Universidade do Minho
Escola de Engenharia

Estruturas Criptográficas

# Iniciação aos Corpos Finitos Primos

## Curvas Elípticas sobre esses corpos e Esquemas Criptográficos baseados nos mesmos

***Submitted To:***
José Valença
Professor Catedrático
Tecnologias da Informação e Segurança

***Submitted By :***
Diogo Araújo, A78485
Diogo Nogueira, A78957
Group 4

# Conteúdo

# 1 Implementação do esquema KEM-RSA-OAEP

## 1.1 Descrição do Exercício

A ideia do exercício passa por criar toda uma classe em Python que seja capaz de implementar o esquema **KEM-RSA-OAEP**.

Dado que os algoritmos e a forma como funcionam são públicos, o grupo apenas precisou de compreender como cada um deles funciona, transformando essa ideia para modo **SageMath**.

## 1.2 Descrição da Implementação

Estando feita estra triagem de informação inicial, o grupo penso desde logo na ideia de criar duas classes. Uma que respondesse aos pedidos do OAEP e outra que fizesse depois a continuidade em modo RSA, possibilitando dessa forma a criação das chaves pública e privada e toda a parte de crifragem e decifragem.

**Com a pesquisa necessária e com a ideia do funcionamento do algortimo em mente, estabelecem-se as seguintes implementacões:**

- **Passos da algoritmia do OAEP (Codificação e Descodificação):**

  **(a) Para codificar (*padding*):**

  1. Mensagens ficam com um padding de k1-zeros para ficarem com o tamanho $n - k0$ bits;
  2. $r$ é uma string random de tamanho $k0$ bits;
  3. $G$ expande os $k0$ bits do $r$ para $n - k0$ bits;
  4. $X$ = *m00...0* $\oplus G(r)$;
  5. $H$ reduz os $n - k0$ bits do X para $k0$ bits;
  6. $Y$ = $r \oplus H(X)$;
  7. O output é $X||Y$.

  **Assim a mensagem pode ser agora cifrada pelo RSA. A propriedade determinística do RSA é evitada usando o encoding OAEP.**

**(b) Para descodificar (*unpadding*):**

1. Recuperar a random string $r$ = Y $\oplus$ $H(X)$;
2. Recuperar a messagem *m00...0* = X $\oplus$ $G(r)$.

- **Passos da algoritmia do RSA:**

1. Gerar primos aleatórios pela utilização da função do **SageMath** que nos oferece um primo até ao limite superior (1º argumento), bem como com o limite inferior (3º argumento);

   **Assim neste caso temos um primo** $2^{b-1} < primo < 2^b - 1$.
2. Criação dos primos "$p$"e "$q$", bem como o módulo $n$;
3. Computação rápida de Fórmula de Euler de n, conhecendo $p$ e $q$;

   **Assim,** $\varphi(n) = (p-1)(q-1)$.
4. Criação do ring dos inteiros modulo phi e a escolha aleatória dum inteiro para ser o $e$;

   **Assim, escolhe-se um inteiro que** $1 < e < \varphi(n)$ **e** $gcd(e, \varphi(n)) = 1$**;** $e$ **e** $\varphi(n)$ **são co-primos.**
5. Criação do $d$. Como $d \equiv e^{-1} \pmod{\varphi(n)}$, então utilizamos o algoritmo Euclideano extendido para passar para esta forma: $1 = de - k \cdot \varphi(n)$ e assim fica na identidade de *Bézout*: $g = gcd(x, y) = sx + ty$;

   **Assim, o trio da variável vai ser $(1, d, -k).**
6. Com todos os números criados, temos assim as chaves criadas em SageMath.

