Daa 2 – huffmancode

```
# 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])
```

daa 3 fractional knapsack

```
# Structure for an item which stores weight and
# corresponding value of Item
class Item:
	def __init__(self, value, weight):
		self.value = value
		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.value/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.value

        # If we can't add current Item,
        # add fractional part of it
        else:
            finalvalue += item.value * 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)
```

daa 3 0-1 knapsack

```
def knapSack(W, wt, val, n):
	K = [[0 for x in range(W + 1)] for x in range(n + 1)]

	# Build table K[][] in bottom up manner
	for i in range(n + 1):
		for w in range(W + 1):
			if i == 0 or w == 0:
				K[i][w] = 0
			elif wt[i-1] <= w:
				K[i][w] = max(val[i-1]
						+ K[i-1][w-wt[i-1]],
							K[i-1][w])
			else:
				K[i][w] = K[i-1][w]

	return K[n][W]

def InputList():
  lst=[]
  n=int (input("enter number of elements: "))
```

```
  for i in range(0,n):
    ele = int (input())
    lst.append(ele)

  return lst



# Driver code
#val = [60, 100, 120]
val = InputList()
wt = InputList()
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```

daa – 5

```
""" Python3 program to solve N Queen Problem using
backtracking """
N = 4

""" ld is an array where its indices indicate row-col+N-1
(N-1) is for shifting the difference to store negative
indices """
ld = [0] * 30

""" rd is an array where its indices indicate row+col
and used to check whether a queen can be placed on
right diagonal or not"""
rd = [0] * 30
```

```
"""column array where its indices indicates column and
used to check whether a queen can be placed in that
	row or not"""
cl = [0] * 30

""" A utility function to print solution """
def printSolution(board):
	for i in range(N):
		for j in range(N):
			print(board[i][j], end = " ")
		print()

""" A recursive utility function to solve N
Queen problem """
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):

		""" Check if the queen can be placed on board[i][col] """
		""" A check if a queen can be placed on board[row][col].
		We just need to check ld[row-col+n-1] and rd[row+coln]
		where ld and rd are for left and right diagonal respectively"""
		if ((ld[i - col + N - 1] != 1 and
```

```
				rd[i + col] != 1) and cl[i] != 1):

				""" Place this queen in board[i][col] """
				board[i][col] = 1
				ld[i - col + N - 1] = rd[i + col] = cl[i] = 1

				""" recur to place rest of the queens """
				if (solveNQUtil(board, col + 1)):
					return True

				""" If placing queen in board[i][col]
				doesn't lead to a solution,
				then remove queen from board[i][col] """
				board[i][col] = 0 # BACKTRACK
				ld[i - col + N - 1] = rd[i + col] = cl[i] = 0

				""" If the queen cannot 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
prints placement of queens in the form of 1s.
Please 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):
                printf("Solution does not exist")
                return False
        printSolution(board)
        return True

# Driver Code
solveNQ()

# This code is contributed by SHUBHAMSINGH10
```