HW03: Bitcoin/Blockchain R11944024 黃秉茂

Use the elliptic curve “secp256k1” as Bitcoin and Ethereum. Let G be the base point in the standard. Let d be the last 4 digits of your student ID number.

In [1]:

```
import pycoin.ecdsa.secp256k1 as secp256k1

G = secp256k1.secp256k1_generator
student_ID = 4024
```

In [2]:

```
def log_point(x, y, text=''):
    print('Point', text, '\nx: ', x, '\ny: ', y)
```

In [3]:

```
x = hex(G.raw_mul(1)[0])
y = hex(G.raw_mul(1)[1])
log_point(x, y, text='G')
```

```
Point G
x:  0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
y:  0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
```

1. Evaluate 4G

In [4]:

```
# (x_4, y_4) = G.raw_mul(4)
(x_4, y_4) = G * 4
log_point(hex(x_4), hex(y_4), text='4G')
```

```
Point 4G
x:  0xe493dbf1c10d80f3581e4904930b1404cc6c13900ee0758474fa94abe8c4cd13
y:  0x51ed993ea0d455b75642e2098ea51448d967ae33bfbdfe40cfe97bdc47739922
```

2. Evaluate 5G

In [5]:

```
# (x_5, y_5) = G.raw_mul(5)
(x_5, y_5) = G * 5
log_point(hex(x_5), hex(y_5), text='5G')
```

```
Point 5G
x:  0x2f8bde4d1a07209355b4a7250a5c5128e88b84bddc619ab7cba8d569b240efe4
y:  0xd8ac222636e5e3d6d4dba9dda6c9c426f788271bab0d6840dca87d3aa6ac62d6
```

3. Evaluate Q = dG

In [6]:

```
d = student_ID
(x_d, y_d) = G * d
log_point(hex(x_d), hex(y_d), text='Q')
```

```
Point Q
x:  0xdb25da2c9538aacb991c94cf0dcbbf152f00b80893c4005a25e3b4c3d9ad3ec
y:  0xf4a20005738a24bf9a59711c1c5ffc3d7c6efa778502db471a296949a6576a17
```

4. With standard Double-and Add algorithm for scalar multiplications, how many doubles and additions respectively are required to evaluate dG?

In [7]:

```python
def int_to_binary(int_value):
    # '0b{binary_value}'
    return bin(int_value)[2:]

def binary_to_int(bin_value):
    return int(bin_value, 2)
```

In [8]:

```python
d = student_ID
binary = int_to_binary(d)
print(d, 'G =', binary, 'G\n')
binary_str = str(binary)

n_double = 0
n_add = 0

number = int(binary_str[0])
print('initial\t', int_to_binary(number))
for bin_value in binary_str[1:]:
    if bin_value == '0':
        number <<= 1
        n_double += 1
        print('double\t', int_to_binary(number))
    elif bin_value == '1':
        number <<= 1
        n_double += 1
        print('double\t', int_to_binary(number))
        number += 1
        n_add += 1
        print('add\t', int_to_binary(number))

# print('\ndouble:', len(binary_str)-1)
print('\ndouble', n_double, 'times.')
# print('add:', binary_str.count('1')-1)
print('add', n_add, 'times.')
```

```
4024 G = 111110111000 G

initial  1
double  10
add  11
double  110
add  111
double  1110
add  1111
double  11110
add  11111
double  111110
double  1111100
add  1111101
double  11111010
add  11111011
double  111110110
add  111110111
double  1111101110
double  11111011100
double  111110111000
```

```
double 11 times.
add 7 times.
```

5. Note that it is effortless to find –P from any P on a curve. If the addition of an inverse point is allowed, try your best to evaluate dG as fast as possible. Hint: 31P = 2(2(2(2(2P)))) – P

In [9]:

```
def expansion(abbreviation):
    if abbreviation == 'a':
        return 'add'
    if abbreviation == 'd':
        return 'double'
    if abbreviation == 's':
        return 'subtract'

def check(integer, operations):
    number = 1
    for op in operations:
        if op == 'a':
            number += 1
        elif op == 'd':
            number <<= 1
        elif op == 's':
            number -= 1
        # print(int_to_binary(number))
    return integer == number

def reconstruct(operations):
    number = 1
    print('\ninitial  ', int_to_binary(number))
    for op in operations:
        if op == 'a':
            number += 1
        elif op == 'd':
            number <<= 1
        elif op == 's':
            number -= 1
        print(f'{expansion(op):<9}', int_to_binary(number))
```

In [10]:

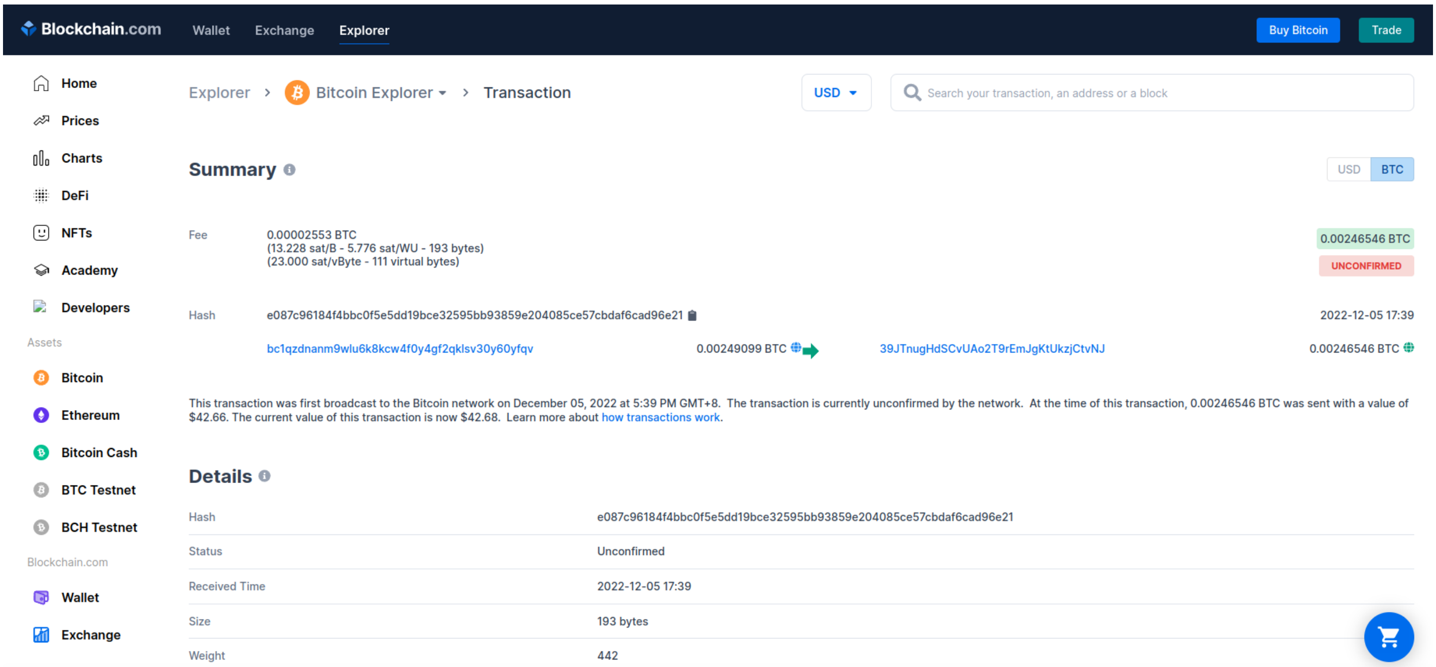
```
print('-' * 20, 'standard algorithm', '-' * 20 + '\n')
standard_operations = []
d = student_ID

while d > 1:
    if d & 0x1 == 1:
        standard_operations.append('a')
        d -= 1
    else:
        standard_operations.append('d')
        d >>= 1
standard_operations = standard_operations[::-1]
d = student_ID
binary = int_to_binary(d)
print(d, 'G =', binary, 'G\n')
print('double %d times.' % (standard_operations.count('d')))
print('add %d times.' % (standard_operations.count('a')))
print('total %d times.' % (len(standard_operations)))
print('\ndatailed operations steps:', list(map(expansion, standard_operations)))
reconstruct(standard_operations)
print()

print('-' * 20, 'optimized algorithm', '-' * 20 + '\n')
# build a replace list: 11*n -> 10*n+1 - 1 =>  da*n -> ad*ns
replace_pairs = []
half_len = int(len(str(binary)) / 2)
```

```
for len_i in range(half_len + 1, 2, -1):
    replace_pairs.append(('da' * len_i, 'a' + 'd' * len_i + 's'))

operations_str = ''.join(standard_operations)
for replace_pair in replace_pairs:
    dan_form, adns_form = replace_pair
    operations_str = operations_str.replace(dan_form, adns_form)

replace_operations = list(operations_str)
check(d, replace_operations)
print(d, 'G =', binary, 'G\n')
print('double %d times.' % (replace_operations.count('d')))
print('add %d times.' % (replace_operations.count('a')))
print('subtract %d times.' % (replace_operations.count('s')))
print('total %d times.' % (len(replace_operations)))
print('\ndatailed operations steps:', list(map(expansion, replace_operations)))
reconstruct(replace_operations)
```

```
-------------------- standard algorithm --------------------

4024 G = 111110111000 G

double 11 times.
add 7 times.
total 18 times.

datailed operations steps: ['double', 'add', 'double', 'add', 'double', 'add', 'double',
'add', 'double', 'double', 'add', 'double', 'add', 'double', 'add', 'double', 'double', '
double']

initial   1
double    10
add       11
double    110
add       111
double    1110
add       1111
double    11110
add       11111
double    111110
double    1111100
add       1111101
double    11111010
add       11111011
double    111110110
add       111110111
double    1111101110
double    11111011100
double    111110111000

-------------------- optimized algorithm --------------------

4024 G = 111110111000 G

double 11 times.
add 2 times.
subtract 2 times.
total 15 times.

datailed operations steps: ['add', 'double', 'double', 'double', 'double', 'subtract', 'd
ouble', 'add', 'double', 'double', 'double', 'subtract', 'double', 'double', 'double']

initial   1
add       10
double    100
double    1000
double    10000
double    100000
subtract  11111
double    111110
add       111111
```

```
double     1111110
double     11111100
double     111111000
subtract   111110111
double     1111101110
double     11111011100
double     111110111000
```