**Estando todos estes algoritmos definidos e entendidos, desenvolveram-se os metodos necessários para cada classe Python e através de um mini teste pode-se verificar a verdade de toda esta implementação.**

## 1.3 Resolução do Exercício

### Classe Python OAEP

```
In [1]: import random

        class OAEP():

            # Função que inicializa toda a instância e os valores globais␣
 ↪necessários à sua execução
```

```python
    # Recebe o valor do módulo de n como parâmetro.
    def __init__(self, n):

        self.n = n # Tamanho do RSA modulus (key length)
        self.k0 = 128 # Tamanho-bits da string r (random)


    def string_to_strbits(self, string):

        strbits = ''.join(format(ord(i), 'b') for i in string)

        return strbits

    def strbits_to_string(self, strbits):

        string = ''

        for i in range(0, len(strbits), 7):

            tempcharbit = strbits[i:i + 7]
            decimalChar = int(tempcharbit, 2)
            string = string + chr(decimalChar)

        return string

    def pad(self, mensagem):

        # Passar mensagem para String bits
        m_strbits = self.string_to_strbits(mensagem)

        # O valor de k1 é n-k0-tamanhoMensagem (para depois fazer padding de zeros, caso seja preciso)
        m_tamanho = len(m_strbits)

        self.k1 = self.n-self.k0-m_tamanho

        # Efetuar o padding de zeros à mensagem (String Bits)
        m_strbits_pad = m_strbits + ('0'*self.k1)

        # Criação da string random de k0-bits
        r_number = random.getrandbits(self.k0)
        r_strbits = str(bin(r_number))[2:]

        while len(r_strbits) != self.k0:
```

```python
            r_strbits = r_strbits + '0' # Só para o raro caso de r não ser perto de k0-bits

        # Expandir o r para (n-k0) bits (tamanho do X) --- G(r)
        tam_expansao = (self.n - 2*self.k0)
        r_exp_strbits = r_strbits + ('0'*tam_expansao)

        # Efetuar o XOR da mensagem pad e o G(r) --- X
        x_strbits = str(bin(int(m_strbits_pad,2) ^^ int(r_exp_strbits,2))[2:]).zfill(len(m_strbits_pad))

        # Truncar o X para k0 bits --- H(X)
        h_x_strbits = x_strbits[0:self.k0]

        # Efetuar o XOR do r com o H(x) --- Y
        y_strbits = str(bin(int(r_strbits,2) ^^ int(h_x_strbits,2))[2:]).zfill(len(r_strbits))

        return x_strbits+y_strbits


    def unpad(self, x, y):

        # Truncar o X para k0 bits --- H(X)
        h_x_strbits = x[0:self.k0]

        # Efetuar o XOR do Y com o H(x) --- r
        r_strbits = str(bin(int(y,2) ^^ int(h_x_strbits,2))[2:]).zfill(len(y))

        # Expandir o r para (n-k0) bits (tamanho do X) --- G(r)
        tam_expansao = (self.n - 2*self.k0)
        g_r_strbits = r_strbits + ('0'*tam_expansao)

        # Efetuar o XOR do X com o G(r) --- mensagemPad
        mensagemPad = str(bin(int(x,2) ^^ int(g_r_strbits,2))[2:]).zfill(len(x))

        tamMensagem = self.n - self.k0 - self.k1

        mensagem_strbits = mensagemPad[0:tamMensagem]
        mensagem = self.strbits_to_string(mensagem_strbits)

        return mensagem
```

### Teste Classe Python OAEP

```
In [2]: myOAEP = OAEP(1024)
        rsa = myOAEP.pad("TesteString")

        x = rsa[:896]
        y = rsa[896:]

        mensagem = myOAEP.unpad(x, y)

        print "X: {} com o tamanho-bit de {}".format(x, len(x))
        print "Y: {} com o tamanho-bit de {}".format(y, len(y))
        print "RSA: {} com o tamanho-bit de {}".format(rsa, len(rsa))

        print "Mensagem: {}".format(mensagem)
```

```
X: 0010010001000010000010100011011001100100111101100111000001000100101011011
101000000111010101110001110110000101110111010101011010100000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
com o tamanho-bit de 896
Y: 1010100110010111100111110100110010110100111101001110010110100111101110110
0111000000000000000000000000000000000000000000000000000
com o tamanho-bit de 128
RSA: 00100100010000100000101000110110011001001111011001110000010001001010110
1110100000011101010111000111011000010111011101010101101010000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
(...) com o tamanho-bit de 1024
Mensagem: TesteString
```

