Blockchain Sturcture

Ch 2 : Cryptography

암호화폐 동작 암호기술 : 공개키, 해시, 전자서명

암호의 역사

고대

문자 위치 바꾸는 전치 암호(transposition cipher) : 원통 암호
다른 문자로 바꾸는 치환암호(substitution cipher) : shift2 암호

근대

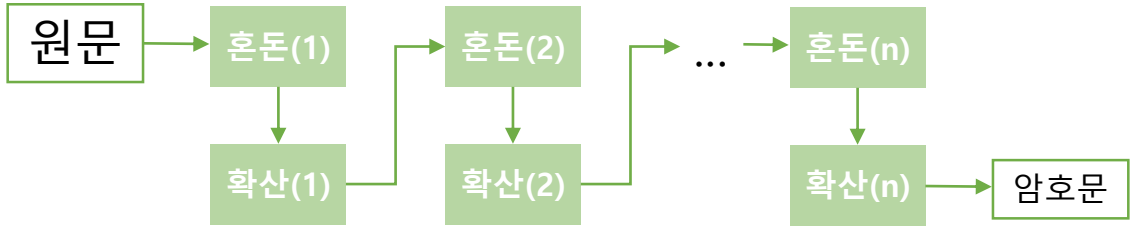
Enigma – 로터를 통해 문자를 치환
암호를 만들거나 풀 때 키가 필요하다

둘 다 암호문을 만들거나 풀 때 키가 사용된다. 푸는 난이도만 달라짐
보낸 사람과 받는 사람이 동일한 키를 갖고 있어야 한다 (비밀키 = secret key(symmetric key))
암호문 자체는 공개적으로 보내도 된다. 단 키는 매우 중요

따라서 현대 암호는 비대칭키(asymmetric key) 혹은 **공개키(public key) 방식**을 사용
이제 키를 전달할 필요가 없다. BTC 네트워크도 암호 방식 사용

암호의 요건

클로드 섀넌 : 안전한 암호문을 위해서는 **혼돈과 확산 과정의 반복**이 필요하다.
현대 암호는 혼돈과 확산의 반복(rounding). 근대 암호인 Enigma는 혼돈만 거쳐서 안전하지 않았다

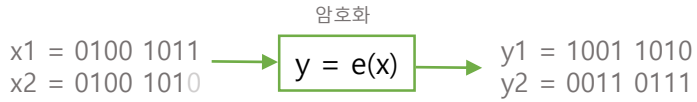


혼돈(Confusion) 과정

원문과 암호문의 관계가 모호해야한다. 상관 관계가 존재하면 비밀키나 원문이 추정될 수 있다

확산(Diffusion) 과정

원문의 특정 위치에 있는 내용은 암호문의 여러 위치에 영향을 미쳐야 한다.
특정 위치에만 영향을 미친다면 원문과 암호문 간의 관계를 추론할 수 있다.

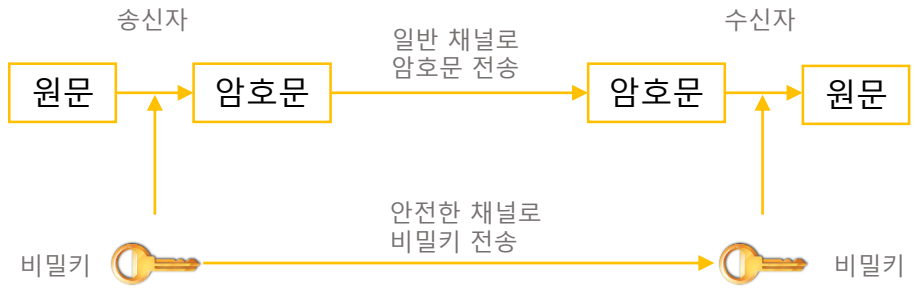


원문의 특정 지점이 암호의 여러 점과 대응되야한다.
예) 1 비트만 바뀌여도 결과값이 완전히 달라지는 것

대칭키 기반 암호기술

암호문 만들거나 풀 때 동일한 키 사용 기술
예) DES, AES (주로 사용)

동작 방식



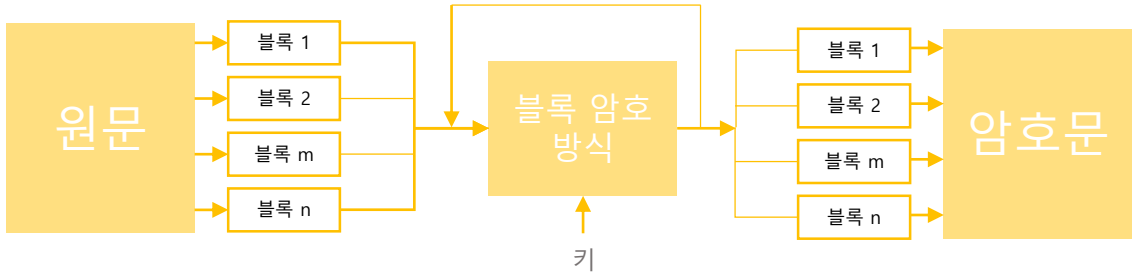
비밀키가 노출되면 암호문이 풀리므로 비밀키는 자주 바꿔야 한다. 일회용 비밀키를 사용하기도 함

블록 암호 방식(block cipher) : 문서 암호화 시 일반적으로 문서를 여러 개의 블록(block)으로 나눈 후 키를 적용
스트림 암호 방식(stream cipher) : 원문이 실시간 데이터인 경우, 휴대폰과 기지국 사이 음성 통화 암호화

대칭키 기반 암호기술

블록 암호 방식 (Block Cipher)

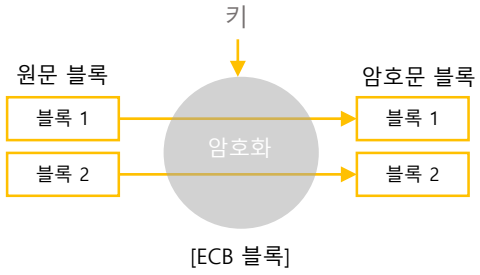
원문을 여러 블록으로 나눠서 블록 별로 암호화 후, 암호화된 블록을 다시 합친다.