6. Take a Bitcoin transaction as you wish. Sign the transaction with a random number k and your private key d.



In [11]:

```
import hashlib
import pycoin.ecdsa.secp256k1 as secp256k1
import random

def signing():
    print('-' * 20, 'ECDSA Signing', '-' * 20 + '\n')
    G = secp256k1.secp256k1_generator # base point
    dA = student_ID # private key
    n = G.order()  # group order
    QA = dA * G # public key curve point

    # 1. Hash message
    message = '4024'
    hash_func = hashlib.sha256()
    hash_func.update(message.encode('utf-8'))
    msg_hashed = hash_func.hexdigest()

    # 2. transaction after hash
    z = 0xe087c96184f4bbc0f5e5dd19bce32595bb93859e204085ce57cbdaf6cad96e21

    r = 0
```

```
    while r == 0:
        # 3. Select random integer k from [1, n - 1]
        k = random.randint(1, n-1) # The ephemeral key select from cryptographically sec
ure random.

        # 4. calculate the curve point (x1, y1) = k * G
        x1, y1 = k * G

        # 5. calculate r = x1 mod n, k and n_order should be co-prime, otherwise no modin
v exists.
        r = x1 % n

    # 6, calculate s = k ^ -1 * (z + r * dA) mod n
    k_inv = G.inverse_mod(k, n)
    s = k_inv * (z + r * dA) % n

    print('r = %s \ns = %s' % (hex(r), hex(s)))
    return G, n, r, s, z, QA
```

In [12]:

```
G, group_order, r, s, msg_hashed, public_key = signing()
```

```
-------------------- ECDSA Signing --------------------

r = 0x6ac0b147b5abc0786b1505777bee2b3e49b8f74a7f6cdfb20176440c1514e9de
s = 0xce109c2aebab90ca88df7a6c649d500aa971a975b5f6b3585e0c1b7fe5dce324
```

7. Verify the digital signature with your public key Q.

In [13]:

```
def verifying(G, n, r, s, z, QA):
    print('-' * 20, 'ECDSA Verifying', '-' * 20 + '\n')
    print('Q:', QA, '\n')
    if r < 1 or r > n:
        print('Invalid signature!')
        return
    elif s < 1 or s > n:
        print('Invalid signature!')
        return

    # calculate w = s ^ -1 mod n
    w = G.inverse_mod(s, n)
    u1 = (z * w) % n
    u2 = (r * w) % n
    x1, y1 = u1 * G + u2 * QA

    print('r  mod n =', hex(r % n))
    print('x1 mod n =', hex(x1 % n))
    if r % n == x1:
        print('\nSignature verified successfully')
```

In [14]:

```
verifying(G, group_order, r, s, msg_hashed, public_key)
```

```
-------------------- ECDSA Verifying --------------------

Q: (61952120602367647469610853420592469181846729788270495274890958861001646950 20, 1106505
64425778572861829106954624974923991684809505813926638099517454549477911)

r  mod n = 0x6ac0b147b5abc0786b1505777bee2b3e49b8f74a7f6cdfb20176440c1514e9de
x1 mod n = 0x6ac0b147b5abc0786b1505777bee2b3e49b8f74a7f6cdfb20176440c1514e9de

Signature verified successfully
```

8. Over Z_{10007}, construct the quadratic polynomial p(x) with p(1) = 10,

p(2) = 20, and p(3) = d

$$d = 4024$$

$$p(x) = (10 * \frac{(x-2)(x-3)}{(1-2)(1-3)} + 20 * \frac{(x-1)(x-3)}{(2-1)(2-3)} + 4024 * \frac{(x-1)(x-2)}{(3-1)(3-2)}) \mod 10007$$

In [15]:

```
def quadratic_polynomial(x):
    d = student_ID
    value = int(10 * ((x - 2) * (x - 3)) / ((1 - 2) * (1 - 3)) + 20 * ((x - 1) * (x - 3
)) / ((2 - 1) * (2 - 3)) + d * ((x - 1) * (x - 2)) / ((3 - 1) * (3 - 2))) % 10007
    return value

def log_values(xs):
    for x in xs:
        print(f'p({x}) = {quadratic_polynomial(x)}')
```

In [16]:

```
log_values([1, 2, 3])
```

```
p(1) = 10
p(2) = 20
p(3) = 4024
```