Fintech 2017 Homework

Bitcoin & Blockchain

- b03902089 資工四 林良翰
- Using $secp256k1$

```
import numpy as np
from pycoin.ecdsa import generator_secp256k1 as g
```

Problem 1

- Compute $4G$

```
x, y = (4 * g).pair()
print("x =", hex(x), "\ny =", hex(y))
```

```
x = 0xe493dbf1c10d80f3581e4904930b1404cc6c13900ee0758474fa94abe8c4cd13
y = 0x51ed993ea0d455b75642e2098ea51448d967ae33bfbdfe40cfe97bdc47739922
```

Problem 2

- Compute $5G$

```
x, y = (5 * g).pair()
print("x =", hex(x), "\ny =", hex(y))
```

```
x = 0x2f8bde4d1a07209355b4a7250a5c5128e88b84bddc619ab7cba8d569b240efe4
y = 0xd8ac222636e5e3d6d4dba9dda6c9c426f788271bab0d6840dca87d3aa6ac62d6
```

Problem 3

- Compute dG

```
x, y = (902089 * g).pair()
print("x =", hex(x), "\ny =", hex(y))
```

```
x = 0xa7209262b3a12654a736face446cdf0e4f1abd71aec71bc9fb8d129582164479
y = 0xa1d2f2ce0f3b301f54be8ef09a844066652a672dab090076071f2080437561 3f
```

Problem 4

- Double & Add Algorithm

```python
def Evaluate(steps):
    value = 1
    for s in steps:
        if s is 'a': value += 1
        if s is 'd': value *= 2
        if s is 's': value -= 1
    return value

def DoubleAdd(bstring):
    print("binary:", bstring)
    steps = ['init']
    for b in bstring[1:]:
        steps += ['d']
        if b == '1': steps += ['a']

    print("add:", steps.count('a'), "\ndouble:", steps.count('d'),
"\ntotal:", len(steps)-1)
    print("steps:", steps)
    print("decimal:", Evaluate(steps))

DoubleAdd(bin(902089)[2:])
```

```
binary: 11011100001111001001
add: 10
double: 19
total: 29
steps: ['init', 'd', 'a', 'd', 'd', 'a', 'd', 'a', 'd', 'a', 'd', 'd', 'd',
'd', 'd', 'a', 'd', 'a', 'd', 'a', 'd', 'a', 'd', 'd', 'd', 'a', 'd', 'd',
'd', 'a']
decimal: 902089
```

Problem 5

- Double & Add Algorithm with Substraction
- If there is a continuous 1 sequence with length $n > 2$, using add $\rightarrow$ double n times $\rightarrow$ substract is faster.

```python
def DoubleAddSub(bstring):

    def AddOnes(ones):
        if ones > 1:
            return ['a'] + ['d'] * ones + ['s']
        elif ones == 1:
            return ['d', 'a']

    print("binary:", bstring)
    ones, steps = 0, ['init']
    for b in bstring[1:]:
        if b == '1':
            ones += 1
        else:
            if (ones):
                steps += AddOnes(ones)
                ones = 0
            steps += ['d']
    if (ones):
        steps += AddOnes(ones)

    print("add:", steps.count('a'), "\ndouble:", steps.count('d'),
          "\nsubstract:", steps.count('s'), "\ntotal:", len(steps)-1)
    print("steps:", steps)
    print("decimal:", Evaluate(steps))

DoubleAddSub(bin(902089)[2:])
```

```
binary: 11011100001111001001
add: 5
double: 19
substract: 2
total: 26
steps: ['init', 'd', 'a', 'd', 'a', 'd', 'd', 'd', 's', 'd', 'd', 'd', 'd',
'a', 'd', 'd', 'd', 'd', 's', 'd', 'd', 'd', 'a', 'd', 'd', 'd', 'a']
decimal: 902089
```

Problem 6

- ECDSA Signing
- $nx + my = gcd(n, m)$
  $\Rightarrow$ we want to find: $x \ \ s.t. \ \ nx \equiv 1 \mod m$, where m is prime.

```
def ExtendedEuclidean(n, m):
    if (m == 0):
        return 1, 0
    else:
        x, y = ExtendedEuclidean(m, n % m)
        x, y = y, (x - (n // m) * y)
        return x, y

def ModularInverse(k, n): # inverse of k
    return ExtendedEuclidean(k, n)[0]
```

```
d = 902089 # private key
QA = d * g

# z = "qhan1028" (SHA256)
z = 0x38316DC32F31B3BC25DC18A61E682E86837877689209A3EC1562CE59E47CE13B
k = 1028 # random k, ephemeral key
x1, y1 = (k * g).pair()

n = g.order() # mod n
K = ModularInverse(k, n) # k inverse
r = x1 % n
s = K * (z + r * d) % n
print("r =", hex(r), "\ns =", hex(s))
```

```
r = 0xedd95e9da4ba6509dc91cc47c3acb8fec508450fcfa88a09b30a884b6bef8dfa
s = 0xf32318f16d3b452a8ea2034271bbffeefa1caddcd9552543ddd14bed20bcbd49
```

Problem 7

- ECDSA Verifying

```
x, y = QA.pair()
p = curve.p()
if ((y ** 2) % p == (x ** 3 + 7) % p): print("QA valid")

# z = "qhan1028" (SHA256)
z = 0x38316DC32F31B3BC25DC18A61E682E86837877689209A3EC1562CE59E47CE13B
w = ModularInverse(s, n)
u1 = z * w % n
u2 = r * w % n
x1, y1 = (u1 * g + u2 * QA).pair()
if (r % n == x1): print("Verified")
```

```
QA valid
Verified
```