Exercício 2 - DSA

Neste exercício temos de implementar um algoritmo de DSA.

Esta implementação pode-se dividir em 4 processos distintos:

- **Gerar parâmetros**
- **Criação das chaves**
- **Assinatura de uma mensagem**
- **Verificação da assinatura de uma mensagem**

Ao longo deste documento vamos explicar o algoritmo de cada um dos processos acima expostos.

Gerar Parâmetros

Neste processo vamos gerar os 3 parâmetros essenciais que são a base deste algoritmo de DSA: `p`, `q`, e `g`.

Parâmetro q:

- **q** - Um número primo de N-bits

In [1]:
```
def generateQ(N):
    q = random_prime(2^(N-1))
    print("q: "+ str(q))
    return q
```

Parâmetro p:

- **p:**
  - Número primo de L-bits
  - `p-1` é múltiplo de `q`

In [2]:
```
def generateP(L,q):
    while(true):
        p = random_prime(2^(L-1))
        if(mod(ZZ(p-1),q) == 0):
            break
        #print('next')
    print("p: "+ str(p))
    return p
```

Parâmetro g :

- **g**
  - Temos de computar: `h^((p-1)/q) mod p`

Como vemos na expressão em cima é necessário um parâmetro auxiliar: `h` .

De realçar que se com um determinado valor de `h` , a computação de `g` for igual a 1, devemos repetir esta computação, mas assumindo `h=2` .

Parâmetro h :

- **h**
  - Inteiro compreendido entre os valores: 1 < h < p-1

In [3]:
```
def generateH(p):
    h = randint(2,p-2)
    print("h: "+ str(h))
    return h
```

In [4]:
```
def generateG(p,q):
    h = generateH(p)
    exph = (p-1)/q
    g = power_mod(h,ZZ(exph),p)
    if (g == 1):
        print('g = 1 ...')
        h = 2
        g = pow(h,aux,p)
    print('g: ' + str(g))
    return g
```

Gerador de parâmetros:

`generateParameters()` é a função é responsável por gerar os parâmetros:

- **p**
- **q**
- **g**

In [5]:
```
def generateParameters(L,N):
    q = generateQ(16)
    p = generateP(128,q)
    g = generateG(p,q)
    parameters = (p,q,g)
    return parameters
```

In [6]:
```
parameters = generateParameters(2048,256)
```

```
q: 22279
p: 144398932945256214146235797358038236531
h: 681522073872587233592443799031021352555
g: 793112386268288241311576196117740382955
```

Criação das chaves

Agora que temos os parâmetros necessários, podemos passar para o processo de criação das chaves. Assim na função seguinte vamos gerar dois valores:

- **x**
  - Inteiro aleatório compreendido no intervalo: 0 < x < q
  - Corresponde à **Chave Privada**
- **y**
  - Computar `g^x mod p`
  - Corresponde à **Chave Pública**

In [7]:
```
def generateKeys(parameters):
    (p,q,g) = parameters
    #x
    x = randint(1,q-1)
    privateKey = x
    print('Private Key: ' + str(privateKey))
    y = power_mod(g,privateKey,p)
    publicKey = y
    print('Public Key: ' + str(publicKey))
    return privateKey, publicKey
```

In [8]:
```
privateKey, publicKey = generateKeys(parameters)
```

```
Private Key: 3203
Public Key: 19747773144800009831469514182127128013
```

Assinatura de uma mensagem

Com os parâmetros gerados e criadas as chaves, podemos passar à produção de assinaturas de qualquer mensagem `m`. Este processo pode ser dividido em 3 processos mais pequenos:

- **Calcular r** - `(g^k mod p) mod q`. Se `r==0`, gerar outro `k` e repetir este processo
- **Calcular s** - `[ k^(-1) * ( hash(m) + x*r ) ] mod q`
- **Construir assinatura** - Par: `(r,s)`

Calcular r

Este cálculo necessita de um valor auxiliar `k`.

Como foi referido em cima `r` tem de ser diferente de `0`. Por isso no raro caso de isso acontecer, será necessário gerar outro `k` e recalcular `r`.