### Classe Python RSA

```
In [3]: class RSA():

            # Função que inicializa toda a instância e os valores globais␣
 ↪necessários à sua execução
            def __init__(self, n):

                self.keylength = n

            def randomprime(self, bits):
                return random_prime(2**bits-1,True,2**(bits-1))

            def generatekeys(self):

                t = self.keylength/2

                self.modulus = 0

                while len(str(bin(self.modulus)[2:])) != self.keylength:

                    # Valor dos primos "p" e "q"
                    p = self.randomprime(t)
                    q = self.randomprime(t)

                    # Valor do Módulo N
                    self.modulus = p * q

                self.r = IntegerModRing(self.modulus)


                # Valor do phi(n)
                phi = (p-1)*(q-1)

                e = ZZ.random_element(phi)

                while gcd(e, phi) != 1: ## e tem de ser coprimo de phi
                    e = ZZ.random_element(phi)

                # G = IntegerModRing(phi)
                # e = G(randomprime(512))

                # O S é igual ao D dado que o inverse multiplicative module␣
 ↪é igual a seguir o teorema de bezout.
```

```
            # Apenas colocado aqui para nível pedagógico
            # s = 1/e
            bezout = xgcd(e,phi)
            d = Integer(mod(bezout[1],phi));

            print "Tamanho do N (key-length): " + str(len(str(bin(self.
 ↪modulus)[2:])))

            # RSA public key
            print "Chave pública: (e: {})".format(e)

            # RSA private key
            print "Chave privada: (d: {})".format(d)

            return (e, d)

        def cifrar(self, m, e):

            a = self.r(m)

            # cm = (m ^ e) % self.modulus

            cm = a**e

            return cm

        def decifrar(self, cm, d):

            b = self.r(cm)

            # dm = (cm ^ d) % self.modulus

            dm = b**d

            return dm
```

### Teste Classe Python RSA

```
In [4]: tamanhoN = 256

        '''
        rsamsg = int(rsamsg_strbits, 2)

        myRSA = RSA(tamanhoN)
```

```
        (pubkey, privkey) = myRSA.generatekeys()

        ciphertext = myRSA.cifrar(rsamsg, pubkey)

        print ciphertext

        rsamensagem = myRSA.decifrar(ciphertext, privkey)

        print rsamensagem

        rsamensagem_strbits = str(bin(int(rsamensagem,2))[2:]).
 ↪zfill(len(rsamensagem))

        x = rsamensagem[:128]
        y = rsamensagem[128:]

        mensagemfinal = myOAEP.unpad(x,y)

        print mensagemfinal

        '''

        myRSA = RSA(tamanhoN)

        (pubkey, privkey) = myRSA.generatekeys()

        ciphertext = myRSA.cifrar(110, pubkey)

        mensagem = myRSA.decifrar(ciphertext, privkey)

        print "RSA funcionou? ",mensagem == 110
```

```
Tamanho do N (key-length): 256
Chave pública: (e:␣
 ↪45442704347296726502485820465662804526282074652382115438007791410159709848499)
Chave privada: (d:␣
 ↪50635373912691781259367731963541570719007932522770514866395590312930160020887)
RSA funcionou?  True
```

## 1.4 Referências


- Wikipedia, *Optimal asymmetric encryption padding* https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding (Acedido a 18 abril 2020)
- Wikipedia, *Ciphertext indistinguishability* https://en.wikipedia.org/wiki/Ciphertext_indistinguishability (Acedido a 18 abril 2020)
- Wikipedia,RSAhttps://en.wikipedia.org/wiki/RSA_(cryptosystem) (Acedido a 18 abril 2020)

# 2 Implementação do esquema DSA

## 2.1 Descrição do Exercício

**Construir uma classe Python que implemente o DSA. A implementação deve:**

1. A iniciação, receber como parâmetros o tamanho dos primos $p$ e $q$;
2. Conter funções para assinar digitalmente e verificar a assinatura.

