FinTech HW3-Blockchain secp256k1

name: 張皓鈞
student ID: R08922125

Prepare

先設定secp256k1協定中橢圓曲線的數字。其中$P = mG$，m 為private key。

```
# The proven prime
Pcurve = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1

# Number of points in the field
N = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

# This defines the curve. y^2 = x^3 + Acurve * x + Bcurve
Acurve = 0
Bcurve = 7

Gx = 55066263022277343669578718895168534326250603453777594175500187360389116729240
Gy = 32670510020758816978083085130507043184471273380659243275938904335757337482424
GPoint = (Gx, Gy)

# replace with any private key
privKey = 2125

```

為了在橢圓曲線上做double/add運算，以下為運算推導

Addition:

$$(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$$

Doubling:

$$(x_3, y_3) = [2]\,(x_1, y_1)$$

$$s = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} \bmod p & \text{(addition)} \\ \dfrac{3x_1^2 + a}{2y_1} \bmod p & \text{(doubling)} \end{cases}$$

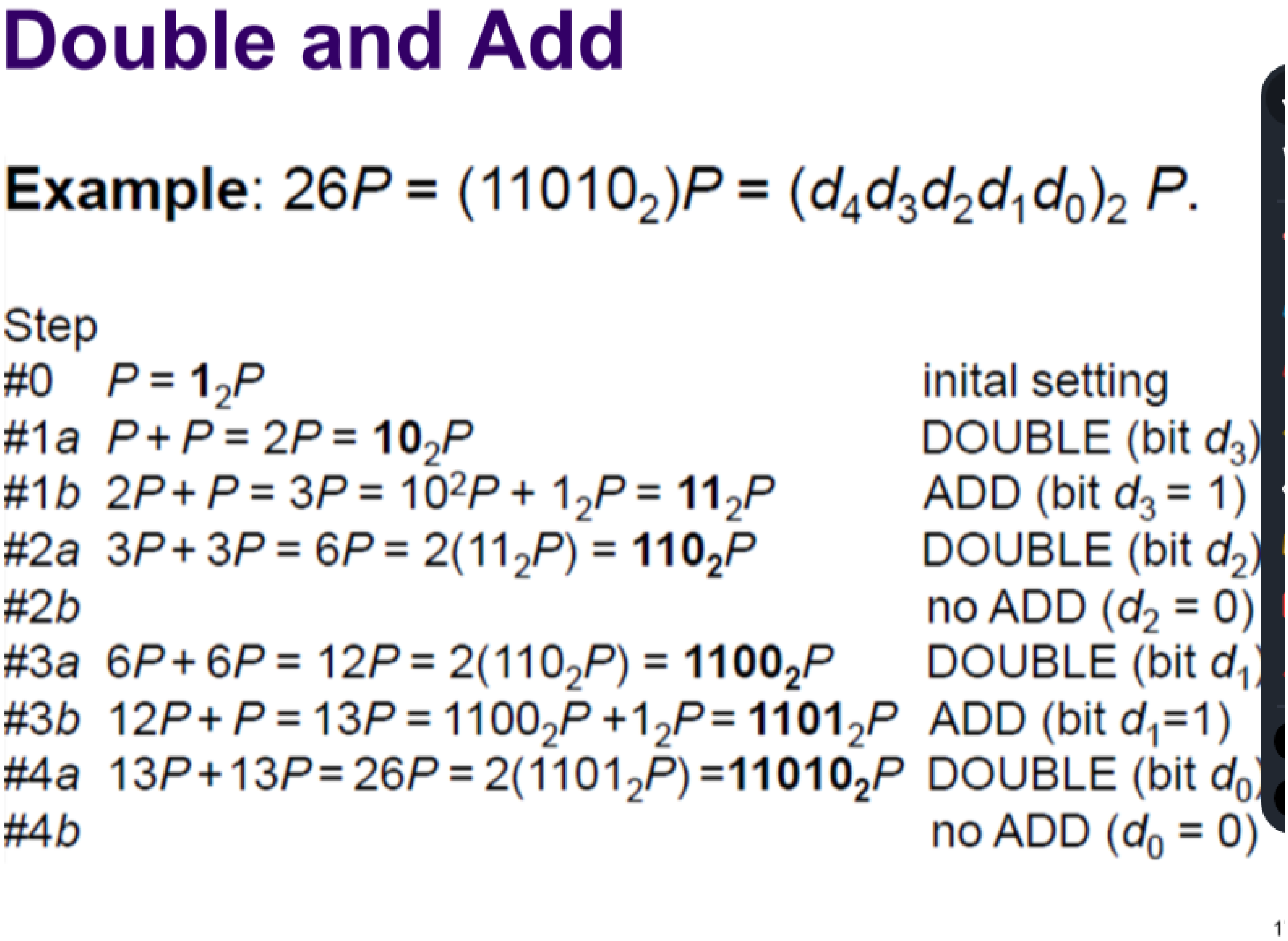
$$x_3 = s^2 - x_1 - x_2 \mod p$$

$$y_3 = s(x_1 - x_3) - y_1 \mod p$$

其中上式的除法為在mod prime field中找反元素，這裡使用Extended Euclidean Algorithm division以求出反元素。

```python

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)


def modinv(a, m):# Extended Euclidean Algorithm/'division' in elliptic curves
    if a < 0:
        a += m
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m
def ECadd(x1, y1, x2, y2):
    LamNumer = y2 - y1
    LamDenom = x2 - x1
    s = (LamNumer * modinv(LamDenom, Pcurve)) % Pcurve
    x3 = (s * s - x1 - x2) % Pcurve
    y3 = (s * (x1 - x3) - y1) % Pcurve
    return (x3, y3)


