Exercício 3 - ECDSA

Neste exercício vamos implementar um ECDSA segundo uma curva elíptica definidas no **FIPS186-4**. Numa pesquisa encontramos os valores dos parâmetros de várias curvas, tendo optado pela curva **P-384**, para ir de acordo com a curva utilizado no trabalho prático 1.

Parâmetros

- Os parâmetros desta curva são:

In [1]:
```
FIPS = dict()
FIPS['P-384'] = {
    'p': 39402006196394479212279040100143613805079739270465446667948293404245721771496870329047266088258938001861606973112319,
    'n': 39402006196394479212279040100143613805079739270465446667946905279627659399113263569398956308152294913554433653942643,
    'seed': 'a335926aa319a27a1d00896a6773a4827acdac73',
    'c': '79d1e655f868f02fff48dcdee14151ddb80643c1406d0ca10dfe6fc52009540a495e8042ea5f744f6e184667cc722483',
    'b': 'b3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8edd3ec2aef',
    'Gx': 'aa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a385502f25dbf55296c3a545e3872760ab7',
    'Gy': '3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113b5f0b8c00a60b1ce1d7e819d7a431d7c90ea0e5f',
}
```

Chaves

Para gerar as chaves vamos primeiro obter o **Ponto Gerador** (**basepoint**) e a partir daí já podemos calcular as chaves.

Ponto gerador

Este ponto é um ponto que se assemelha ao início da curva.

Para calcular precisamos dos parâmetros `p` e `b` tabelados para a partir disso obtermos a curva elítica correspondente. Sabendo que a curva tem o formato: `y^2 = x^3 - 3x + b mod p` a nossa curva pode ser obttida através: `E = EllipticCurve(GF(p),[-3,b])`

In [2]:
```
def generateGeneratorPoint():
    p = FIPS['P-384']['p']
    b = ZZ(FIPS['P-384']['b'],16)

    E = EllipticCurve(GF(p),[-3,b])
    #print(E)

    gx = ZZ(FIPS['P-384']['Gx'], 16)
    gy = ZZ(FIPS['P-384']['Gy'], 16)
    generatorPoint = E((gx,gy))
    print('GeneratorPoint: ' + str(generatorPoint))
    return generatorPoint
```

Chave privada

A chave privada é facilmente obtida, visto que é apenas necessário que seja um valor aleatório entre 0 e `n`, onde `n` representa a ordem do **basepoint**.

In [3]:
```
def generatePrivateKey(n):
    d = ZZ.random_element(1,n-1)
    print('PrivateKey: ' + str(d))
    return d
```

Chave pública

A chave pública é obtida a partir do **basepoint** e da chave privada : `q = d * G`

In [4]:
```
def generatePublicKey(generatorPoint, d):
    q = d * generatorPoint
    print('PublicKey: ' + str(q))
    return q
```

Na função abaixo geramos o **basepoint** e depois a **chave privada** e a respetiva **chave pública**.

In [5]:
```
def generateKeys(n):
    generatorPoint = generateGeneratorPoint()
    d = generatePrivateKey(n)
    q = generatePublicKey(generatorPoint, d)
    return (generatorPoint,d,q)
```

Assinatura de uma mensagem

Para assinar uma mensagem precisamos da **chave privada**, do **basepoint**, do `n` e da mensagem `m`.

O primeiro passo é calcular `e : e = hash(m)`

Depois temos de calcular os pares de chave temporarios (`k,r`) tal que `r e s` , depois de computados, têm de ser maiores que 0.