## 2.2 Descrição da Implementação

Para a inicialização do DSA existiu a criação duma classe que tivesse como parâmetros de construção os tamanhos dos primos $p$ e $q$, também comummente chamados de $L$ e $N$, com valores recomendados de $(1024, 160), (2048, 224), (2048, 256), (3072, 256)$.

1. **Geração dos parâmetros DSA:**
   - $q$ é um **número primo** de tamanho $2^{N˘1} < p < 2^N$;
   - $p$ é um **número primo** de tamanho $2^{L˘1} < p < 2^L$, tal que $(p-1)$ tem de ser múltiplo de $q$;
   - $g$ é um gerador tal que $g := h^{(p-1)/q} \mod p$.
2. **Geração das chaves DSA:**

   Após existirem os parâmetros DSA $(p, q, g)$ podemos assim criar as chaves DSA da seguinte forma:
   - Seleção dum inteiro que $\in \{1 \ldots q-1\}$ que servirá de **chave privada** DSA;
   - A geração da **chave pública** utilizando a seguinte equação $g^x \mod p$.
3. **Assinatura DSA:**

   Agora com as chaves DSA já geradas, podemos efetuar uma assinatura duma mensagem simples, usando para tal efeito a **chave privada** criada anteriormente.
   - Seleção dum inteiro $k$ que $\in \{1 \ldots q-1\}$;
   - A geração do $r$ sendo que $r := (g^k \mod p) \mod q$;
   - A geração do $s$ sendo que $s := (k^{-1}(H(m) + xr)) \mod q$.

   Ficando assim com o par que representa a assinatura DSA $(r, s)$.

4. **Verificação DSA:**

   Para verificar que a assinatura DSA $(r, s)$ é válida para a mensagem $m$ segue-se os seguintes passos:

   - Verifica-se se $0 < r < q$ e $0 < s < q$;
   - Gera-se $w := s^{-1} \bmod q$;
   - Gera-se $u_1 := H(m) \cdot w \mod q$;
   - Gera-se $u_2 := r \cdot w \mod q$.
   - Gera-se $v := (g^{u_1} y^{u_2} \bmod p) \bmod q$

   A assinatura é válida para a mensagem se e só se $v == r$.

## 2.3 Resolução do Exercício

```
In [1]: from random import randint


        class DSA():

            # Função que inicializa toda a instância e os valores globais␣
 ↪necessários à sua execução
            def __init__(self, l, n):

                self.tamprimop = l
                if n <= 256:
                    self.tamprimoq = n
                else: self.tamprimoq = 256 # Por causa do tamanho de N <= |␣
 ↪H | e a hash a usar é SHA-256


            def gerarparametros(self):

                # Escolha de primos grandes
                self.p = random_prime ( 2^self.tamprimop , proof=False ,␣
 ↪lbound=2^(self.tamprimop-1) )
                print "P: ",self.p

                self.q = random_prime ( 2^self.tamprimoq , proof=False ,␣
 ↪lbound=2^(self.tamprimoq-1) )

                print "Q: ",self.q
```

```
            h = Integer(1, self.p-1)

            self.g = ( ( h ^ ( (self.p-1) / self.q ) ) % self.p)

        def gerarchaves(self):

            privkey = randint(1, self.q-1) # privkey é um inteiro de {1 .. q-1}

            pubkey = power_mod(self.g, privkey, self.p) # pubkey = g^x mod p

            return (privkey, pubkey)

        def assinar(self, privkey, mensagem):

            print "--- A assinar a mensagem: {} ---".format(mensagem)

            # Criação do inteiro random k
            k = randint(1,self.q-1)

            # Criar o par da assinatura DSA
            r = ( ( self.g^k ) % self.p ) % self.q
            s = ( ( k^-1 ) * ( hash ( mensagem ) + privkey * r ) ) % self.q

            # Assinatura = ( r , s )
            return (r,s)


        def verificar(self, assinatura, pubkey, mensagem):

            print "--- A verificar a assinatura para a mensagem: {} ---".format(mensagem)

            r = assinatura[0]
            s = assinatura[1]

            w = (s^-1) % self.q

            u1 = (hash (mensagem) * w) % self.q
            u2 = (r * w) % self.q

            v = ( ( (self.g^u1) * (pubkey^u2) ) % self.p ) % self.q
```

```
                if v == r: print "--- Assinatura verificada para a mensagem.
 ↪ ---"
                else: print "--- Assinatura não corresponde à mensagem. ---"
```

