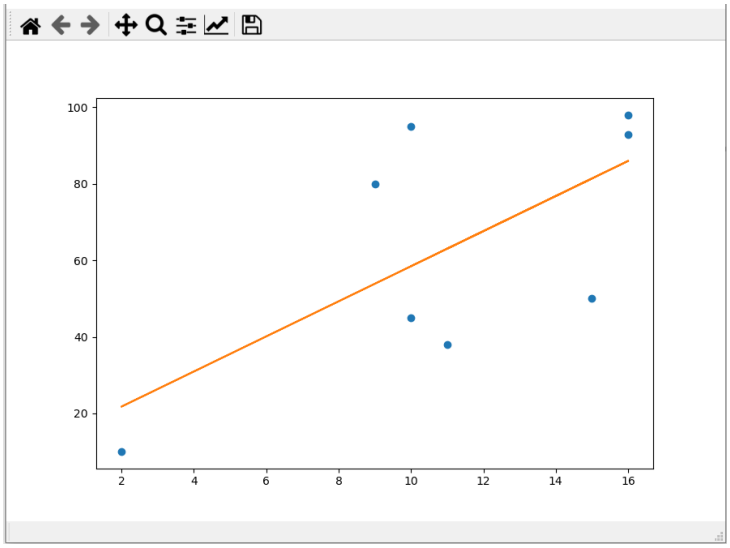
//ASSIGNMENT 01
//CODE

```
import matplotlib.pyplot as plt
import pandas as pd
# Read Dataset
dataset=pd.read_csv("hours.csv")
X=dataset.iloc[:,:-1].values
y=dataset.iloc[:,1].values

# Import the Linear Regression and Create object of it
from sklearn.linear_model import LinearRegression
regressor=LinearRegression()
regressor.fit(X,y)
Accuracy=regressor.score(X, y)*100
print("Accuracy :")
print(Accuracy)

# Predict the value using Regressor Object
y_pred=regressor.predict([[10]])
print(y_pred)

# Take user input
hours=int(input('Enter the no of hours'))

#calculate the value of y
eq=regressor.coef_*hours+regressor.intercept_
y_new='%f*%f+%f' %(regressor.coef_,hours,regressor.intercept_)
print("y :")
print(y_new)
print("Risk Score : ", eq[0])
plt.plot(X,y,'o')
plt.plot(X,regressor.predict(X));
plt.show()
```

//OUTPUT

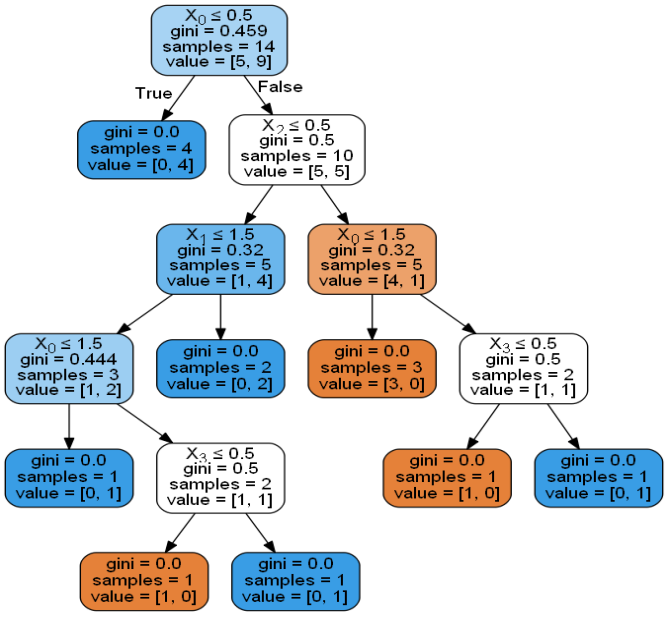
```
runfile('C:/Users/HP/Desktop/LP/III/LP-3/A1/A1.py', wdir='C:/Users/HP/Desktop/LP/III/LP-3/A1')
Accuracy : 43.709481451010035
Enter the no of hours11
y: 4.587899*11.000000+12.584628
Risk Score:  63.05151267375307
```



