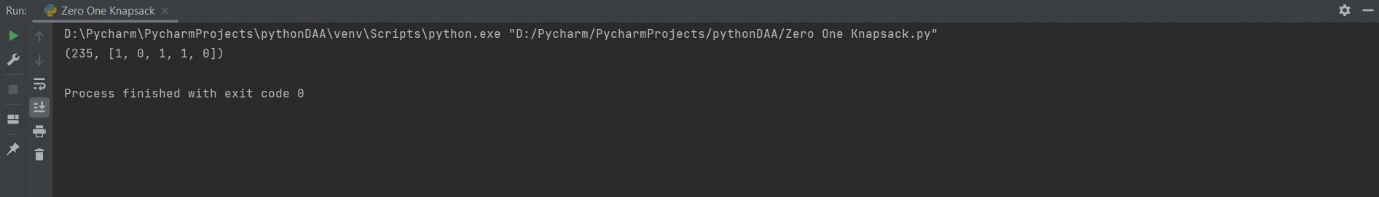
Branch And Bound Method:

from queue import Queue  
from collections import deque  
from collections import namedtuple  
  
  
class Node:  
 def \_\_init\_\_(self):  
 self.level = None  
 self.profit = None  
 self.bound = None  
 self.weight = None  
 self.contains = []  
  
 def \_\_str\_\_(self):  
 return "Level: %s Profit: %s Bound: %s Weight: %s" % (self.level, self.profit, self.bound, self.weight)  
  
  
def bound(node, n, W, items):  
 if(node.weight >= W):  
 return 0  
  
 profit\_bound = int(node.profit)  
 j = node.level + 1  
 totweight = int(node.weight)  
  
 while ((j < n) and (totweight + items[j].weight) <= W):  
 totweight += items[j].weight  
 profit\_bound += items[j].value  
 j += 1  
  
 if(j < n):  
 profit\_bound += (W - totweight) \* items[j].value / float(items[j].weight)  
  
 return profit\_bound  
  
Q = deque([])  
  
def KnapSackBranchNBound(weight, items, total\_items):  
 items = sorted(items, key=lambda x: x.value/float(x.weight), reverse=True)  
  
 u = Node()  
  
 u.level = -1  
 u.profit = 0  
 u.weight = 0  
  
 Q.append(u)  
 maxProfit = 0  
 bestItems = []  
  
 while (len(Q) != 0):  
  
 u = Q[0]  
 Q.popleft()  
 v = Node()  
  
 if u.level == -1:  
 v.level = 0  
  
 if u.level == total\_items - 1:  
 continue  
  
 v.level = u.level + 1  
 v.weight = u.weight + items[v.level].weight  
 v.profit = u.profit + items[v.level].value  
 v.contains = list(u.contains)  
 v.contains.append(items[v.level].index)  
  
 if (v.weight <= weight and v.profit > maxProfit):  
 maxProfit = v.profit  
 bestItems = v.contains  
  
 v.bound = bound(v, total\_items, weight, items)  
 if (v.bound > maxProfit):  
 # print v  
 Q.append(v)  
  
 v = Node()  
 v.level = u.level + 1  
 v.weight = u.weight  
 v.profit = u.profit  
 v.contains = list(u.contains)  
  
 v.bound = bound(v, total\_items, weight, items)  
 if (v.bound > maxProfit):  
 # print v  
 Q.append(v)  
  
 taken = [0] \* len(items)  
 for i in range(len(bestItems)):  
 taken[bestItems[i]] = 1  
  
 return maxProfit, taken  
  
def get\_solution(optimal\_value, taken):  
 output\_data = None  
 output\_data = str(optimal\_value) + ' ' + str(1) + '\n'  
 output\_data += ' '.join(map(str, taken))  
 return output\_data  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 Item = namedtuple("Item", ['index', 'value', 'weight'])  
 input\_data = open("D:\\test.data").read()  
 lines = input\_data.split('\n')  
  
 firstLine = lines[0].split()  
 item\_count = int(firstLine[0])  
 capacity = int(firstLine[1])  
  
 items = []  
  
 for i in range(1, item\_count+1):  
 line = lines[i]  
 parts = line.split()  
 items.append(Item(i-1, int(parts[0]), float(parts[1])))  
 kbb = KnapSackBranchNBound(capacity, items, item\_count)  
 print(kbb)

Output:

Test.data =

5 10  
40 2  
50 3.14  
100 1.98  
95 5  
30 3



5) Problem Statement:

Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen‘s matrix.

Program:

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()

Output:

D:\Pycharm\PycharmProjects\pythonDAA\venv\Scripts\python.exe D:/Pycharm/PycharmProjects/pythonDAA/N-queens.py

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

Process finished with exit code 0