# EC point doubling, invented for EC. It doubles Point-P.
def ECdouble(x1, y1):
    LamNumer = 3 * x1 ** 2 + Acurve
    LamDenom = 2 * y1
    s = (LamNumer * modinv(LamDenom, Pcurve)) % Pcurve
    x3 = (s * s - 2 * x1) % Pcurve
    y3 = (s * (x1 - x3) - y1) % Pcurve
```

定義好橢圓曲線上的各運算後，只需將m拆解成二進制$(10...10)_2$，即可進行double and add。




```python
def EccMultiply(xs, ys, Scalar):  # Double & add. EC Multiplication, Not true mult
    if Scalar == 0 or Scalar >= N:
        raise Exception("Invalid Scalar/Private Key")

    ScalarBin = str(bin(Scalar))[2:]

    Qx, Qy = xs, ys
    for i in range(1, len(ScalarBin)):  # This is invented EC multiplication.
        Qx, Qy = ECdouble(Qx, Qy)  # print "DUB", Qx; print
        # print("DUB")
        if ScalarBin[i] == "1":
            # print ("ADD")
            Qx, Qy = ECadd(Qx, Qy, xs, ys)  # print "ADD", Qx; print

    return (Qx, Qy)
```

1. Evaluate 4G

將private key設為4，可得
xPublicKey:
$(103388573995635080359749164254216598308788835304023601477803095234286494993683)$

yPublicKey:
$(37057141145242123013015316630864329550140216928701153669873286428255828810018)$

2. Evaluate 5G

將private key設為5，可得
xPublicKey:
$(21505829891763648114329055987619236494102133314575206970830385799158076338148)$
yPublicKey:
$(98003708678762621233683240503080860129026887322874138805529884920309963580118)$

3. Evalute dG, d = last 4 digits of student id

將private key設為2125，可得
xPublicKey:
$(101781878184007084671381261094362501380942865028961237641824038035511380961002)$
yPublicKey:
$(58369962163175720309519279668836655893250110774156566655796446087224164166873)$