원문과 암호문이 1:1일수도 있고, 서로 섞일 수도 있다.

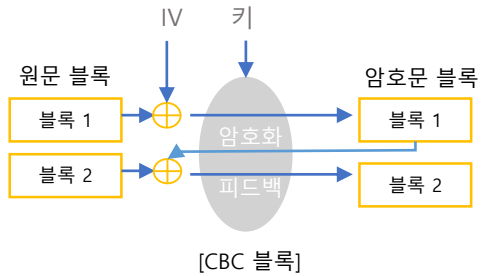
Electronic Code book (ECB) : 일대일 대응. 블록마다 키를 적용해 암호화.

Cipher Block Chain (CBC) : 일대일 아님. 암호문 블록이 피드백되어 다음 원문 블록에 반영된다. 초기는 피드백 없어 초기 백터 Initial Vector(IV) 사용한다. 1내용이 이후 내용도 영향. IV는 키와 함께 보내져야한다.



[ECB 블록]

ECB는 병렬 처리가 가능하고,
누락돼도 정상적으로 받은 것은 풀수 있다.
확산 과정이 없어서 안전하지 못하다



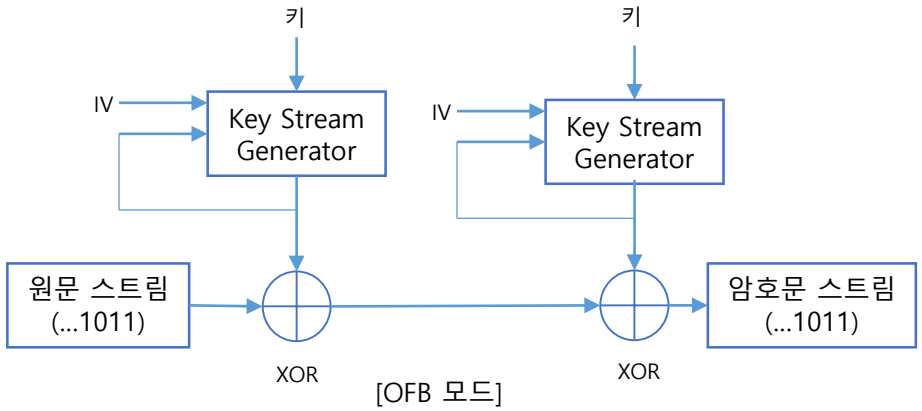
[CBC 블록]

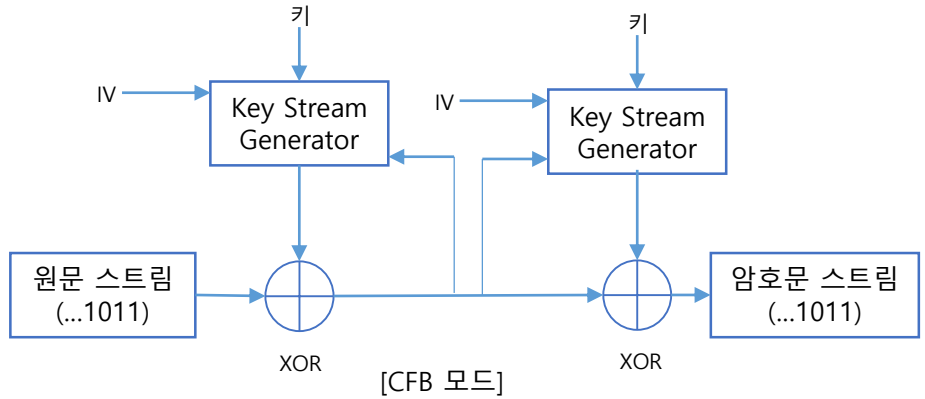
병렬처리는 어렵지만
확산과정이 있어서 안전하다

대칭키 기반 암호기술

스트림 암호 방식 (Stream Cipher)

원문 스트림을 비트 단위로 암호화. 초기 키는 IV와 섞임
Output Feedback(OFB) 모드 : 이전 키가 다음 키로 피드백된다
Cipher Feedback(CFB) 모드 : 암호 비트가 다음 키에 피드백된다





Data Encryption Standard(DES) 알고리즘

블록 암호 방식으로 원문을 64비트 블록으로 나누고 각 블록에 56비트의 키를 적용한다.
DES는 키의 크기가 작다는 등 취약점 존재. (1998년 풀 수 있는 장비 등장 – 보완된 표준 제정)

64비트 원문 블록을 왼쪽 32bit만 서브키로 암호화한 후, 왼쪽과 오른쪽을 교차한다 = 파이스텔 네트워크
이를 16번 반복한다. 암호화와 복호화 과정이 같다. 다만 취약해서 3번 실행하는 3DES 방법이 사용되기도 한다.

Advanced Encryption Standard(AES) 알고리즘

DES 보완 알고리즘으로 채택. 치환-전치 구조를 지님 (Substitution-Permutation Network, SPN)
Substitution은 혼돈(Confusion) 담당, Permutation은 확산(Diffusion) 기능
AES는 128비트 키가 사용되고 치환-확산을 총 10번 반복한다

BTC 코어 프로그램에서 지갑 암호화할 때 256비트 AES 알고리즘(CBC모드)를 사용한다
사용자가 비밀번호 설정 시 이것을 대칭키로 지갑의 키들을 암호화한다

[실습] 대칭키 암호 절차(AES)