#### Teste da Classe

Segue-se abaixo um teste desenvolvido para esta classe. **A ideia passa pelos seguintes passos:**

- Gerar um par de chaves (privkey,pubkey);
- Assinar essa mensagem inicial com a *Private Key* X;
- Criar uma nova mensagem diferente;
- Verificar a assinatura com essa nova mensagem.

```
In [2]: # Criar uma instância e sua inicialização
        myDSA = DSA(2048, 256)

        # Gerar os parâmetros de instância (p, q, g)
        myDSA.gerarparametros()

        # Obter o par de chaves DSA
        (privkey, pubkey) = myDSA.gerarchaves()

        # Assinar uma pequena mensagem
        assinatura =  myDSA.assinar(privkey, "Pequena mensagem")

        # Verificar se a assinatura é da mensagem
        myDSA.verificar(assinatura, pubkey, "Pequena Mensagem")
```

```
P: ␣
 ↪267557916636702751005062208257692016299278666040350638463769679960161784495259
302727460122935736080093975317572466106905102085054477333476249029511083825581910
639536565398978742357996795452128941909813649113416195584038645936351716734531952
494606590527704663809293329490166636236041228593274198214602339023815764477469011
561174102706250185444984697459909528297850413883857805776886889420167001346338037
944340301715053801373416635255919114261655092553992342057688672695764847588232913
000422504830859760971001938112359303253772770791989136609739466989117245522629463
72383121357705741389749040481553173643925910965737 49
Q: ␣
 ↪10443306703380092314944990557137253307060515656695695579789461461218363246272 9
--- A assinar a mensagem: Pequena mensagem ---
--- A verificar a assinatura para a mensagem: Pequena Mensagem ---
--- Assinatura verificada para a mensagem. ---
```

## 2.4 Observações Finais

- O algoritmo **DSA** encontra-se detalhadamente descrito em várias fontes *online*;
- Depois de compreender o seu funcionamento torna-se mais simples de começar a aplicar toda a ideia em SageMath.

## 2.5 Referências


- Wikipedia, SHA-2 `https://en.wikipedia.org/wiki/SHA-2` (Acedido a 14 de Abril 2020)
- Python, HashLib package `https://docs.python.org/3/library/hashlib.html` (Acedido a 14 de Abril 2020)
- Wikipedia, Digital Signature Algorithm `https://en.wikipedia.org/wiki/Digital_Signature_Algorithm` (Acedido a 14 abril 2020)

# 3 Implementação do ECDSA com a Curva Elíptica prima P-192

## 3.1 Descrição do Exercício

A ideia do exercício passa por criar toda uma classe em Python que seja capaz de implementar o ***Eliptic Curve Digital Signature Algorithm*** **(ECDSA)** com o uso de uma das curvas elípticas definidas no FIPS186-4.

Dessa forma, o grupo recorreu ao documento de seu nome **FIS PUB 186-4** que permitiu estabelecer a escolha desta curva. Escolheu-se assim a curva P-192, cujos seus valores *standard* são fornecidos pelo próprio documento em si.

## 3.2 Descrição da Implementação

Com a curva elíptica prima escolhida, podem-se definir o conjunto de definicões que a classe Python terá e que permitirão no final fazer um pequeno teste em termos de resultados. **Com a pesquisa necessária e com a ideia do funcionamento do algortimo em mente, estabelecem-se as seguintes implementacões:**

- **Criação do par das chaves em modo Curva Elíptica:**

  1. *Private Key* - $d_A$ - que corresponderá a um inteiro gerado aleatoriamente dentro do intervalo $[1, n-1]$;
  2. *Public Key* - $Q_A$ - que corresponderá a um ponto da Curva Elíptica tal que $Q_A = d_A \times G$ onde $G$ **é um ponto gerador da Curva Elíptica**.

