Enunciado

Utilizando o SageMath

2. Construir uma classe Python que implemente o EcDSA a partir do “standard” FIPS186-5
   A. A implementação deve conter funções para assinar digitalmente e verificar a assinatura.
   B. A implementação da classe deve usar uma das “Twisted Edwards Curves” definidas no standard e escolhida na iniciação da classe: a curva “edwards25519” ou “edwards448”.

In [1]:
```
import hashlib
import random
from sage.all import *
```

Exercício 2.2.

-p: Primo grande, define o corpo finito GF(p)

-a, d: Parâmetros da curva Edwards

Curva de Edwards "twisted": $ax^2 + y^2 = 1 + dx^2y^2$

Conversão da curva de Edwards para a curva de Weierstrass:

$A = \frac{2(a+d)}{a-d}$

$B = \frac{4}{a-d}$

-Calcula a4, a6 para definir a equação da curva $y^2 = x^3 + a_4x + a_6$ (fórmula de Weierstrass)

In [2]:
```
class EcDSA_Ed25519:
    def __init__(self, p, a, d):
        assert a != d and is_prime(p) and p > 3
        K = GF(p) # Corpo finito de p

        A =  2*(a + d)/(a - d) #Centraliza a curva ao converter o eixo x
        B =  4/(a - d)  #Normaliza os valores, ajusta a escala da curva

        self.alfa = A/(3*B) #Deslocamento no eixo X na transformação de Edwards para Weiers
        self.s = B #Ajusta a escala da curva

        a4 =  self.s**(-2) - 3*self.alfa**2
        a6 =  -self.alfa**3 - a4*self.alfa

        self.EC = EllipticCurve(K,[a4,a6])


        #Ponto base (ponto de partida para a geração de chaves)
        self.Px = K(151122213495354007725011514095885315114540126930418572060461132839498477
        self.Py = K(463168356949264781694283940034751631413079938662562256157830336031652518

        self.L = ZZ(2**252 + 27742317777372353535851937790883648493) #Ordem do grupo de pon
        self.P = self.ed2ec(self.Px, self.Py)

        self.private_key = self.generate_private_key()
        self.public_key = self.generate_public_key()
```

```python
    def generate_private_key(self):
        return randint(1, self.L - 1) #Chave privada aleatória

    def generate_public_key(self):
        return self.private_key * self.P #Chave pública gerada a partir da chave privada

    def ed2ec(self,x,y):      ## mapeia Ed --> EC
        if (x,y) == (0,1):
            return self.EC(0)
        z = (1+y)/(1-y) ; w = z/x
        alfa = self.alfa; s = self.s
        return self.EC(z/s + alfa , w/s)
```

Exercício 2.1.

Etapas do ECDSA

Gerar Assinatura

Dada uma mensagem m, a assinatura $\sigma = (r, s)$ é gerada da seguinte forma:

Escolher um número aleatório k:

k é um número aleatório gerado a cada assinatura.

Esse número k deve ser escolhido de forma segura para garantir que não seja reutilizado em duas assinaturas diferentes.

Calcular o ponto $(x_1, y_1) = k.P$:

O número x_1 (coordenada x) do ponto (x_1, y_1) na curva elíptica é usado para gerar o valor r. Se $r = 0$, escolhe-se outro valor de k e recalcula-se.

$r = x_1 mod \ \ L$, onde L é a ordem do ponto P.

Calcular s:

A variável s é calculada usando a chave privada d, a mensagem m (convertida para um valor numérico, usando um hash), e o número aleatório k: $s = k^{-1}(h(m) + rd) mod \ \ L$

Onde:

-$h(m)$ é o hash da mensagem.

-k^{-1} é o inverso multiplicativo de k módulo L.

A assinatura é então o par (r, s).

Validar assinatura

Para verificar a assinatura $\sigma = (r, s)$ de uma mensagem m é necessário uma chave pública Q

Validar r e s:

r e s devem ser números no intervalo $[1, n-1]$. Se não forem, a assinatura é inválida.

Calcular valores intermédios:

-$h(m)$ é o hash da mensagem m

-Calcular $w = s^{-1} mod L$, o inverso de s módulo L

-Calcular $u_1 = h(m) w mod L$ e $u_2 = r.w mod L$

Calcular o ponto $(x_1, y_1) = u_1.P + u_2.Q$**:**

-Se $x_1 mod \quad n = r$, então a assinatura é válida

In [3]:
```
def sign(e, message):
        h = hashlib.sha512(message).digest()
        h_int = int.from_bytes(h, 'big') % e.L

        while True:
            k = randint(1, e.L - 1)
            R = k * e.P

            r = int(R[0]) % e.L
            if r == 0:
                continue

            k_inv = pow(k, -1, e.L)
            s = (k_inv * (h_int + r * e.private_key)) % e.L

            if s == 0:
                continue

            return (r, s)

def verify(e, message, signature):
    r, s = signature

    if not (1 <= r < e.L and 1 <= s < e.L):
        return False

    h = hashlib.sha512(message).digest()
    h_int = int.from_bytes(h, 'big') % e.L

    k_inv = pow(s, -1, e.L)
    u1 = (h_int * k_inv) % e.L
    u2 = (r * k_inv) % e.L

    R_prime = u1 * e.P + u2 * e.public_key

    return int(R_prime[0]) % e.L == r
```

Teste

In [4]:
```
p = 2**255 - 19
K = GF(p)
a = K(-1)
d = K(-121665) / K(121666)

ecdsa = EcDSA_Ed25519(p, a, d)
message = b"Ola!"
signature = sign(ecdsa, message)
is_valid = verify(ecdsa, message, signature)

print(f"Assinatura válida? {is_valid}")
```

```
Assinatura válida? True
```