LAB 10

Aim : Implement following Digital signature algorithms (i) DSS (ii) ECC-DSS
Note: Clearly implement Signing and Verification algorithms.

Digital Signature Standard (DSS) scheme:

Code :

```
import random
e1,e2,p,q,d,S1,S2=0,0,0,0,0,0,0
def MultiplicativeInverse(a,n):
    t1=0
    t2=1
    if(a>n):
        r1=a
        r2=n
    else:
        r1=n
        r2=a
    while(r2>0):
        q=int(r1/r2)
        r=r1-q*r2
        t=t1-q*t2
        r1=r2
        r2=r
        t1=t2
        t2=t
    t=t1
    gcd=r1
    if(gcd==1):
        if(t<0):
            multiplicateInverse=t%n
```

```
        else:
            multiplicateInverse=t
    else:
        return "not possible because gcd("+str(a)+","+str(n)+")!=1",gcd
    return multiplicateInverse,gcd

def Multiply_and_Square(a,x,n):
    y = 1
    binary = bin(x).split('b')
    binary_of_x = binary[1]
    reverse_binary_of_x = binary_of_x[::-1]
    for i in range(0,len(reverse_binary_of_x)):
        if(reverse_binary_of_x[i]=='1'):
            y = (y*a)%n
        a = (a*a)%n
    return y

def Primitive_roots(p):
    Zp = []
    for i in range(1,p):
        multiplicateInverse,gcd = MultiplicativeInverse(i,p)
        if(gcd==1):
            Zp.append(i)
    phi_of_p = p-1
    ord_a = []
    temp =[]
    temp.append(-1)
    for a in range(1,p):
        minimum = 0
        for i in range(1,p):
            temp.insert(i,Multiply_and_Square(a,i,p))
```

```
            if(temp[i] == 1):
                if(minimum == 0):
                    minimum = i
                if(minimum > i):
                    minimum = i
        ord_a.insert(a,minimum)
    primitive_roots = []
    for i in range(0,len(ord_a)):
        if(ord_a[i]==phi_of_p):
            primitive_roots.append(i+1)
    return primitive_roots, Zp

def Key_Generation(p,q):
    print('Key Generation:')
    primitive_roots,Zp = Primitive_roots(p)
    e1 = random.choice(primitive_roots)
    e1 = Multiply_and_Square(e1, int((p-1)/q) , p )
    d = random.randrange(1,p-1)
    e2 = Multiply_and_Square(e1,d,p)
    print("p:",p)
    print("q:",q)
    print("e1:",e1)
    print("e2:",e2)
    print("d:",d)
    print('\n')
    return p,q,e1,e2,d

def Signing(msg,p,q,e1):
    r = random.randint(1,(p-2) )
    tmp = Multiply_and_Square(e1,r,p)
```

```
    S1 = tmp % q
    multiplicateInverse,gcd = MultiplicativeInverse(r,q)
    S2 = ( ( msg + (d*S1)) *  multiplicateInverse) % q
    print('Signature:')
    print('S1:',S1)
    print('S2:',S2)
    print('\n')
    return S1,S2

def Verifying(msg,p,q,e1,e2,S1,S2):
    t2 = S1
    S2_inverse,gcd = MultiplicativeInverse(S2,q)
    t1 = ((Multiply_and_Square(e1,(msg*S2_inverse),p) *
Multiply_and_Square(e2,(S1*S2_inverse),p)) % p ) %q

    print('Verification:')
    print("t1:",t1)
    print("t2 = S1:",t2)
    if(t1==t2):
        print("\nt1=t2, Hence verification is successful and message is accepted")
    else:
        print("\nVerification not successful")
    return

if __name__=="__main__":
    p=751; #prime number
    q=5; #prime divisor of (p-1)
    p,q,e1,e2,d=Key_Generation(p,q)
    msg = 105
    print('Message')
    print("Message: " + str(msg))
```

```
    S1,S2=Signing(msg,p,q,e1)
    Verifying(msg,p,q,e1,e2,S1,S2)
```

Output :

```
Key Generation:
p: 751
q: 5
e1: 460
e2: 80
d: 179


Message
Message: 105
Signature:
S1: 2
S2: 1


Verification:
t1: 2
t2 = S1: 2

t1=t2, Hence verification is successful and message is accepted
```

Elliptical Curve Digital Standard Scheme (ECDSS) Scheme :

