Exp 1 A  
  
# Program to display the Fibonacci sequence up to n-th term using recursive method

def fibonacci(n):

if(n <= 1):

return n

else:

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

n = int(input("Enter number of terms:"))

print("Fibonacci sequence:")

for i in range(n):

print(fibonacci(i))

//OUTPUT

Enter number of terms:5

Fibonacci sequence:

0

1

1

2

3

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

nterms = int(input("Enter number of 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

//OUTPUT

Enter number of terms 4

Fibonacci sequence:

0

1

1

2

Exp 2 A  
  
//Code for Huffman Encoding in Python

# 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, rig

ht)

heapq.heappush(nodes, newNode)

# Huffman Tree is ready!

printNodes(nodes[0])

//OUTPUT

f -> 0

c -> 100

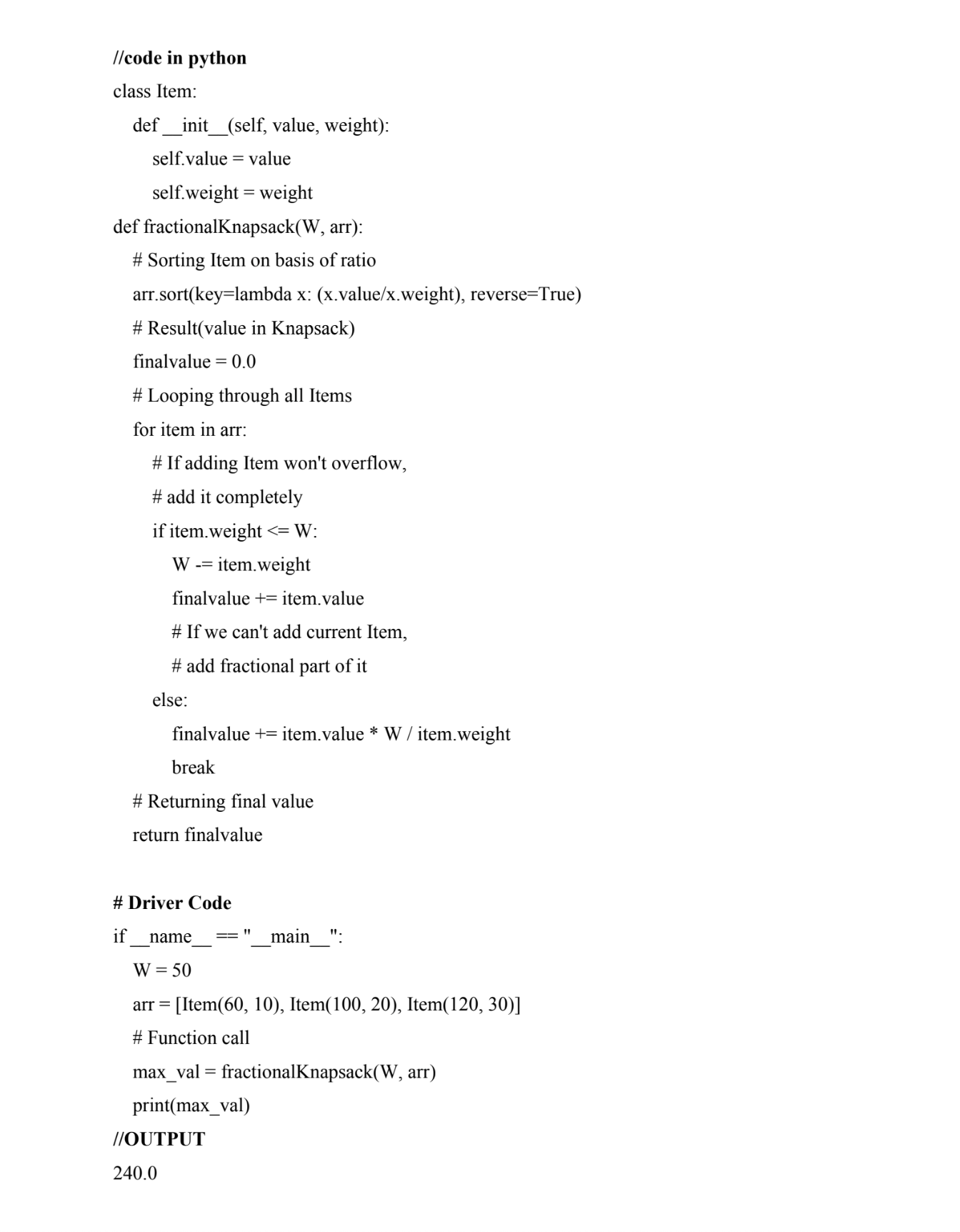
d -> 101

a -> 1100

b -> 1101

e -> 111

exp 3 A



Exp 4 A  
# code

# A Dynamic Programming based Python

# Program for 0-1 Knapsack problem

# Returns the maximum value that can

# be put in a knapsack of capacity W

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

dp = [0 for i in range(W+1)] # Making the dp array

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

# taking first i elements

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

# starting from back,so that we also have data of

# previous computation when taking i-1 items

if wt[i-1] <= w:

# finding the maximum value

dp[w] = max(dp[w], dp[w-wt[i-1]]+val[i-1])

return dp[W] # returning the maximum value of knapsack

# Driver code

val = [60, 100, 120]

wt = [10, 20, 30]

W = 50

n = len(val)

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

Output

220

Exp 5 A  
//Code:

# Python3 program to solve N Queen

# Problem using backtracking

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

print(board[i][j], end = " ")

print()

# A utility function to check if a queen can

# be placed on board[row][col]. Note that this

# function is called when "col" queens are

# already placed in columns from 0 to col -1.

# So we need to check only left side for

# attacking queens

def isSafe(board, row, col):

# Check this row on left side

for i in range(col):

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

return False  
# Check upper diagonal on left side

for i, j in zip(range(row, -1, -1),

range(col, -1, -1)):

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

return False

# Check lower diagonal on left side

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):

# base case: If all queens are placed

# then return true

if col >= N:

return True

# Consider this column and try placing

# this queen in all rows one by one

for i in range(N):

if isSafe(board, i, col):

# Place this queen in board[i][col]

board[i][col] = 1

# recur to place rest of the queens

if solveNQUtil(board, col + 1) == True:

return True  
# If placing queen in board[i][col

# doesn't lead to a solution, then

# queen from board[i][col]

board[i][col] = 0

# if the queen can not be placed in any row in

# this column col then return false

return False

# This function solves the N Queen problem using

# Backtracking. It mainly uses solveNQUtil() to

# solve the problem. It returns false if queens

# cannot be placed, otherwise return true and

# placement of queens in the form of 1s.

# note that there may be more than one

# solutions, this function prints one of the

# feasible solutions.

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

# Driver Code

solveNQ()