Group A- Assignment No. 1

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

Code:-

```
def fibo_recursive(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    return fibo_recursive(n - 1) + fibo_recursive(n - 2)


def fibo_iterative(n):
    a, b = 0, 1
    if n == 1:
        return a
    elif n == 2:
        return b
    while n - 2 > 0:
        c = a + b
        a, b = b, c
        n = n - 1
    return c


num = int(input("Enter the number of fibonacci numbers required : "))
iterative, recursive = [], []
for i in range(1, num+1):
    iterative.append(str(fibo_iterative(i)))
```

```
    recursive.append(str(fibo_recursive(i)))
print('Iterative: ' + ' '.join(iterative))
print('Recursive: ' + ' '.join(recursive))
```

OUTPUT :

```
Enter the number of fibonacci numbers required : 5
Iterative: 0 1 1 2 3
Recursive: 0 1 1 2 3

Process finished with exit code 0
```

Group A -Assignment No. 2

Aim: Write a program to implement Huffman Encoding using a greedy strategy

Code :-

```
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 = [1, 0, 3, 4, 5, 6]
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)

print("Huffman Tree : ")
printNodes(nodes[0])
```

OUTPUT:

```
Huffman Tree :
d -> 00
b -> 0100
a -> 0101
c -> 011
e -> 10
f -> 11

Process finished with exit code 0
```

Group A- Assignment No. 3

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

Code:

```
class Item:
  def __init__(self, value, weight):
    self.value = value
    self.weight = weight


def fractionalKnapsack(W, arr):

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

  finalvalue = 0.0

  for item in arr:

    if item.weight <= W:
      W -= item.weight
      finalvalue += item.value

    else:
      finalvalue += item.value * W / item.weight
      break
  return finalvalue


if __name__ == "__main__":

  W = 50
  arr = [Item(70, 10), Item(100, 20), Item(120, 30)]

  max_val = fractionalKnapsack(W, arr)
  print(max_val)
```

OUTPUT:-

250.0

Process finished with exit code 0

Group A -Assignment No. 4

Aim: Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

CODE :-

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


val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print("Maximum total value : ", knapSack(W, wt, val, n))
```

OUTPUT :-

```
Maximum total value :  220


Process finished with exit code 0
```

Group A -Assignment No. 5

Aim: Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen_s matrix.

CODE :-

```
global N
N = 4
cols = set([i for i in range(N)])


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
    i, j = row - 1, col + 1
    while i >= 0 and j < N:
        if board[i][j] == 1:
            return False
        i -= 1
        j += 1
    i, j = row + 1, col + 1
    while i < N and j < N:
        if board[i][j] == 1:
            return False
        i += 1
        j += 1
    i, j = row - 1, col - 1
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
            return False
```

```
        i -= 1
        j -= 1
    i, j = row + 1, col - 1
    while i < N and j >= 0:
        if board[i][j] == 1:
            return False
        i += 1
        j -= 1
    return True


def solveNQUtil(board):
    if not cols:
        return True
    col = list(cols)[0]
    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            cols.remove(col)
            if solveNQUtil(board) == True:
                return True
            cols.add(col)
            board[i][col] = 0
    return False

def solveNQ():
    board = []
    for i in range(N):
        temp = []
        for j in range(N):
            temp.append(0)
        board.append(temp)
    i, j = input("Enter i, j position of first queen : ").split()
    i, j = int(i), int(j)
    board[i][j] = 1
    cols.remove(j)
    if solveNQUtil(board) == False:
        print("Solution does not exist")
        return False

    printSolution(board)
    return True

solveNQ()
```

OUTPUT :-

Enter i, j position of first queen : 2 3

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

Process finished with exit code 0