Após isso calculamos `s : ( ( e + d*r ) / k ) mod n`

Terminados os cálculos, juntamos estes dois fatores para formar a assinatura: `assinatura = (r,s)`

Pares de chave temporários

- **k** - inteiro compreendido: 0 < k < n-1
- **r** - para calcular `r` precisamos de:
  - **R** - `k * basepoint`
  - **Rx** - Componente x de R
  - **r** - `Rx mod n`

In [6]:
```
def generateEphemeral(n,generatorPoint):
    while(true):
        k = ZZ.random_element(1,n-1)
        R = k*generatorPoint
        Rx = R[0]
        #print('Rx: ' + str(Rx))
        r = mod(Rx,n)
        if(r > 0):
            break
    #print('k: ' + str(k))
    #print('r: ' + str(r))
    return (r,k)
```

Na função em baixo vemos o processo de assinatura de uma mensagem `m`.

In [7]:
```
def signMessage(privateKey,n,generatorPoint,m):
    e = hash(m)

    while(true):
        (r,k) = generateEphemeral(n,generatorPoint)

        dr = privateKey*r
        s1 = e + dr
        s = power_mod(s1/k,1,n)
        if(s > 0):
            break
    signature = (r,s)
    print('Signature: ' + str(signature))
    return signature
```

Verificar Assinatura

Após a assinatura é necessário a operação de verificação de assinatura.

O primeiro passo é verificar os tamanhos de `r e s` :

- `0 < r < n`
- `0 < s < n`

Depois calculamos a hash da mensagem: `e = hash(m)` .

Depois calculamos:

- **w**: `w = s^(-1) mod n`
- **u1** : `u1 = (e*w) mod n`
- **u2** : `u2 = (r*w) mod n`
- **P**: `P = (u1*G) + (u2*q)`

A partir deste `P` , pegamos na sua componente x: `px = P[0]` .

A verificação da assinatura corresponde ao valor de verdade da comparação: `px == r mod n`

In [8]:
```
def verifySignature(signature,n,generatorPoint,publicKey,m):
    (r,s) = signature

    nr = n-r
    ns = n-s
    if(r<=0):
        print('r negativo!')
        return ''
    if(nr<=0):
        print('r maior que n!')
        return ''
    if(s<=0):
        print('s negativo!')
        return ''
    if(ns<=0):
        print('s maior que n!')
        return ''

    e = hash(m)

    w = power_mod(ZZ(s),-1,n)

    u1 = power_mod(e*w,1,n)
    u2 = power_mod(r*w,1,n)

    P1 =(u1*generatorPoint)
    P2 = (ZZ(u2)*(publicKey))
    P = P1+P2
    px = ZZ(P[0])
    #print('px: ' + str(px))
    r2 = mod(r,n)
    if(px==r2):
        return 'Assinatura válida'
    else:
        return 'Assinatura inválida'
```

In [9]:
```
def main():
    n = FIPS['P-384']['n']
    (generatorPoint,privateKey,publicKey) = generateKeys(n)
    # privateKey: d
    #publicKey: q
    m = b'Teste1'
    signature = signMessage(privateKey,n,generatorPoint,m)

    ver1 = verifySignature(signature,n,generatorPoint,publicKey,m)
    print(ver1)

    m2 = b'Teste2'
    ver2 = verifySignature(signature,n,generatorPoint,publicKey,m2)
    print(ver2)
```

In [10]:
```
main()
```

```
GeneratorPoint: (26247035095799689268623156744566981891852923491109213387815615900925518854738050089022388053975719786650872476732087 : 832571096148902998554675128952010817928785304886131559470920590248050319988441922443864376039294733307808651162787 1 : 1)
PrivateKey: 14999611940890415949892677496934663755160619559661734457385406090500285461364294685714177593732828588826292614964377
PublicKey: (38356239539061280559295253027596776409053173527846434327980116481614234960832146275096817321063885359745657986714655 : 1793371818299485588731286105334920071485131113490719798111646009576835956241194133432643864731702148702724385957217 : 1)
Signature: (28942175561032238902075224737620032476565108248748786905064954331365788513231411645991391898240976798141982687068775, 9569388589252713622592385943309847000080762154327147246231516782944273912956358613768422028028527659566392364964506)
Assinatura válida
Assinatura inválida
```

In [ ]: