```
#Assignment 1
#Recursive

def recur_fibo(n):
   if n<=1:
      return n
   else:
      return(recur_fibo(n-1) + recur_fibo(n-2))
nterms = 10
# check if the number of terms is valid
if nterms <= 0:
  print("Plese enter a positive integer")
else:
  print("Fibonacci sequence:")
  for i in range(nterms):
      print(recur_fibo(i))

#non-recursive

def fib(n):
    if n == 1:
      return [1]
    if n == 2:
        return [1, 1]
    fibs = [1, 1]
    for _ in range(2, n):
        fibs.append(fibs[-1] + fibs[-2])
        return fibs
print(fib(3))


#Assignment 2
# Huffman Coding in python

string = 'BCAADDDCCACACAC'
# Creating tree nodes
class NodeTree(object):
 def __init__(self, left=None, right=None):
  self.left = left
  self.right = right
 def children(self):
  return (self.left, self.right)
 def nodes(self):
  return (self.left, self.right)
 def __str__(self):
  return '%s_%s' % (self.left, self.right)
# Main function implementing huffman coding
def huffman_code_tree(node, left=True, binString=''):
  if type(node) is str:
     return {node: binString}
  (l, r) = node.children()
  d = dict()
  d.update(huffman_code_tree(l, True, binString + '0'))
  d.update(huffman_code_tree(r, False, binString + '1'))
  return d
# Calculating frequency
freq = {}
for c in string:
  if c in freq:
    freq[c] += 1
  else:
    freq[c] = 1
freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)
nodes = freq
while len(nodes) > 1:
  (key1, c1) = nodes[-1]
  (key2, c2) = nodes[-2]
  nodes = nodes[:-2]
  node = NodeTree(key1, key2)
  nodes.append((node, c1 + c2))
  nodes = sorted(nodes, key=lambda x: x[1], reverse=True)
huffmanCode = huffman_code_tree(nodes[0][0])
print(' Char | Huffman code ')
print('----------------------')
for (char, frequency) in freq:
  print(' %-4r |%12s' % (char, huffmanCode[char]))
```

```
#Assignment 3
# 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)



#Assignment 4

# A naive recursive implementation
# of 0-1 Knapsack Problem
# Returns the maximum value that
# can be put in a knapsack of
# capacity W
def knapSack(W, wt, val, n):
# Base Case
  if n == 0 or W == 0:
     return 0
# If weight of the nth item is
# more than Knapsack of capacity W,
# then this item cannot be included
# in the optimal solution
  if (wt[n-1] > W):
     return knapSack(W, wt, val, n-1)
# return the maximum of two cases:
# (1) nth item included
# (2) not included
  else:
     return max(
        val[n-1] + knapSack(
           W-wt[n-1], wt, val, n-1),
        knapSack(W, wt, val, n-1))
# end of function knapSack
#Driver Code
val= [60, 100, 120]
wt = [10, 20, 30]
W=50
n=len(val)
print(knapSack(W, wt, val, n))
```

```
//SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract Student_management{
   struct Student{
      int stud_id;
      string name;
      string department;
   }
   Student[] Students;

   function add_stud(int stud_id,string memory name, string memory
department)public{
      Student memory stud = Student(stud_id,name,department);
      Students.push(stud);
  }

  function getStudent(int stud_id)public view returns(string memory, string memory){
     for (uint i=0;i<Students.length;i++){
        Student memory stud = Students[i];
        if(stud.stud_id == stud_id){
           return (stud.name,stud.department);
        }
     }
     return ("Not found","Not Found");
  }
}
```

```
//SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract banking{

  mapping(address=>uint) public user_account;
  mapping(address=>bool) public user_exists;

  function ceate_account()public payable returns(string memory){
     require (user_exists[msg.sender]==false,"Account already created");
     if(msg.value==0){
        user_account[msg.sender]=0;
        user_exists[msg.sender]=true;
        return "Account Created";
     }
     require(user_exists[msg.sender]==false,"Account already created");
        user_account[msg.sender]=msg.value;
        user_exists[msg.sender]=true;
        return "Account Created";
  }
  function deposit() public payable returns(string memory){
     require(user_exists[msg.sender]==true,"Account not Created");
     require(msg.value>0,"Value for deposit is zero");
     user_account[msg.sender]=user_account[msg.sender]+msg.value;
     return "Deposited Sucessfully";
  }

  function withdrao(uint amount) public payable returns(string memory){
     require(user_account[msg.sender]>amount,"Insufficient Balance");
     require(user_exists[msg.sender]==true,"Account not created");
     require(amount>0,"Amount should be not zero");
      user_account[msg.sender]=user_account[msg.sender]-amount;
     payable(msg.sender).transfer(amount);
     return "Withdrawl Sucess";
  }

  function transfer(address payable userAddress,uint amount) public returns(string
memory){
     require(user_account[msg.sender]>amount,"Insufficient balence in bank");
     require(user_exists[msg.sender]==true,"Account not created");
     require(user_exists[userAddress]==true,"Transfer account not exists");
     require(amount>0,"Account should not be 0");
     user_account[msg.sender]=user_account[msg.sender]-amount;
     user_account[userAddress]=user_account[userAddress]+amount;
     return "Transfer Sucess";
  }

  function send_amt(address payable toAddress, uint256 amount) public payable
returns(string memory){
     require(user_account[msg.sender]>amount,"Insufficient balance in bank");
     require(user_exists[msg.sender]==true,"Account is not created");
     require(amount>0,"Account shoule not be 0");
     user_account[msg.sender]=user_account[msg.sender]-amount;
```

```
#Assignment 5
#Design 8 queen 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):
# 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):
board[i][col] = 0
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()
```

```
   toAddress.transfer(amount);
      return "Tramsfer sucess";
   }

   function user_bal()public view returns(uint){
return user_account[msg.sender];
   }

   function account_exists()public view returns(bool){
      return user_exists[msg.sender];
   }

}
```