4. How many doubles and adds required for dG

$d = 2125 = (100001001101)_2$
doubles: 二進制後為12位數，共11次
Adds: 除了開頭的1外有4個1，共4次

5. If effortless to find -P from P, try as fast as possible to evaluate dG

$d = 2125 = (100001001101)_2 = (100001010000)_2 - (11)_2$

若用等號右式去算doubles and adds會得到
$(100001010000)_2$: doubles=11次、adds=2次
$(11)_2$: doubles=1次、adds=1次
兩式相減為加上$(-11)_2$: adds=1次
以上共會有 doubles=11+1=12次，adds=2+1+1=4次，比原本還要多一次double，故沒辦法透過加上反元素來加速運算。

6. Sign the transaction with random number k and private key

Parameter	
CURVE	the elliptic curve field and equation used
G	elliptic curve base point, a generator of the elliptic curve with large prime order *n*
n	integer order of G, means that $n * G = O$

Suppose Alice wants to send a signed message to Bob. Initially, they must agree on the curve parameters $(CURVE, G, n)$. In addition to the field and equation of the curve, we need G, a base point of prime order on the curve; n is the multiplicative order of the point G.

Alice creates a key pair, consisting of a private key integer d_A, randomly selected in the interval $[1, n-1]$; and a public key curve point $Q_A = d_A * G$. We use $*$ to denote elliptic curve point multiplication by a scalar.

For Alice to sign a message m, she follows these steps:

1. Calculate $e = \mathrm{HASH}(m)$, where HASH is a cryptographic hash function, such as SHA-1.
2. Let z be the L_n leftmost bits of e, where L_n is the bit length of the group order n.
3. Select a random integer k from $[1, n-1]$.
4. Calculate the curve point $(x_1, y_1) = k * G$.
5. Calculate $r = x_1 \bmod n$. If $r = 0$, go back to step 3.
6. Calculate $\boxed{s = k^{-1}(z + rd_A) \bmod n}$. If $s = 0$, go back to step 3.
7. The signature is the pair (r, s).

k : ephemeral key

http://en.wikipedia.org/wiki/Elliptic_Curve_DSA

這邊先假設transaction or message已經透過hash得出結果，接著代入ECDSA signing的公式中。

```
RandNum = random.randrange(1, N-1)
# the hash of your message/transaction
HashOfThingToSign = 86032112319101611046176971828093669637772856272773459297323797

print ("******* Signature Generation *********")
xRandSignPoint, yRandSignPoint = EccMultiply(Gx, Gy, RandNum)
r = xRandSignPoint % N
print ("r =", r)
s = ((HashOfThingToSign + r * privKey) * (modinv(RandNum, N))) % N
print ("s =", s)
```

得到以下signature pair (privkey=2125)

r =

93772431501136186088668087526387751603797786880760459347166986734278045988868

s =

78306064907336131440146611767770888203084770145384663455006976421517049272379

7. Signature verification

ECDSA Verification 驗章

For Bob to authenticate Alice's signature, he must have a copy of her public-key curve point Q_A. Bob can verify Q_A is a valid curve point as follows:

1. Check that Q_A is not equal to the identity element O, and its coordinates are otherwise valid
2. Check that Q_A lies on the curve
3. Check that $n * Q_A = O$

After that, Bob follows these steps:

1. Verify that r and s are integers in $[1, n-1]$. If not, the signature is invalid.
2. Calculate $e = \mathrm{HASH}(m)$, where HASH is the same function used in the signature generation.
3. Let z be the L_n leftmost bits of e.
4. Calculate $w = s^{-1} \bmod n$.
5. Calculate $u_1 = zw \bmod n$ and $u_2 = rw \bmod n$.
6. Calculate the curve point $(x_1, y_1) = u_1 * G + u_2 * Q_A$.
7. The signature is valid if $r \equiv x_1 \pmod{n}$, invalid otherwise.