AES 알고리즘을 사용해 원문을 암호문으로 변환하고 동일한 키를 사용해서 암호문을 다시 원문으로 변환한다.
블록 암호 방식은 CBC를 사용하는데, 원문과 암호문 블록이 일대일 대응이 아니다.

```
# Advanced Encryption Standard(AES) 알고리즘 연습
# CBC(Cipher Block Chain) 모드로 암호화한다.
from Crypto.Cipher import AES
from Crypto import Random
import numpy as np

# 대칭키를 만든다. 대칭키는 128-bit, 192-bit, 256-bit를 사용할 수 있다.
secretKey128 = b'0123456701234567'
secretKey192 = b'012345670123456701234567'
secretKey256 = b'0123456701234567012345670123456701234567'

# 128-bit key를 사용한다.
secretKey = secretkey128
plainText = 'This is Plain text. It will be encrypted using AES with CBC
mode.'
print("₩n₩n")
print("원문 :")
print(plainText)

# CBC 모드에서는 plain text가 128-bit(16byte)의 배수가 돼야 하므로
padding이 필요함
# padding은 NULL 문자를 삽입. 수신자는 별도로 padding을 제거할 필요는
없음.
n = len(plainText)
if (n % 16) != 0:
    n = n + 16 - (n % 16)
    plainText = plainText.ljust(n, '₩0'))

# initialization vector. iv도 수신자에게 보내야 한다.
iv = Random.new().read(AES.block_size)
ivcopy = np.copy(iv) # 수신자에게 보낼 복사본

# 송신자는 secretKey와 iv로 plainText를 암호문으로 변환한다.
iv = Random.new().read(AES.block_size)
ivcopy = np.copy(iv)
aes = AES.new(secretKey, AES.MODE_CBC, iv)
cipherText = aes.encrypt(plainText)
print('₩n₩n₩n')
print('암호문 :')
print(cipherText.hex())
```

```
# 암호문, secretKey, ivcopy를 수신자에게 보내면 수신자는 암호문을 해독할 수 있다.
aes = AES.new(secretKey, AES.MODE_CBC, ivcopy)
plainText2 = aes.decrypt(cipherText)
plainText2 = plainText2.decode()
print('₩n₩n₩n')
print('해독문 :')
print(plainText2)
```

실행 결과

(아직)

대칭키 기반 암호 방식의 문제점

해결되야 할 문제점

- 키를 보내기 위해 현실적으로 완전히 안전한 채널이 존재하는가?
- 송신자가 불특정 다수에게 암호문을 보낼 때 비밀키는 어떻게 전달해야 하나?
- 수신자가 암호문을 받았을 때 이 문서가 그 송신자가 보낸 것이 맞는지 어떻게 확인할 수 있을까? **인증 문제**(Authentication problem)
- 송신자가 암호문을 보내놓고 보낸 적이 없다고 주장하면 어떻게 이를 증명할 수 있을까? **부인 방지 문제**(Non-repudiation)

공개키 기반 암호 방식을 사용하면 해결된다

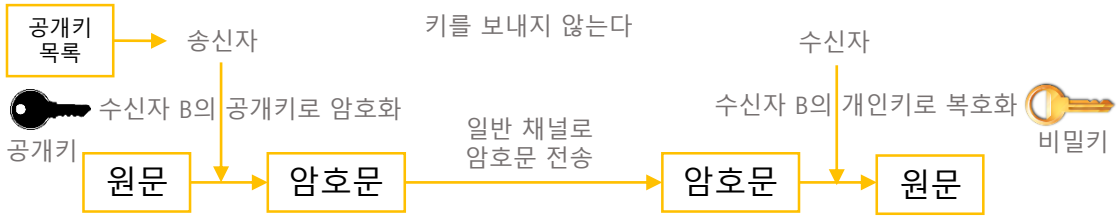
BTC도 인증 문제 해결 위해 공개키 기반 기술 사용

이를 통해 전자 서명으로 소유주를 증명한다

공개키 기반 암호기술

개인키와 공개키를 이용한다. 이 들은 특별한 관계를 맺을 쌍이다.
키를 전달할 필요 없이 대칭키 암호 방식의 문제점들(인증문제, 부인 방지 문제)을 해결할 수 있다

동작 방식



키가 별도로 넘어가지 않고 이미 나와 있는 공개키를 사용하는 것이 특징이다

공개키로 암호화한 문서는 개인키로 풀 수 있지만, 개인키로 암호화한 문서는 공개키로 풀 수 없다.

현재 나온 기술 중 가장 혁신적 : 공인 인증서, 전자상거래, 웹 보안, 블록체인에도 핵심적인 기술로 사용된다.

공개키 기반 암호기술

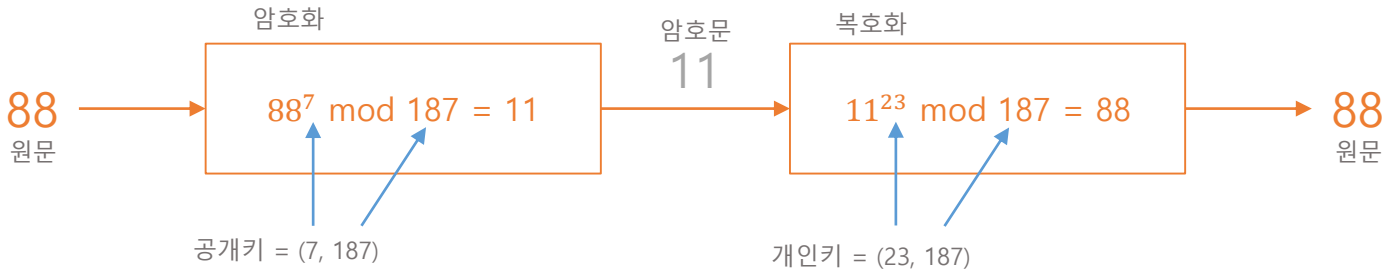
RSA 알고리즘

개인키와 공개키는 수학적 근거로 생성된다 : 정수론, 유한체, 타원곡선군에 대한 이산 대수 등
어떤 수를 두 개의 큰 소수로 분해하는 것은 어렵다는 것을 이용한다

공개키 쌍 (7, 187) // 개인키 쌍 (23, 187)
예시) 원문 88을 공개키로 암호화 해 암호문인 11을 만든 후, 다시 개인키를 적용해 복호화하는 과정

1. 두 개의 (큰) 소수를 선택한다. p = 17, q = 11(실제에서는 큰 소수를 사용한다 – 클 수록 안정성 상승)
2. n = p · q = 17 * 11 = 187
3. Φ(n) = (p-1) * (q-1) = 16 * 10 = 160
4. Φ(n) = 160과 서로소 관계에 있으며 Φ(n) 이하인 정수 한 개(e)를 선택한다. e=7
5. d · e ≡1 mod 160, and d<160를 만족하는 d를 찾는다. (by EEA) d=23→23*7=161 ≡ 1 mod 160
6. Public key = (e, n) = (7, 187), Private key = (d, n) = (23,187)