Calcular k

- **k** - Inteiro aleatório compreendido no intervalo: 0 < k < q

In [9]:
```
def generateK(q):
    k = randint(1,q-1)
    print('k: ' + str(k))
    return k
```

In [10]:
```
def calculateR(parameters,k):
    (p,q,g) = parameters
    r = mod(power_mod(g,k,p),q)
    if (r == 0):
        print('r = 0')
        generateK(q)
        r = calculateR(parameters,k)
    print('r: ' + str(r))
    return r
```

Calcular s

Para calcular `s` precisamos do `k` em cima gerado também.

Para processar a expressão `[ k^(-1) * ( hash(m) + x*r ) ] mod q` vamos dividir em tarefas mais pequenas, devido ao grande esforço computacional que requerem.

Primeiro vamos computar: `k^(-1) mod q` e só depois multiplicar esse valor por `(hash(m) + x*r)`

In [11]:
```
def calculateS(parameters,privateKey,k,m,r):
    (p,q,g) = parameters
    hm = hash(m)
    #print('hm: ' + str(hm))
    s1 = power_mod(k,-1,q)
    xr = privateKey*r
    s2 = hm + xr
    s = s1*s2
    print('s: ' + str(s))
    return s
```

Construir assinatura

Este processo é bastante simples na medida que apenas pega nos valores de `r` e `s` e junta-os como um par: `signature = (r,s)`

Na função em baixo, `signMessage` vamos executar os 3 processos descritos em cima e vamos devolver a assinatura.

In [12]:
```
def signMessage(parameters,privateKey,m):
    (p,q,g) = parameters
    k = generateK(q)
    r = calculateR(parameters,k)
    s = calculateS(parameters,privateKey,k,m,r)
    signature = (r,s)
    print('signature: ' + str(signature))
    return signature
```

In [13]:
```
m = b'teste12'
signature = signMessage(parameters,privateKey,m)
```

```
k: 8868
r: 12542
s: 18608
signature: (12542, 18608)
```

Verificação da assinatura de uma mensagem

Esta operação diz respeito à validação de uma assinatura a uma dada mensagem.

Este processo de verificação também pode ser dividido em algumas etapas:

- **Tamanho de `r` e `s`**
  - 0 < r < q
  - 0 < s < q
- **Cálculo de w** - `s^(-1) mod q`
- **u1** - `hash(m) * w mod q`
- **u2** - `r*w mod q`
- **v** - `( g^(u1) * y^(u2) mod p ) mod q`
- **Verificar** : `v == r`

In [14]:
```
def verifySignature(publicKey,parameters,signature,m):
    (p,q,g) = parameters
    #print('q: ' + str(q))
    (r,s) = signature
    qr = q-r
    #print('qr: ' + str(qr))
    qs = q-s
    #print('qs: ' + str(qs))
    if(r<=0):
        #print('r negativo')
        return 'ERRO R!'
    if(qr <= 0):
        #print('r>=q')
        return 'ERRO R!'
    if(s<=0):
        #print('s negativo')
        return 'ERRO S!'
    if(qs <= 0):
        #print('s>=q')
        return 'ERRO S!'

    w = power_mod(ZZ(s),-1,q)
    #print('w: ' + str(w))

    mw = hash(m) * w
    #print('mw: ' + str(mw))
    u1 = power_mod(mw,1,q)
    #print('u1: ' + str(u1))

    rw = ZZ(r)*w
    #print('rw: ' + str(rw))
    u2 = mod(rw,q)
    #print('u2: ' + str(u2))

    gu1 = g^u1
    pu2 = publicKey^u2
    gu1pu2=gu1*pu2

    v1 = mod(gu1pu2,p)
    #print('v1: ' + str(v1))

    v = mod(v1,q)
    print('v: ' + str(v))

    if(v == r):
        return 'True'
    else:
        return 'False'
```

In [15]:
```
result = verifySignature(publicKey,parameters,signature,m)
print('Resultado da verificação 1: ' + result)
```

```
v: 12542
Resultado da verificação 1: True
```

In [16]:
```
m2 = b'Teste2'
result2 = verifySignature(publicKey,parameters,signature,m2)
print('Resultado da verificação 2: ' + str(result2))
```

```
v: 2555
Resultado da verificação 2: False
```