Practical 1. Write a program to calculate Fibonacci numbers and its step count

```
def fibonacci(n):
    if(n<=1):
        return n
    else:
        return (fibonacci(n-1)+fibonacci(n-2))

n=int(input("Enter number of term print:"))
print("fibonacci sequence")
for i in range(n):
    print(fibonacci(i))
```

Practical 2: Implement job sequencing with deadlines using a greedy method.

```
def printjobscheduling(arr, t):
    n = len(arr)

    for i in range(n):
        for j in range(n - 1 - i):
            if arr[j][2] < arr[j + 1][2]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    result = [False] * t
    job = ['-1'] * t

    for i in range(len(arr)):
        for j in range(min(t - 1, arr[i][1] - 1), -1, -1):
            if not result[j]:
                result[j] = True
                job[j] = arr[i][0]
```

```
            break

    print(job)

if __name__ == '__main__':
    arr = [['a', 2, 100],
           ['b', 1, 19],
           ['c', 2, 27],
           ['d', 1, 25],
           ['e', 3, 15]]
    print("Following is the maximum profit sequence of jobs:")
    printjobscheduling(arr, 3)
```

Practical 3: To solve a fractional knapsack problem using a greedy method.

```
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(60, 10), Item(100, 20), Item(120, 30)]
    max_val = fractionalKnapsack(W, arr)
    print ('Maximum value we can obtain = {}'.format(max_val))
```

Practical 4: Write a program to solve a 0-1 knapsack problem using dynamic programming or branch and bound strategy

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

Practical 5: Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queen to generate the final queen's matrix.

```
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()
```