오일러 파이 함수 → n까지 양의 정수 중 n과 서로소인 정수들의 개수
(예시를 들어보면 신기하게 일치함을 볼 수 있다)



[실습] RSA 알고리즘

코드

```
# Public Key(RSA) 알고리즘 연습
from Crypto.PublicKey import RSA

# Private key와 Public key 쌍을 생성한다
# Private key는 소유자가 보관하고, Public key는 공개한다
keyPair = RSA.generate(2048)
privKey = keyPair.exportKey() # 키 소유자 보관용
pubKey = keyPair.publicKey() # 외부 공개용

# keyPair의 p, q, e, d를 확인해 본다
keyObj = RSA.importKey(privKey)
print("p = ", keyObj.p)
print("q = ", keyObj.q)
print("e = ", keyObj.e)
print("d = ", keyObj.d)

# 암호화할 원문
plainText = "This is Plain text. It will be encrypted using RSA."
print()
print("원문 : ")
print(plainText)

# 공개키로 원문을 암호화한다.
cipherText = pubKey.encrypt(plainText.encode(), 10)
print("₩n")
print("암호문 : ")
print(cipherText[0].hex())

# Private key를 소유한 수신자는 자신의 Private key로 암호문을 해독한다.
# pubKey와 쌍을 이루는 privKey만이 이 암호문을 해독할 수 있다.
key = RSA.importKey(privKey)
plainText2 = key.decrypt(cipherText)
plainText2 = plainText2.decode()
print("₩n")
print("해독문 : ")
print(plainText2)
```

실행 결과

Diffie-Hellman Key 교환 알고리즘(DHKE)

개인키와 공개키를 이용한다. 이 들은 특별한 관계를 맺을 쌍이다.
키를 전달할 필요 없이 대칭키 암호 방식의 문제점들(인증문제, 부인 방지 문제)을 해결할 수 있다

동작 방식

송신자	Alice	수신자	Bob
개인키	a ∈ { 2, 3, ..., p-2}	개인키	b ∈ { 2, 3, ..., p-2}
공개키	A = G^a mod p	공개키	B = G^b mod p
공통키	K_{ab} = B^a mod p	공통키	K_{ab} = A^b mod p

각자의 개인키와 공개키를 이용해 공통의 키를 만드는 알고리즘

사전의 동의된 정보 (G, p)를 이용해 각자의 개인키와 공개키, 그리고 공통키를 만든다

G, p는 표준 문서의 정의된 큰 값의 소수 // 각자의 개인키 a와 b → 이를 통해 공개키 A와 B를 생성한다

G, p를 알고 개인키가 있으므로 공개키 A,B를 계산할 수 있다

공개키 A, B는 서로에게 알려주므로 공식을 통해 공통키 K_{ab}를 만들 수 있다

송신자는 K_{ab}로 암호문을 만들고 수신자에게 보낸다. 수신자도 알고 있으므로 암호문을 볼 수 있다

키를 전달하지 않고도 키가 교환되었다

+) Elgamal 알고리즘

DHKE와 유사하지만 암호문을 보낼 때마다 키를 바꿀 수 있어 더욱 안전한 방식이다. 임시키(ephemeral key)와 마스킹 키 (masking key)를 만들어 마스킹키로 문서를 암호화하는 방식.

마스킹키가 계속 바뀌기 때문에 동일한 문서라도 암호문이 바뀐다.

Square-and-Multiply 알고리즘 (공개키 계산)

DHKE에서 대단히 큰 수를 이용해 공개키 생성시, Square-and-Multiply 알고리즘을 이용하면 쉽게 계산할 수 있다
공개키 계산시 곱셈을 단순히 반복한다면 시간이 오래 걸릴 것이다 (몇 백년)

동작 방식 (예제)

$$3^{324}\ mod\ 15 = ?$$

$$234_{(10)} = 11101010_{(2)}$$

실제에서는 256비트의 큰 수를 사용
mod는 나머지 값을 의미한다

$bit\ 1\ (1):3\ mod\ 15$ (초기화)

$bit\ 2\ (1):\ 3^2\ mod\ 15 = 9\ \mathrm{mod}\ 15\ (Square)$
$9\ *3\ mod\ 15 = 12\ \mathrm{mod}\ 15\ (Multiply)$

$bit\ 3\ (1):\ 12^2\ mod\ 15 = 9\ \mathrm{mod}\ 15\ (Square)$
$9\ *3\ mod\ 15 = 12\ \mathrm{mod}\ 15\ (Multiply)$

$bit\ 4\ (0):\ 12^2\ mod\ 15 = 9\ \mathrm{mod}\ 15\ (Square)$

$bit\ 5\ (1):\ 9^2\ mod\ 15 = 6\ \mathrm{mod}\ 15\ (Square)$
$6\ *3\ mod\ 15 = 3\ \mathrm{mod}\ 15\ (Multiply)$

$bit\ 6\ (0):\ 3^2\ mod\ 15 = 9\ \mathrm{mod}\ 15\ (Square)$

$bit\ 7\ (1):\ 9^2\ mod\ 15 = 6\ \mathrm{mod}\ 15\ (Square)$
$6\ *3\ mod\ 15 = 3\ \mathrm{mod}\ 15\ (Multiply)$

$bit\ 8\ (0):\ 3^2\ mod\ 15 = 9\ \mathrm{mod}\ 15\ (Square)$

1. 승수인 234를 이진수로 변환한다
2. 왼쪽 비트부터 오른쪽으로 가면서 Square-and-Multiply 알고리즘을 적용한다
3. 첫 번째 비트는 베이스인 3으로 초기화한다
4. 두 번째 비트부터 Square-and-Multiply를 적용한다
5. 비트가 '1'이면 Square(이전 결과 제곱 mod 15)를 계산한 후 Multiply(이전결과 * 3 mod 15)를 적용한다
6. 비트가 '0'이면 Square만 적용한다.
7. 마지막 비트까지 진행하면 정답은 9다