Note that using Straus's algorithm (also known as Shamir's trick) a sum of two scalar multiplications $u_1 * G + u_2 * Q_A$ can be calculated faster than with two scalar multiplications.[3]

32

http://en.wikipedia.org/wiki/Elliptic_Curve_DSA

```
print ("******* Signature Verification *********")
w = modinv(s, N)
xu1, yu1 = EccMultiply(Gx, Gy, (HashOfThingToSign * w) % N)
xu2, yu2 = EccMultiply(xPublicKey, yPublicKey, (r * w) % N)
x, y = ECadd(xu1, yu1, xu2, yu2)
if (r % N  == x % N):
    print("Verification Success")
else:
    print("Verification Fail")
```

將程式碼所有執行一遍得到output如下，因為簽章過程中的random integer k 為隨機選取，所以每次signature pair: (r, s) 輸出會不同，但不影響驗章結果。

```
******* Public Key Generation *********
the private key (in base 10 format):
2125
the public key:
xPublicKey: 1017818781840070846713812610943625013809428650289612376418240380355113
yPublicKey: 5836996216317572030951927966883665589325011077415656665579644608722416
******* Signature Generation *********
r = 40821342333621665316682207759328835203186422262565723048077324388716727957385
s = 33455016392615915206076738263827486215583963124742440863978264345580360798783
******* Signature Verification *********
Verification Success
```

8. Custruct quadratic polynomial $p(x)$ with $p(1) = 10, p(2) = 100, p(3) = d$, over Z_{10007}

由三個已知點可以得出一元二次多項式，這邊採用Lagrange Interpolation

- Lagrange Interpolation Formula

  $$p(x) = \sum_{i=0}^{k} p_i(x) = \sum_{i=0}^{k} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

  is the unique polynomial of degree $\leq k$ passing through the $k+1$ points (x_i, y_i), where $x_i \neq x_j$ for $i \neq j$

- Note that $p(x_i) = y_i$ since $p_i(x_i) = y_i$ and $p_j(x_i) = y_j \prod_{k \neq j} \frac{x_i - x_k}{x_j - x_k} = 0$

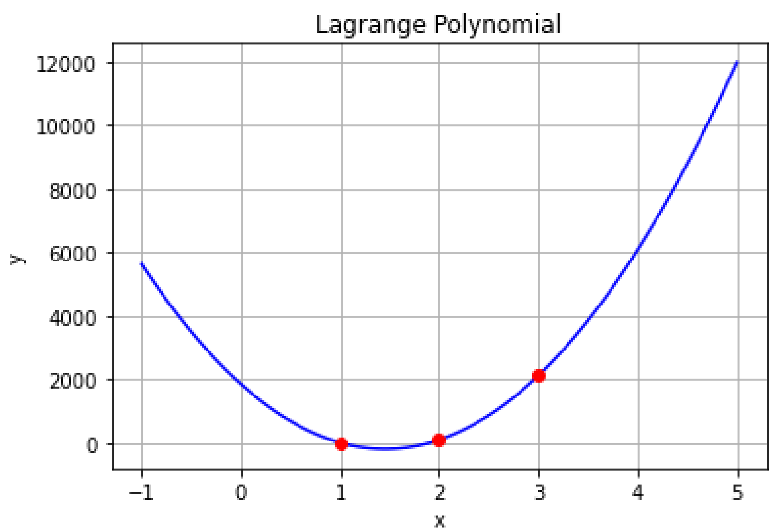
- Denote the factor of recovery $\prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$ by $r_i(x; x_0, \ldots, x_k)$

```
import numpy as np
from numpy.polynomial.polynomial import Polynomial
import matplotlib.pyplot as plt
from scipy.interpolate import lagrange

d = 2125
x = np.array([1 ,2  ,3])
y = np.array([10,100,d])
# type of f is poly1D[a,b,c] = ax^2+bx+c
f = lagrange(x, y)

# Polynomial coefficient parameters polynomial([a,b,c]) = a+bx+c^2
quadratic = Polynomial(f.coef[::-1])
```

```
fig = plt.figure()
x_new=range(5)
plt.plot(*quadratic.linspace(domain=[0,10007]), 'b', x,y,'ro')
plt.title('Lagrange Polynomial')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

得到quadratic:

$$967.5x^2 - 2812.5x + 1855$$



tags: **`FinTech`** **`Blockchain`** **`secp256k1`** **`elliptic curve`**