Exercicio 3

In [46]:
```
import os
import math
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.hashes import Hash,SHA256
from sage.all import *
import hashlib
from sage.crypto.util import ascii_to_bin, bin_to_ascii
```

Introdução

Neste exercicio foi nos pedido construir uma classe EdCSA segundo a norma do FIPS186-5 usando curvas de edwards. A curva utilizada foi a 22519.

In [47]:
```
# Edwards 22519
p = (2^255)-19
K = GF(p)
a = K(-1)
d = -K(121665)/K(121666)
#

ed25519 = {
'b'  : 256,
'Px' : K(151122213495354007725011514095885315114540126930418572060461132
'Py' : K(463168356949264781694283940034751631413079938662562256157830360
'L'  : ZZ(2^252 + 27742317777372353535851937790883648493), ## ordem do su
'n'  : 254,
'h'  : 8
}
```

Para a inicialização da classe usamos o exemplo fornecido pelo professor sobre a curva 22519. Apenas criamos o metodo para a criação da chave publica, sendo esta um ponto gerada pelo gerador P.

In [48]:
```
class Ed(object):
    def __init__(self,p, a, d , ed = None):
        assert a != d and is_prime(p) and p > 3
        K = GF(p)
        A =  2*(a + d)/(a - d)
        B =  4/(a - d)

        alfa = A/(3*B) ; s = B

        a4 =  s^(-2) - 3*alfa^2
        a6 =  -alfa^3 - a4*alfa

        self.K = K
        self.b=ed['b']
        self.constants = {'a': a , 'd': d , 'A':A , 'B':B , 'alfa':alfa ,
        self.EC = EllipticCurve(K,[a4,a6])
        self.n=ed['n']

        if ed != None:
            self.L = ed['L']
            self.P = self.ed2ec(ed['Px'],ed['Py'])
        else:
            self.gen()

    def order(self):
        # A ordem prima "n" do maior subgrupo da curva, e o respetivo cof
        oo = self.EC.order()
        n,_ = list(factor(oo))[-1]
        return (n,oo//n)

    def gen(self):
        L, h = self.order()
        P = O = self.EC(0)
        while L*P == O:
            P = self.EC.random_element()
        self.P = h*P ; self.L = L

    def is_edwards(self, x, y):
        a = self.constants['a'] ; d = self.constants['d']
        x2 = x^2 ; y2 = y^2
        return a*x2 + y2 == 1 + d*x2*y2

    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.constants['alfa']; s = self.constants['s']
        return self.EC(z/s + alfa , w/s)

    def ec2ed(self,P):        ## mapeia EC --> Ed
        x,y = P.xy()
        alfa = self.constants['alfa']; s = self.constants['s']
        u = s*(x - alfa) ; v = s*y
        return (u/v , (u-1)/(u+1))



    def generate_public_key(self):

        self.private_key = H(os.urandom(32))

        a1 = 2^(self.b-2) + sum(2^i * bit(self.private_key,i) for i in ra
```

```
        key_msg=key_hashed+msg
        s=0
        r=0

        e= hashing(key_msg)
        r = mod(e,self.n)
        r_int=ZZ(r)

        #3

        R  = r_int*self.P

        #4
        msg_intermedia=R+self.public_key_point
        msg_total=str(msg_intermedia).encode('utf-8')+msg
        msg_hashed = hashing(msg_total)
        h= mod(msg_hashed,self.n)

        #5
        s=mod(r_int+ZZ(h)*bytes_to_int(self.private_key),self.n)

        #6
        return R, s

    def verify(self,msg,R,s):
        #1
        msg_intermedia=R+self.public_key_point
        msg_total=str(msg_intermedia).encode('utf-8')+msg
        msg_hashed = hashing(msg_total)
        h= mod(msg_hashed,self.n)

        #2
        P1=ZZ(s)*self.P

        #3
        P2=R+(ZZ(h)*(self.public_key_point))

        #4
        print(P1==P2)
```

In [49]:

```
def bytes_to_int(bytes):
    result = 0
    for b in bytes:
        result = result * 256 + int(b)
    return result

def H(m):
        return hashlib.sha512(m).digest()


def hashing(m):
    return int(hashlib.sha512(str(m).encode("utf-8")).hexdigest(), 16)

def bit(h,i):
```

Para a parte do sign seguimos os seguintes passos:

1.Calcular a chave publica que é um ponto gerado através da chave privada e do gerador.

2.Geramos deterministacamente um segredo que basicamente consistiu na concatenação da mensagem com a chave privada onde nesta foi aplicada uma função hash, a sha 512 como referido na norma.Após isto aplicou-se a função hash a concatenação destas duas e o mod de n tendo assim o r.

3.Calculou-se o ponto R multiplicando o ponto r gerado anteriormente pelo gerador.

4.Calculou-se o h que é a concatenação do ponto R,da chave publica e de mensagem,posteriormente aplicado a função de hash e realizando o mod de n

5.Calculou-se o que é igual a seguinte expressão: s = (r + h * privKey) mod n

6.Por fim retornamos o ponto R e o s.

Assumindo que de input recebemos o s, o R gerados no sign e a mensagem, para a parte do verify seguimos os seguintes passos:

1.Calculou-se o h da mesma forma que se calculou no sign

2.Para o ponto P1 multiplicamos o gerador P pelo parametro s

3.Para o o ponto P2 somamos o parametro R a multiplicação da chave privada pelo h.

4.No fim retorna-se a igualdade entre o P1 e o P2.

In [51]:
```
E = Ed(p,a,d,ed25519)
print(E.EC)
print()
print(E.order())
Px = ed25519['Px']; Py = ed25519['Py']
print(E.is_edwards(Px,Py))
E.ed2ec(Px,Py)

E.generate_public_key()
R,s=E.sign(b"ola")
E.verify(b"ola",R,s)
```

```
Elliptic Curve defined by y^2 = x^3 + 422041017956698223164489531199450479457090990152259961749339889434781241894 85*x + 1314834172054291958757092074419044647942534449144043611621331643553417295939 6 over Finite Field of size 57896044618658097711785492504343953926634992332820282019728792003956564819949

(72370055773322622139731865630429942408571163593799076060019509382854542509989, 8)
True
False
```

Conclusão

A partida tudo indicaria que deveria funcionar.Porém os pontos obtidos são diferentes. O grupo fez o esforço para tentar encontrar o problema mas sem sucesso, uma vez que seguimos os passos todos indicados para a aplicação da EdCSA. Suspeita-se que seja no modo das operações, como somas e multiplicações que tenham de ser feitas de forma diferente.