파이썬 pow() 함수도 실제로 이를 통해 계산하기 때문에 매우 빠르다

$1234567890^{288493847364595948272634}\ mod\ 28374264893 = ?$

pow(1234567890, 288493847364595948272634, 28374264893) = 25380403324

$$3^{324}\ mod\ 15 = 9$$

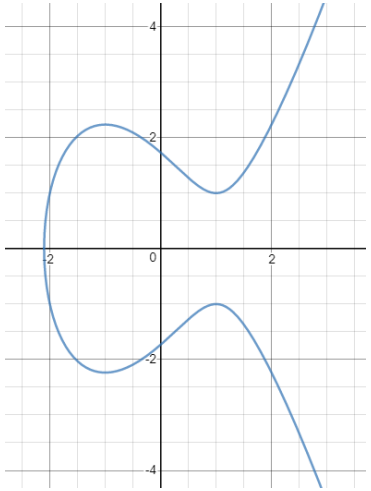
G = 3, p = 15, 개인키 234, 공개키 9
개인키를 알고 있으면 쉽게 계산 가능
공개키만 알 경우 3번을 여러 번 곱하며 9가 나올때까지 반복해야한다
실제는 대단히 큰 수여서 개인키를 현실적으로 알아내는게 불가능하다

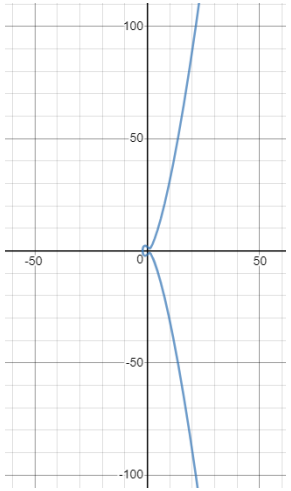
타원곡선암호 알고리즘(Eliptic Curve Crytography : ECC)

작은 수로 키를 만들어도 RSA만큼 성능을 지닌 매우 우수한 암호기술 (160 ~256 bit ECC ≒ 1024 ~ 3072 bit RSA 와 성능 유사)

계산속도 빠르고 안전성 높음 → 비트코인 지갑이 개인키 & 공개키 만들 시 사용

동작 방식





$y^2 = x^3 - 3x + 3$

ECC는 $y^2 = x^3 + ax + b$ 함수식 사용한다 *(예시 : $a = -3, b = 3$)*

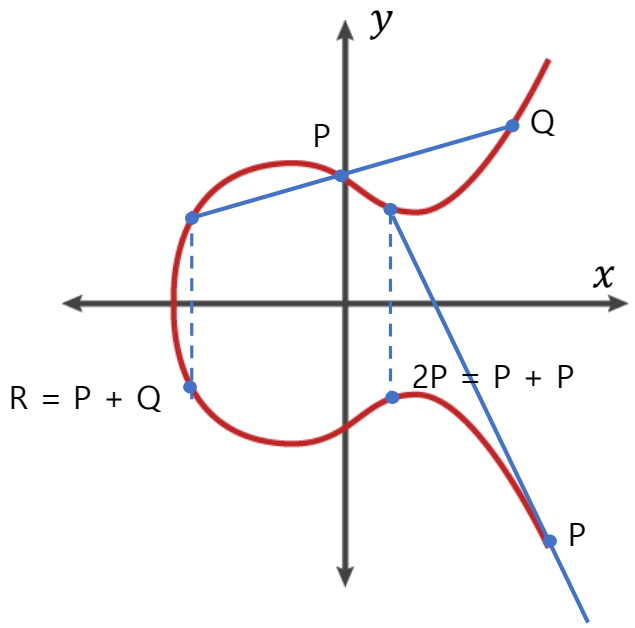
이 그림은 실수 영역에서 그린 것이고
실제 ECC는 $y^2 = x^3 + ax + b$ (mod p) 로 유한체 (x, y 모두 자연수) 공간산의 점들로 구성

ECC는 타원곡선상 점들을 연산하는 규칙을 정의하고 (덧셈규칙), 특정 위치 점을 여러 번 더해 개인키와 공개키 생성

다른 알고리즘과 마찬가지로 개인키 → 공캐키 생성은 쉽지만 그 반대는 어렵다

타원곡선암호 알고리즘(Eliptic Curve Crytography : ECC)

덧셈 연산자(Addition operator)



+) 실제 계산 수식

P = (x_1, y_1), Q = (x_2, y_2), R or 2P = (x_3, y_3)

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

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

$$s = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} \bmod p \;\; ; \text{if } P \neq Q \; (point\ addition) \\ \frac{3x_1^2 + a}{2y_1} mod \; p \;\; ; if \; P = Q \; (point\ doubling) \end{cases}$$

타원 곡선상 서로 다른 두 점 P와 Q의 덧셈은 P와 Q를 잇는 선을 연장해 타원곡선과 만나는 점을 찾고,

그 점이 x축과 대칭인 점 R로 정의한다 (R = P + Q) → 이를 덧셈 연산자(addition)라 한다

한 점에서 계산하는 것은 2배를 하며, 접선과 타원곡선 만나는 점에 대칭한다 → doubling 연산자라 한다

기하학적으로 점 R과 2P를 구하기 위해서는 각 선의 기울기를 계산하고, 직선과 타원곡선의 교점을 구하면 된다

타원곡선암호 알고리즘(Eliptic Curve Crytography : ECC)

덧셈 연산자 연습

공개키를 만들기 위해 게산하자

EC : $y^2 = x^3 + 2x + 2 \bmod 17$

P = (5, 1) → 2P = P + P = (5, 1) + (5, 1) = (x_3, y_3)

s = $\frac{3x_1^2+a}{2y_1} mod\ 17 = (2 \cdot 1)^{-1}(3 \cdot 5^2 + 2)\ mod 17 = 2^{-1} \cdot 9 mod\ 17$

$2^{-1}\ mod\ 17 = 2^{17-2}\ mod\ 17 = 2^{15}\ mod\ 17 = 9\ mod\ 17$

s = $9 \cdot 9 mod\ 17 = 13\ mod\ 17$

$x_3 = 13^2 - 5 - 5\ mod\ 17 = 6\ mod\ 17$

$y_3 = 13(5-6) - 1\ mod\ 17 = 3\ mod\ 17$

2P = (6, 3)

페르마 소정리 (Fermat's Little Theorm)

$a^p \equiv a\ mod\ p, a^{p-1} \equiv 1\ mod\ p, a^{-1} \equiv a^{p-2}\ mod\ p$

P = (5, 1) → 2P = (6, 3)

이렇게 나온 결과에서 개인키는 2이고, 공개키는 (6,3) 이다

기울기 s를 구할 때 분모는 나눈다는 의미가 아닌다.

mod p는 나눌 수 없으므로 역원을 구해서 곱해야하는데,

페르마의 소정리가 이용된다.