//ASSIGNMENT 02
//CODE

```
#import packages
import pandas as pd
import numpy as np
# Read dataset
dataset=pd.read_csv("tree1.csv")
x=dataset.iloc[:,:-1]
y=dataset.iloc[:,5]
#Label encoder
from sklearn.preprocessing import LabelEncoder
le=LabelEncoder()
x=x.apply(le.fit_transform)
print(x)
# 1 1 0 0
#import Decesion Tree Classifier
from sklearn.tree import DecisionTreeClassifier
# Create decision tree classifer object
regressor=DecisionTreeClassifier()
# Train model
regressor.fit(x.iloc[:,1:5],y)
x_in=np.array([1,1,0,0])
y_pred=regressor.predict([x_in])
print(y_pred)
from sklearn.externals.six import StringIO
#from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus
dot_data=StringIO()
export_graphviz(regressor,out_file=dot_data,filled=True,rounded=True,special_characters=True)
#Draw Graph
graph=pydotplus.graph_from_dot_data(dot_data.getvalue())
# Show graph & Create png File
graph.write_png("tree.png")
```

//OUTPUT



//ASSIGNMENT 03
//CODE

```
#import the packages
import pandas as pd
import numpy as np
#Read dataset
dataset=pd.read_csv("kdata.csv")
X=dataset.iloc[:,:-1].values
y=dataset.iloc[:,2].values
#import KNeighborshood Classifier and create object of it
from sklearn.neighbors import KNeighborsClassifier
#Creating model
classifier=KNeighborsClassifier(n_neighbors=3)
# Training model
classifier.fit(X,y)
#predict the class for the point(6,6)
X_test=np.array([6,6])
# Predictions for test data
y_pred=classifier.predict([X_test])
print(y_pred)
# KNeighborsClassifier looks for the 5 nearest neighbors
#If set to uniform, all points in each neighbourhood have
classifier=KNeighborsClassifier(n_neighbors=3,weights='distance')
classifier.fit(X,y)
#predict the class for the point(6,6)
X_test=np.array([6,2])
y_pred=classifier.predict([X_test])
print(y_pred)
```

//OUTPUT

```
['negative']
['positive']
```

```
//ASSIGNMENT 04
//CODE
#import packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
#create dataset using DataFrame
df=pd.DataFrame({'X':[0.1,0.15,0.08,0.16,0.2,0.25,0.24,0.3],
                 'y':[0.6,0.71,0.9,0.85,0.3,0.5,0.1,0.2]})
f1 = df['X'].values
f2 = df['y'].values
X = np.array(list(zip(f1, f2)))
print(X)
#centroid points
C_x=np.array([0.1,0.3])
C_y=np.array([0.6,0.2])
centroids=C_x,C_y
#plot the given points
colmap = {1: 'r', 2: 'b'}
plt.scatter(f1, f2, color='k')
plt.show()
#for i in centroids():
plt.scatter(C_x[0],C_y[0], color=colmap[1])
plt.scatter(C_x[1],C_y[1], color=colmap[2])
plt.show()
C = np.array(list((C_x, C_y)), dtype=np.float32)
print (C)
#plot given elements with centroid elements
plt.scatter(f1, f2, c='#050505')
plt.scatter(C_x[0], C_y[0], marker='*', s=200, c='r')
plt.scatter(C_x[1], C_y[1], marker='*', s=200, c='b')
plt.show()
#import KMeans class and create object of it
from sklearn.cluster import KMeans
model=KMeans(n_clusters=2,random_state=0)
model.fit(X)
labels=model.labels_
print(labels)
#using labels find population around centroid
count=0
for i in range(len(labels)):
    if (labels[i]==1):
        count=count+1
print('No of population around cluster 2:',count-1)
#Find new centroids
new_centroids = model.cluster_centers_
print('Previous value of m1 and m2 is:')
print('M1==',centroids[0])
print('M1==',centroids[1])
print('updated value of m1 and m2 is:')
print('M1==',new_centroids[0])
print('M1==',new_centroids[1])

//OUTPUT
[[0.1  0.6 ]
 [0.15 0.71]
```