- **Criação da assinatura da mensagem em si:**

  1. Cálculo do valor de $e = HASH(m)$, em que $HASH$ corresponderá a uma função de hash criptográfica em que o seu *return* se irá converter em um inteiro;

     **A função de** $HASH$ **será declara à parte**.

  2. Criação/Escolha de um inteiro $k$ gerado aleatoriamente dentro do intervalo $[1, n-1]$;
  3. Cálculo de um ponto da Curva tal que $(x1, y1) = k \times G$;

     $G$ **continua a ser o mesmo ponto gerador da Curva Elíptica definido inicialmente**.

4. Cálculo do valor de $r$ tal que $r = x_1 \bmod n$, sendo esse *mod* calculado com base nos ensinamentos de **Corpos Finitos** abordados no **Trabalho 1**;

   **Dessa forma, cria-se um Corpo Finito em torno do primo $n$ e aplica-se o valor de $x1$ a esse mesmo tal que $r = F_n(x_1)$.**

5. Cálculo do valor de $s$ tal que $r = x_1 \bmod n$;

   **Assume-se $z$ como sendo o valor calculado para $e$ e aplica-se o mesmo princípio do valor de $r$ tal que $s = F_n(k)^{-1} \times (e + r \times d)$.**

- **Verificação dessa mesma assinatura:**

1. Calcular $e = HASH(m)$, em que $HASH$ terá de ser a mesma função usada para criar a assinatura;

2. Calcular o valor de $w$ tal que $w = s^{-1} \bmod n$;

3. Calcular o valor de dois valores necessários para um ponto da Curva;

   **Calcular o valor de $u_1$ tal que $u_1 = zw \bmod n$;**

   **Calcular o valor de $u_2$ tal que $u_2 = rw \bmod n$.**

4. Com os dois valores de cima calcular um ponto da Curva tal que $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$;

5. Caso $r \equiv x_1 \pmod{n}$, a assinatura é dada como válida.

   **Aplicando-se o princípio dos Corpos Finitos a assinatura é valida se e só se $r \equiv F_n(x_1)$.**

**Estando todos estes algoritmos definidos e entendidos, desenvolveram-se os metodos necessários para classe Python e através de um mini teste pode-se verificar a verdade de toda esta implementação.**