Code :

```
import random
import math
def isPerfectSquare(x):
    if(x >= 0):
        temp = math.sqrt(x)
        return ((temp*temp) == x)
    return False


def MultiplicativeInverse(n,a):
    t1=0
    t2=1
    if(a>n):
        r1=a
        r2=n
```

```
    else:
        r1=n
        r2=a
    while(r2>0):
        q=int(r1/r2)
        r=r1-q*r2
        t=t1-q*t2
        r1=r2
        r2=r
        t1=t2
        t2=t
    t=t1
    gcd=r1
    multiplicateInverse=t
    if(gcd==1):
        if(t<0):
            multiplicateInverse=t+n
        else:
            multiplicateInverse=t
    return gcd,multiplicateInverse


def Addition_of_points(p,q,n,a):
    if(p[0] != q[0]):
        gcd,inverse = MultiplicativeInverse(n, (q[0]-p[0])%n)
        if gcd:
            lambdaa = ( (q[1] - p[1]) * int(inverse))%n
            x3 = (lambdaa**2- p[0]-q[0])%n
            y3 = (lambdaa*(p[0]-x3)-p[1])%n
            return [x3,y3]
        else:
            print("Inverse of x2-x1 does not exist")
```

```
    elif p[0] == q[0] and p[1] == q[1]:
        gcd,inverse = MultiplicativeInverse(n,2*p[1]%n)
        if gcd:
            lambdaa = ((3*p[0]**2+a)*inverse) % n
            x3 = (lambdaa**2-p[0]-q[0]) % n
            y3 = (lambdaa*(p[0]-x3)-p[1]) % n
            return [x3,y3]
        else:
            print("Inverse of x2-x1 does not exist")
    elif p[0] == q[0] and p[1] == -q[1]:
        return p
    return "Problem in point"


def Scalar_Multiplication_of_points(p,d,n,a):
    if d==0:
        return ""
    elif d == 1:
        return p
    else:
        if d%2 == 0:
            e = Scalar_Multiplication_of_points(p,d//2,n,a)
            return Addition_of_points(e,e,n,a)
        else:
            temp = d-1
            e = Scalar_Multiplication_of_points(p,temp//2,n,a)
            temp2 = Addition_of_points(e,e,n,a)
            return Addition_of_points(temp2,p,n,a)


def Multiply_and_Square(a,x,n):
    y = 1
    binary = bin(x).split('b')
```

```
    binary_of_x = binary[1]
    reverse_binary_of_x = binary_of_x[::-1]
    for i in range(0,len(reverse_binary_of_x)):
        if(reverse_binary_of_x[i]=='1'):
            y = (y*a)%n
        a = (a*a)%n
    return y

def Elliptic_Curve_Points(a,b,p):
    x=0
    points = []
    while(x<p):
        l1 = ((x*x*x) + (a*x) + b) % p
        find_pow = (p-1) // 2
        if(pow(l1,find_pow) % p == 1):
            while(isPerfectSquare(l1) == False):
                l1 += p
            if(isPerfectSquare(l1)):
                y1 = math.sqrt(l1)
                y2 = -(y1)
                points.append((x,int(y1 % p)))
                points.append((x,int(y2 % p)))
        x+=1
    return points

def Key_Generation(a,b,p,q,d):
    print('Key Generation: ')
    points = Elliptic_Curve_Points(a, b, p)
    e1 = points[random.randint(0, len(points)-1)]
    d = random.randint(1, p-1)
    e2 = Scalar_Multiplication_of_points(e1, d, p, a)
```

```
    print("p:",p)
    print("q:",q)
    print("e1:",e1)
    print("e2:",e2)
    print("d:",d)
    print('\n')
    return e1,e2

def Signing(msg,p,q,d,e1):
    while True:
        r = random.randint(1,(q-2) )
        point_p = Scalar_Multiplication_of_points(e1,r,p,a)
        if(point_p[0]%q != 0 ):
            break
    S1 = point_p[0] % q
    gcd,inverse=MultiplicativeInverse(r,q)
    S2 = ( inverse * ( msg + (d*S1)) ) % q
    print('Signature: ')
    print('S1:',S1)
    print('S2:',S2)
    print('\n')
    return S1,S2

def Verifying(msg,a,p,q,d,e1,e2,S1,S2):
    gcd,S2_inverse = MultiplicativeInverse(S2,q)
    t1 = (msg*S2_inverse) % q
    t2 = (S2_inverse * S1) % q
    print("t1:",t1)
    print("t2:",t2)
    tmp1 = Scalar_Multiplication_of_points(e1,t1,p,a)
    tmp2 = Scalar_Multiplication_of_points(e2,t2,p,a)
```

```
    print("tmp1:",tmp1)
    print("tmp2:",tmp2)
    T = Addition_of_points(tmp1,tmp2,p,a)
    print('Verification:')
    print("t1:",t1)
    print("t2:",t2)
    print("T:",T)
    if(T[0]%q==S1%q):
        print("\nT[0]%q=S1%q, Hence verification is successful and message is accepted")
    else:
        print("\nVerification not successful")
    return

if __name__=="__main__":
    a=1
    b=1
    p=13
    d=3
    q=13
    e1,e2=Key_Generation(a,b,p,q,d)
    msg = 500
    print('Message')
    print("Message: " + str(msg))
    S1,S2=Signing(msg,p,q,d,e1)
    Verifying(msg,a,p,q,d,e1,e2,S1,S2)
```

Output :

```
Key Generation:
p: 13
q: 13
e1: (10, 6)
e2: [10, 7]
d: 2


Message
Message: 500
Signature:
S1: 10
S2: 10


t1: 11
t2: 1
tmp1: [10, 7]
tmp2: [10, 7]
Verification:
t1: 11
t2: 1
T: [10, 6]

T[0]%q=S1%q, Hence verification is successful and message is accepted
```