[0.08 0.9]
[0.16 0.85]
[0.2 0.3]
[0.25 0.5]
[0.24 0.1]
[0.3 0.2]]
[[0.1 0.3]
[0.6 0.2]]
[1 1 1 1 0 0 0 0]
No of population around cluster 2: 3
Previous value of m1 and m2 is:
M1== [0.1 0.3]
M1== [0.6 0.2]
updated value of m1 and m2 is:
M1== [0.2475 0.275]
M1== [0.1225 0.765]

```
//ASSIGNMENT 05
//CODE
from sys import exit
from time import time
KeyLength = 10
SubKeyLength = 8
DataLength = 8
FLength = 4
# Tables for initial and final permutations (b1, b2, b3, ... b8)
IPtable = (2, 6, 3, 1, 4, 8, 5, 7)
FPtable = (4, 1, 3, 5, 7, 2, 8, 6)
# Tables for subkey generation (k1, k2, k3, ... k10)
P10table = (3, 5, 2, 7, 4, 10, 1, 9, 8, 6)
P8table = (6, 3, 7, 4, 8, 5, 10, 9)
# Tables for the fk function
EPtable = (4, 1, 2, 3, 2, 3, 4, 1)
S0table = (1, 0, 3, 2, 3, 2, 1, 0, 0, 2, 1, 3, 3, 1, 3, 2)
S1table = (0, 1, 2, 3, 2, 0, 1, 3, 3, 0, 1, 0, 2, 1, 0, 3)
P4table = (2, 4, 3, 1)
def perm(inputByte, permTable):
    """Permute input byte according to permutation table"""
    outputByte = 0
    for index, elem in enumerate(permTable):
        if index >= elem:
            outputByte |= (inputByte & (128 >> (elem - 1))) >> (index - (elem - 1))
        else:
            outputByte |= (inputByte & (128 >> (elem - 1))) << ((elem - 1) - index)
    return outputByte
def ip(inputByte):
    """Perform the initial permutation on data"""
    return perm(inputByte, IPtable)
def fp(inputByte):
    """Perform the final permutation on data"""
    return perm(inputByte, FPtable)
def swapNibbles(inputByte):
    """Swap the two nibbles of data"""
    return (inputByte << 4 | inputByte >> 4) & 0xff
def keyGen(key):
    """Generate the two required subkeys"""
    def leftShift(keyBitList):
        """Perform a circular left shift on the first and second five bits"""
        shiftedKey = [None] * KeyLength
        shiftedKey[0:9] = keyBitList[1:10]
        shiftedKey[4] = keyBitList[0]
        shiftedKey[9] = keyBitList[5]
        return shiftedKey
    # Converts input key (integer) into a list of binary digits
    keyList = [(key & 1 << i) >> i for i in reversed(range(KeyLength))]
    permKeyList = [None] * KeyLength
    for index, elem in enumerate(P10table):
        permKeyList[index] = keyList[elem - 1]
    shiftedOnceKey = leftShift(permKeyList)
    shiftedTwiceKey = leftShift(leftShift(shiftedOnceKey))
    subKey1 = subKey2 = 0
    for index, elem in enumerate(P8table):
        subKey1 += (128 >> index) * shiftedOnceKey[elem - 1]
```

```
        subKey2 += (128 >> index) * shiftedTwiceKey[elem - 1]
    return (subKey1, subKey2)
def fk(subKey, inputData):
    """Apply Feistel function on data with given subkey"""
    def F(sKey, rightNibble):
        aux = sKey ^ perm(swapNibbles(rightNibble), EPtable)
        index1 = ((aux & 0x80) >> 4) + ((aux & 0x40) >> 5) + \
                ((aux & 0x20) >> 5) + ((aux & 0x10) >> 2)
        index2 = ((aux & 0x08) >> 0) + ((aux & 0x04) >> 1) + \
                ((aux & 0x02) >> 1) + ((aux & 0x01) << 2)
        sboxOutputs = swapNibbles((S0table[index1] << 2) + S1table[index2])
        return perm(sboxOutputs, P4table)
    leftNibble, rightNibble = inputData & 0xf0, inputData & 0x0f
    return (leftNibble ^ F(subKey, rightNibble)) | rightNibble
def encrypt(key, plaintext):
    """Encrypt plaintext with given key"""
    data = fk(keyGen(key)[0], ip(plaintext))
    return fp(fk(keyGen(key)[1], swapNibbles(data)))
def decrypt(key, ciphertext):
    """Decrypt ciphertext with given key"""
    data = fk(keyGen(key)[1], ip(ciphertext))
    return fp(fk(keyGen(key)[0], swapNibbles(data)))
if __name__ == '__main__':
    try:
        assert encrypt(0b0000000000, 0b10101010) == 0b00010001
    except AssertionError:
        print("Error on encrypt:")
        print("Output: ", encrypt(0b0000000000, 0b10101010), "Expected: ", 0b00010001)
        exit(1)
    try:
        assert encrypt(0b1110001110, 0b10101010) == 0b11001010
    except AssertionError:
        print("Error on encrypt:")
        print("Output: ", encrypt(0b1110001110, 0b10101010), "Expected: ", 0b11001010)
        exit(1)
    try:
        assert encrypt(0b1110001110, 0b01010101) == 0b01110000
    except AssertionError:
        print("Error on encrypt:")
        print("Output: ", encrypt(0b1110001110, 0b01010101), "Expected: ", 0b01110000)
        exit(1)
    try:
        assert encrypt(0b1111111111, 0b10101010) == 0b00000100
    except AssertionError:
        print("Error on encrypt:")
        print("Output: ", encrypt(0b1111111111, 0b10101010), "Expected: ", 0b00000100)
        exit(1)

    t1 = time()
    for i in range(1000):
        encrypt(0b1110001110, 0b10101010)
    t2 = time()
    print("Elapsed time for 1,000 encryptions: {:0.3f}s".format(t2 - t1))
```