타원곡선암호 알고리즘(Eliptic Curve Crytography : ECC)

순환군(cycle group)

위 방법으로 EC 위의 모든 점을 찾아보면 총 18개 점이 존재한다 (1P~18P)

19P 생성 과정에서 직선은 타원 곡선과 만나지 않아서 inf로 정의하고, 수학적으로 '0'으로 정의한다

19P = 0 이므로 20P는 다시 P가 된다 → 이 18개 점들을 순환군이라고 한다

P = (5, 1)

2P = P + P = (6, 3)

3P = 2P + P = (10, 6)

4P = 3P + P = (3, 1)

5P = 4P + P = (9, 16)

6P = 5P + P = (16, 13)

7P = 6P + P = (0, 6)

8P = 7P + P = (13, 7)

9P = 8P + P = (7, 6)

10P = 9P + P = (7, 11)

11P = 10P + P = (13, 10)

12P = 11P + P = (0, 11)

13P = 12P + P = (16, 4)

14P = 13P + P = (9, 1)

15P = 14P + P = (3, 16)

16P = 15P + P = (10, 11)

17P = 16P + P = (6, 14)

18P = 17P + P = (5, 16)

19P = 18P + P = inf

20P = 19P + P = (5, 1)

21P = 20P + P = (6, 3)

22P = 21P + P = (10, 6)

23P = 22P + P = (3, 1)

24P = 23P + P = (9, 16)

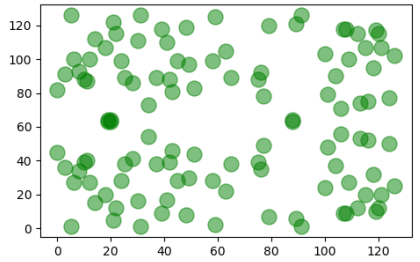
[실습] 타원곡선암호의 순환군 찾기

파이썬을 이용해 타원곡선 $y^2 = x^3 + 2x + 2 \pmod{127}$상 존재하는 모든 점을 찾아보자
덧셈 연산자와 doubling 연산자를 이용한다

코드

```
import math
import numpy as np
import matplotlib.pyplot as plt

# Additive Operation
def addOperation(a,b,p,q,m):
    if q == (math.inf, math.inf):
        return p

    x1 = p[0]
    y1 = p[1]
    x2 = q[0]
    y2 = q[1]

    if p == q:
        # Doubling
        # slope (s) : (3 * x1 ^ 2 + a) / (2 * y1) mod m
        # 분모의 역원부터 계산한다(by Fermat's Little Theorem)
        r = 2 * y1
        rInv = pow(r, m-2, m) # Fermat's Little Theorm
        s = (rInv * (3 * (x1 ** 2) + a)) % m
    else:
        r = x2 - x1
        rInv = pow(r, m-2, m) # Fermat's Little Theorem
        s = (rInv * (y2-y1)) % m
    x3 = (s ** 2 - x1 - x2) % m
    y3 = (s * (x1 - x3) - y1) % m
    return x3, y3

# y^2 = x^3 + 2*x + 2 mod 127
a = 2
b = 2
m = 127 # Prime number여야 한다
P = (5, 1)
Q = P
allPoints = [P]
while(1):
    # R이 P의 inverse인지 확인한다. inverse이면 infinity 지점
임
    # A = (x1, y1), B = (x2, y2)일 때 x1 = x2이고 y1과 y2가
    # (mod m)에 대해 additive inverse이면 infinity
    if (Q[0] == P[0]) & (abs(Q[1] - m) == P[1]):
        # 다음부터 cyclic되므로 여기서 멈춤
        break
    else:
        R = addOperation(a, b, P, Q, m)
        allPoints.append(R)
        Q = R

x, y = np.array(allPoints).T
plt.figure(figsize=(8,6))
plt.scatter(x, y, marker='o', color='green', alpha=0.5, s=150)
plt.show()
print(allPoints)
```

실행 결과 (순환군)




```
[(5, 1), (107, 9), (34, 54), (48, 8), (89, 6), (121, 107), (118, 95), (75,
88), (24, 28), (106, 56), (65, 38), (91, 1), (31, 126), (88, 64), (76, 92)
, (3, 91), (112, 115), (59, 2), (12, 100), (126, 102), (43, 46), (14, 112)
, (119, 117), (21, 5), (109, 100), (8, 34), (108, 9), (0, 82), (39, 118),
(120, 115), (124, 50), (101, 79), (42, 88), (41, 110), (113, 53), (77, 49)
, (116, 52), (100, 24), (104, 90), (49, 30), (10, 39), (58, 28), (11, 87),
(6, 27), (30, 111), (25, 38), (22, 12), (20, 64), (79, 120), (115, 107), (
45, 99), (63, 22), (19, 64), (28, 86), (51, 83), (18, 20), (37, 89), (37,
38), (18, 107), (51, 44), (28, 41), (19, 63), (63, 105), (45, 28), (115, 2
0), (79, 7), (20, 63), (22, 115), (25, 89), (30, 16), (6, 100), (11, 40),
(58, 99), (10, 88), (49, 97), (104, 37), (100, 103), (116, 75), (77, 78),
(113, 74), (41, 17), (42, 39), (101, 48), (124, 77), (120, 12), (39, 9), (
0, 45), (108, 118), (8, 93), (109, 27), (21, 122), (119, 10), (14, 15), (4
3, 81), (126, 25), (12, 27), (59, 125), (112, 12), (3, 36), (76, 35), (88,
63), (31, 1), (91, 126), (65, 89), (106, 71), (24, 99), (75, 39), (118, 32
), (121, 20), (89, 121), (48, 119), (34, 73), (107, 118), (5, 126)]
```