### 3.3 Resolução do Exercício

```
In [1]: import hashlib

        class ECDSA():

            # Função que inicializa toda a instância e os valores globais␣
 ↪necessários à sua execução
            def __init__(self):
```

```
            # Parâmetros da Curva Elíptica P-192
            # Os parâmetros 'p' e 'n' são dados na forma decimal
            # Os restantes são dados na forma hexadecimal
            self.p = 
↪6277101735386680763835789423207666416083908700390324961279
            self.n = 
↪6277101735386680763835789423176059013767194773182842284081
            self.a = -3
            self.b = 0x64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1
            self.Gx = 0x188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012
            self.Gy = 0x07192B95FFC8DA78631011ED6B24CDD573F977A11E794811

            # Corpo Finito Fp em torno do primo p
            self.Fp = FiniteField(self.p)

            # Corpo Finito Fn em torno do primo n
            self.Fn = FiniteField(self.n)

            # Curva Elíptica E
            self.E = EllipticCurve(self.Fp, [self.a, self.b])

            # Ponto Gerador G pertencente à Curva Elíptica E
            self.G = self.E((self.Gx,self.Gy))

        # Função de HASH criptográfica
        # Input: Mensagem para a qual se quer calcular o valor de HASH
        # Output: Devolve o valor de HASH
        def HASH(self, m):
            message = str(m)

            return Integer('0x' + hashlib.sha1(message).hexdigest())

        # Algoritmo de criação do par de chaves da Curva Elíptica
        # Input: Não recebe qualquer valor como Input
        # Output: Um par de chaves (Q, d)
        def keyGenerator(self):

            d = randint(1, self.n-1) # Valor Aleatório entre [1, n-1]
            Q = d * self.G

            return (Q, d)

        # Algoritmo de criação da assinatura
        # Input: Chave Privada da entidade que quer assinar a mensagem,
↪Mensagem a ser assinada
```

```
        # Output: Devolve o par [r,s] a ser usado na verificação da␣
↪assinatura
        def createSignature(self, d, m):

            e = self.HASH(m)

            # Variáveis que vão controlar os dois ciclos
            r = 0
            s = 0

            while s == 0:
                # k = 1 ACHO QUE NAO E PRECISO
                while r == 0:

                    # Cálculo do valor de k
                    k = randint(1, self.n-1)

                    # Cálculo do ponto da Curva Elíptica
                    P = k * self.G
                    (x1, y1) = P.xy()

                    # Cálculo do valor de r
                    r = self.Fn(x1)

                # Cálculo do valor de s
                s = self.Fn(k) ^ (-1) * (e + r*d)

            return [r,s]

        # Algoritmo de verificação da assinatura
        # Input: Chave Pública, Mensagem e par [r,s] obtidos na␣
↪assinatura da Mensagem Inicial/Original
        # Output: Resultado da verificação
        def verifySignature(self, Q, m, r, s):

            e = self.HASH(m)

            w = s ^ (-1)

            u1 = (e * w)
            u2 = (r * w)

            P1 = Integer(u1) * self.G
            P2 = Integer(u2) * Q
```

```
        P = P1 + P2
        (x1, y1) = P.xy()

        return r == self.Fn(x1)
```

### Teste da Classe

Segue-se abaixo um teste desenvolvido para esta classe. **A ideia passa pelos seguintes passos:**

- Gerar um par de chaves (Q,d);
- Criar uma mensagem inicial **"A mensagem que vou pedir para assinar"**;
- Assinar essa mensagem inicial com a *Private Key* $d_A$;
- Criar uma nova mensagem **"Vou tentar uma nova mensagem"**;
- Verificar a assinatura com essa nova mensagem.

**Isto vai permitir verificar que o resultado é *false* dado que não corresponde à mensagem inicialmente assinada pela chave em si.**

In [2]:
```
# Criar uma instância e sua inicialização
myECDSA = ECDSA()

# Obter o par de chaves da Curva Elíptica
(Q, d) = myECDSA.keyGenerator()

print "Eliptic Curve Public Key - ", Q.xy()
print "Eliptic Curve Private Key - ", d

# Assinar a mensagem
originalMessage = 'A mensagem que vou pedir para assinar'
[r, s] = myECDSA.createSignature(d, originalMessage)

print "Mensagem Original - ", originalMessage

# Verificar a assinatura da mensagem
fakeMessage = 'Vou tentar uma nova mensagem'
result = myECDSA.verifySignature(Q, fakeMessage, r, s)

print "Resultado - ", result
```

```
Eliptic Curve Public Key - ␣
 ↪(2038752860592359119426082346061557950338888622821805043945,␣
 ↪306303741022623073521196141447092461082295559180875188991)
Eliptic Curve Private Key - ␣
 ↪2009916947488637617038617525489375040057193637677791988684
Mensagem Original -  A mensagem que vou pedir para assinar
Resultado -  False
```

## 3.4 Observações Finais

- O algoritmo **ECDSA** encontra-se detalhadamente descrito em várias fontes *online*;
- Depois de compreender o seu funcionamento torna-se mais simples de começar a aplicar toda a ideia em SageMath.

## 3.5 Referências


- Cameron F. Kerry, Acting Secretary,Patrick D. Gallagher. (julho 2013). FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION, Digital Signature Standard (DSS) https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
- Wikipedia, Elliptic Curve Digital Signature Algorithm https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm (Acedido a 13 abril 2020)