//OUTPUT
runfile('C:/Users/HP/Desktop/LP/III/LP-3/A2/A2.py', wdir='C:/Users/HP/Desktop/LP/III/LP-3/A2')
Elapsed time for 1,000 encryptions: 0.088s.

```
//ASSIGNMENT 06
//CODE
# Description: Simplified AES implementation in Python 3
import sys
# S-Box
sBox  = [0x9, 0x4, 0xa, 0xb, 0xd, 0x1, 0x8, 0x5,
        0x6, 0x2, 0x0, 0x3, 0xc, 0xe, 0xf, 0x7]
# Inverse S-Box
sBoxI = [0xa, 0x5, 0x9, 0xb, 0x1, 0x7, 0x8, 0xf,
        0x6, 0x0, 0x2, 0x3, 0xc, 0x4, 0xd, 0xe]
# Round keys: K0 = w0 + w1; K1 = w2 + w3; K2 = w4 + w5
w = [None] * 6
def mult(p1, p2):
    """Multiply two polynomials in GF(2^4)/x^4 + x + 1"""
    p = 0
    while p2:
        if p2 & 0b1:
            p ^= p1
        p1 <<= 1
        if p1 & 0b10000:
            p1 ^= 0b11
        p2 >>= 1
    return p & 0b1111
def intToVec(n):
    """Convert a 2-byte integer into a 4-element vector"""
    return [n >> 12, (n >> 4) & 0xf, (n >> 8) & 0xf,  n & 0xf]
def vecToInt(m):
    """Convert a 4-element vector into 2-byte integer"""
    return (m[0] << 12) + (m[2] << 8) + (m[1] << 4) + m[3]
def addKey(s1, s2):
    """Add two keys in GF(2^4)"""
    return [i ^ j for i, j in zip(s1, s2)]
def sub4NibList(sbox, s):
    """Nibble substitution function"""
    return [sbox[e] for e in s]
def shiftRow(s):
    """ShiftRow function"""
    return [s[0], s[1], s[3], s[2]]
def keyExp(key):
    """Generate the three round keys"""
    def sub2Nib(b):
        """Swap each nibble and substitute it using sBox"""
        return sBox[b >> 4] + (sBox[b & 0x0f] << 4)
    Rcon1, Rcon2 = 0b10000000, 0b00110000
    w[0] = (key & 0xff00) >> 8
    w[1] = key & 0x00ff
    w[2] = w[0] ^ Rcon1 ^ sub2Nib(w[1])
    w[3] = w[2] ^ w[1]
    w[4] = w[2] ^ Rcon2 ^ sub2Nib(w[3])
    w[5] = w[4] ^ w[3]
def encrypt(ptext):
    """Encrypt plaintext block"""
    def mixCol(s):
        return [s[0] ^ mult(4, s[2]), s[1] ^ mult(4, s[3]),
                s[2] ^ mult(4, s[0]), s[3] ^ mult(4, s[1])]
```

```
    state = intToVec(((w[0] << 8) + w[1]) ^ ptext)
    state = mixCol(shiftRow(sub4NibList(sBox, state)))
    state = addKey(intToVec((w[2] << 8) + w[3]), state)
    state = shiftRow(sub4NibList(sBox, state))
    return vecToInt(addKey(intToVec((w[4] << 8) + w[5]), state))

def decrypt(ctext):
    """Decrypt ciphertext block"""
    def iMixCol(s):
        return [mult(9, s[0]) ^ mult(2, s[2]), mult(9, s[1]) ^ mult(2, s[3]),
                mult(9, s[2]) ^ mult(2, s[0]), mult(9, s[3]) ^ mult(2, s[1])]
    state = intToVec(((w[4] << 8) + w[5]) ^ ctext)
    state = sub4NibList(sBoxI, shiftRow(state))
    state = iMixCol(addKey(intToVec((w[2] << 8) + w[3]), state))
    state = sub4NibList(sBoxI, shiftRow(state))
    return vecToInt(addKey(intToVec((w[0] << 8) + w[1]), state))

if __name__ == '__main__':
    plaintext = 0b1101011100101000
    key = 0b0100101011110101
    ciphertext = 0b0010010011101100
    keyExp(key)
    try:
        assert encrypt(plaintext) == ciphertext
    except AssertionError:
        print("Encryption error")
        print(encrypt(plaintext), ciphertext)
        sys.exit(1)
    try:
        assert decrypt(ciphertext) == plaintext
    except AssertionError:
        print("Decryption error")
        print(decrypt(ciphertext), plaintext)
        sys.exit(1)
    print("Test ok!")
    sys.exit()

//OUTPUT
runfile('C:/Users/HP/Desktop/LP/III/LP-3/A2/A2.py', wdir='C:/Users/HP/Desktop/LP/III/LP-3/A2')
Test ok!
```

