Fibonacci Series using Recursion:

```
first_num=int(input("Enter the first number of fibonacci Series"))
second_num=int(input("Enter the second number of fibonacci Series"))
num_of_terms=int(input("Enter the number of terms"))

print(first_num,second_num)
print("The numbers in fibonacci series are")

while(num_of_terms-2):
    third_num=first_num+second_num
    first_num=second_num
    second_num=third_num
    print(third_num)
    num_of_terms=num_of_terms-1
```

Time Complexity: Exponential

Fibonacci Series without using Recursion:

```
def recur_fibo(n):
    if n<=1:
        return n
    else:
        return(recur_fibo(n-1)+recur_fibo(n-2))
nterms=int(input("Enter the terms"))

if nterms<=0:
        print("Enter a positive integer")

else:
        print("Fibonacci Sequence")

        for i in range(nterms):
            print(recur_fibo(i))
```

Time Complexity:O(n)

```
# A Huffman Tree Node
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=''):

	# huffman code for current node
	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 = [ 5, 9, 12, 13, 16, 45]

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)

	# 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)


printNodes(nodes[0])
```

Problem Statement: Write a program 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))
```

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

```
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]



val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)

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

Problem Statement: Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-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):
                        # Place this queen in 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()
```