이 곡선상 총 114개의 점이 존재한다. 이산체 타원곡선으로 표현
개인키 하나를 선택하면 대응하는 점 하나를 찾아 공개치로 사용한다
1로 정하면 (5,1) 마지막인 114로 정하면 (5, 126) 이런 식으로 나온다

개인키, 공개키 생성 [미완]

ECC 표준문서에는 타원곡선의 함수식 (a,b,p)과, 곡선상의 한 점인 G(base point or generator), 순환군 점 개수 N으로 정의된다

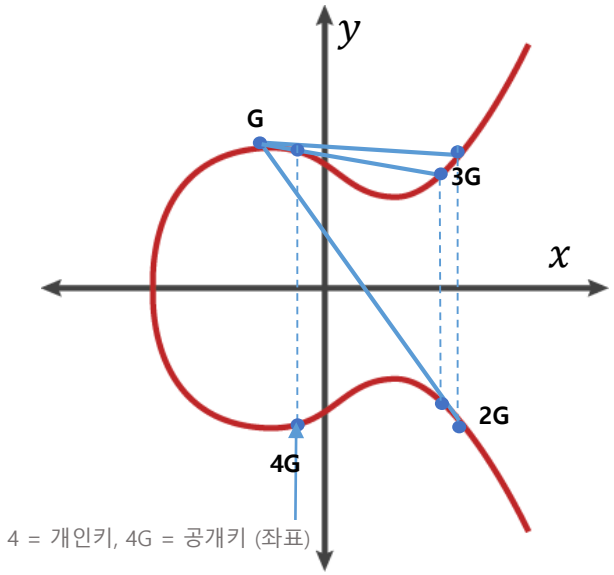
이를 도메인 파라미터라하는데, ECC는 이를 참조해 개인키와 공개키를 만든다

N보다 작은 임의의 키를 선택하면 개인키다(d).

d * G 계산한 값이 공개키다(K). d * G 위에서 정의한 덧셈연산자를 사용해 G를 d번 더한단 의미

현실에서는 매우 큰 값으로 사용해 최신 컴퓨터로도 몇 백년이 걸릴 것이다

square and multiply 알고리즘과 유사한 방식 → Double-and-Add 알고리즘



d(프라이빗 키) · G(Generator) = K (퍼블릭 키)

G + G + ... G(d번) = K

Double-and-Add 알고리즘

G를 26번 더하는 경우를 생각해 보자. Square-and-Multiply 알고리즘처럼 개인키인 26을 이진수로 변환한 다음 왼쪽 비트부터

오른쪽으로 가면서 Double-and-Add 알고리즘을 적용한다. 비트 값이 '1'이면 Double (2배 연산)을 적용한 후 Add(G를 더함)을 적용한다.

비트 값이 '0'이면 double만 적용한다. 이 알고리즘을 사용하면 다음과 같이 6 스텝만에 결과를 얻고

d와 G가 큰 수라도 매우 빠르게 계산할 수 있다

$$26 \cdot G = K \quad 26_{(10)} = 11010_{(2)}$$

bit1 (1) : G (초기화)

bit2 (1) : $G + G = 2G\ (Double)$
$2G + G = 3G\ (Add)$

bit3 (0) : $3G + 3G = 6G\ (Double)$
bit4 (1) : $6G + 6G = 12G\ (Double)$
$12G + G = 13G\ (Add)$
bit5 (0) : $13G + 13G = 26G\ (Double)$

개인키 d를 알고 있으면 위의 알고리즘으로 공개키 K를 빠르게 계산할 수 있지만,

반대로 공개키 K와 G를 알고 있어도 개인키 d를 알아내는 것은 매우 힘들다.

d를 모르기 때문에 Double-and-Add 알고리즘을 사용할 수 없다

작은 수면 brute-force로 노가다 할 수 있지만, 실제 256bit 되는 대단히 큰 수를 사용하므로 추적 불가능하다

[실습] 타원곡선암호의 공개키 생성

개인키를 이용해 Double-and-Add 알고리즘으로 공개키를 생성한다. 도메인 파라미터는 P = 32416189381, a = 2, b = 2, G = (5, 1), 개인키 = 1234567 로 정의한다

코드

```
import math

# Additive Operation
def addOperation(a, b, p, q, m):
    if q == (math.inf, math.inf):
        return p

    x1 = p[0]
    y1 = p[1]
    x2 = q[0]
    y2 = q[1]

    if p == q:
        # Doubling
        # Slope (s) = (3 * x1 ^ 2 + a) / (2 * y1) mod m
        # 분모의 역원으로부터 계산 (by Fermat's Little Theorm)
        # pow() 함수가 내부적으로 Square-and-Multiply 알고리즘 수행
        r = 2 * y1
        rInv = pow(r, m-2, m) # Fermat's Little Theorm
        s = (rInv * (3 * (x1 ** 2) + a)) % m
    else:
        r = x2 - x1
        rInv = pow(r, m-2, m) # Fermat's Little Theorm
        s = (rInv * (y2 - y1)) % m
    x3 = (s**2 - x1 - x2) % m
    y3 = (s * (x1 - x3) - y1) % m
    return x3, y3

# Domain parameter를 정의한다
# y^2 = x^3 + 2x +2 mod 231559
a = 2
b = 2
p =32416189381 # Prime number여야함
G = (5,1)

# 개인키를 선택한다. P보다 작은 임의의 숫자를 선택한다
# 실제로 순환군 범위 내에서 선택한다. 이 부분은 지갑(ch3)서 다룬다
d = 1234567 # Private Key

# Double-and-Add 알고리즘으로 공개키를 생성한다
bits = bin(d)
bits = bits[2:len(bits)]

# Initialize. bits[0] = 1 (always)
K = G

# 두 번째 비트부터 Double-and-Add
bits = bits[1:len(bits)]
for bit in bits:
    # Double
    K = addOperation(a, b, K, K, p)

    # Multiply
    if bit == '1':
        K = addOperation(a, b, K, G, p)

privKey = d
pubKey = K

print("₩nDomain parameters : (P, a, b, G)")
print("P = %d" % p)
print("a = %d" % a)
print("b = %d" % b)
print("G = (%d, %d)" % (G[0], G[1]))
print("EC : y^2 = x^3 + %d * x + %d mod %d" % (a, b, p))
print("₩nKeys : ")
print("Private Key = ", privKey)
print("Public Key = %d * (%d, %d) = (%d, %d)" %(d, G[0], G[1], pubKey[0],
pubKey[1]))
```