//ASSIGNMENT 07
//CODE

```
from random import randint
if __name__ == '__main__':
        # Both the persons will be agreed upon the
        # public keys G and P
        # A prime number P is taken
        P = 23
        # A primitive root for P, G is taken
        G = 9
        print('The Value of P is :%d'%(P))
        print('The Value of G is :%d'%(G))
        # Alice will choose the private key a
        a = 4
        print('The Private Key a for Alice is :%d'%(a))
        # gets the generated key
        x = int(pow(G,a,P))
        # Bob will choose the private key b
        b = 3
        print('The Private Key b for Bob is :%d'%(b))
        # gets the generated key
        y = int(pow(G,b,P))
        # Secret key for Alice
        ka = int(pow(y,a,P))
        # Secret key for Bob
        kb = int(pow(x,b,P))
        print('Secret key for the Alice is : %d'%(ka))
        print('Secret Key for the Bob is : %d'%(kb))
```

//OUTPUT

```
The Value of P is :23
The Value of G is :9
The Private Key a for Alice is :4
The Private Key b for Bob is :3
Secret key for the Alice is : 9
Secret Key for the Bob is : 9
```

//ASSIGNMENT 08
//CODE

```
def gcd(a, b): # calculates GCD of a and d
    while b != 0:
        c = a % b
        a = b
        b = c
    return a
def modinv(a, m): # calculates modulo inverse of a for mod m
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return None
def coprimes(a): # calculates all possible co-prime numbers with a
    l = []
    for x in range(2, a):
        if gcd(a, x) == 1 and modinv(x, phi) != None:
            l.append(x)
    for x in l:
        if x == modinv(x, phi):
            l.remove(x)
    return l
def encrypt_block(m): # encrypts a single block
    c = m ** e % n
    return c
def decrypt_block(c): # decrypts a single block
    m = c ** d % n
    return m
def encrypt_string(s): # applies encryption
    return ''.join([chr(encrypt_block(ord(x))) for x in list(s)])
def decrypt_string(s): # applies decryption
    return ''.join([chr(decrypt_block(ord(x))) for x in list(s)])
if __name__ == "__main__":
    p = int(input('Enter prime p: '))
    q = int(input('Enter prime q: '))
    print("Choosen primes:\np=" + str(p) + ", q=" + str(q) + "\n")
    n = p * q
    print("n = p * q = " + str(n) + "\n")
    phi = (p - 1) * (q - 1)
    print("Euler's function (totient) [phi(n)]: " + str(phi) + "\n")
    print("Choose an e from a below coprimes array:\n")
    print(str(coprimes(phi)) + "\n")
    e = int(input())
    d = modinv(e, phi) # calculates the decryption key d
    print("\nYour public key is a pair of numbers (e=" + str(e) + ", n=" + str(n) + ").\n")
    print("Your private key is a pair of numbers (d=" + str(d) + ", n=" + str(n) + ").\n")
    s = input("Enter a message to encrypt: ")
    print("\nPlain message: " + s + "\n")
    enc = encrypt_string(s)
    print("Encrypted message: ", enc, "\n")
    dec = decrypt_string(enc)
    print("Decrypted message: " + dec + "\n")
```

//OUTPUT

Enter prime p: 7
Enter prime q: 11
Choosen primes:
p=7, q=11
n = p * q = 77
Euler's function (totient) [phi(n)]: 60
Choose an e from a below coprimes array:
[7, 13, 17, 23, 31, 37, 43, 47, 53]
17
Your public key is a pair of numbers (e=17, n=77).
Your private key is a pair of numbers (d=53, n=77).
Enter a message to encrypt: Earth is round
Plain message: Earth is round
Encrypted message: 0<
EA?/A<" BC
Decrypted message: E %'
& %"(!