실행 결과

```
Domain parameters : (P, a, b, G)
P = 32416189381
a = 2
b = 2
G = (5, 1)
EC : y^2 = x^3 + 2 * x + 2 mod 32416189381

Keys :
Private Key =  1234567
Public Key = 1234567 * (5, 1) = (31130113280, 21384452557)
```

공개키 결과는 K = (31130113280, 21384452557) 로 곡선 산 한 점의 좌표로 나온다

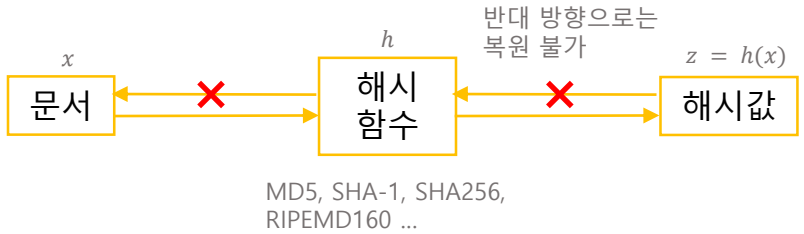
해시(Hash) 알고리즘

해시 : 문서의 내용을 고정된 길이의 어떤 데이터로 변환하는 것
변환된 값을 다이제스트(digest) or 해시 값(hash value)
문서의 일부가 바뀌면 해시 값이 완전히 바뀌므로 문서 조작 여부를 쉽게 확인할 수 있다

암호적 목적 : 데이터 위/변조, 무결성 검출
비암호적 목적 : 파일 다운로드 오류 검출 등

해시함수 → 블록 암호와 유사하게 블록으로 나눠 블록 데이터를 뒤섞거나 XOR 등 연산을 수행한다

해시 값이 같은 경우 해시 충돌이 발생하는데, 숫자가 매우 크므로 발생할 확률이 매우 낮음
안전한 해시함수는 해시 출동확률이 낮다



해시함수의 조건

문서 = x, 해시 함수 = h, 해시 값 = z = h(x)
x는 z의 프리이미지(preimage)다. 해시함수 h는 일대일함수가 아니므로 동일 해시 값 z를 갖는 x가 여러 개 존재가능

1. **프리이미지 저항성(one-wayness)** : z로 x를 찾기 어려워야한다 (단방향적 성질)

2. **제2 프리이미지 저항성 (weak collision resistance)** : x가 주어질 시, z 같은 값을 갖는 x′ 발견이 어려워야 함

3. **충돌 저항성(strong collision resistance)** : 동일한 해시 값 z를 갖는 임의의 두 문서 x1, x2를 찾는 것이 어려워야 함*

* 두 문서 x1, x2를 찾는 공격을 birthday attack이라 한다

[실습] 해시(Hash) 알고리즘

여러 알고리즘을 적용해 문서 해시 값을 게산한다. 일부 변경 시 완전 달라지는거 확인 가능

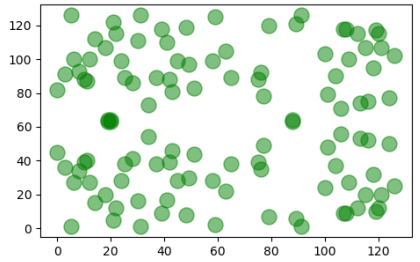
[실습] 타원곡선암호의 순환군 찾기

파이썬을 이용해 타원곡선 $y^2 = x^3 + 2x + 2 \pmod{127}$상 존재하는 모든 점을 찾아보자
덧셈 연산자와 doubling 연산자를 이용한다

코드

```
import math
import numpy as np
import matplotlib.pyplot as plt

# Additive Operation
def addOperation(a,b,p,q,m):
    if q == (math.inf, math.inf):
        return p

    x1 = p[0]
    y1 = p[1]
    x2 = q[0]
    y2 = q[1]

    if p == q:
        # Doubling
        # slope (s) : (3 * x1 ^ 2 + a) / (2 * y1) mod m
        # 분모의 역원부터 계산한다(by Fermat's Little Theorem)
        r = 2 * y1
        rInv = pow(r, m-2, m) # Fermat's Little Theorm
        s = (rInv * (3 * (x1 ** 2) + a)) % m
    else:
        r = x2 - x1
        rInv = pow(r, m-2, m) # Fermat's Little Theorem
        s = (rInv * (y2-y1)) % m
    x3 = (s ** 2 - x1 - x2) % m
    y3 = (s * (x1 - x3) - y1) % m
    return x3, y3

# y^2 = x^3 + 2*x + 2 mod 127
a = 2
b = 2
m = 127 # Prime number여야 한다
P = (5, 1)
Q = P
allPoints = [P]
while(1):
    # R이 P의 inverse인지 확인한다. inverse이면 infinity 지점
임
    # A = (x1, y1), B = (x2, y2)일 때 x1 = x2이고 y1과 y2가
    # (mod m)에 대해 additive inverse이면 infinity
    if (Q[0] == P[0]) & (abs(Q[1] - m) == P[1]):
        # 다음부터 cyclic되므로 여기서 멈춤
        break
    else:
        R = addOperation(a, b, P, Q, m)
        allPoints.append(R)
        Q = R

x, y = np.array(allPoints).T
plt.figure(figsize=(8,6))
plt.scatter(x, y, marker='o', color='green', alpha=0.5, s=150)
plt.show()
print(allPoints)
```

실행 결과 (순환군)




```
[(5, 1), (107, 9), (34, 54), (48, 8), (89, 6), (121, 107), (118, 95), (75,
88), (24, 28), (106, 56), (65, 38), (91, 1), (31, 126), (88, 64), (76, 92)
, (3, 91), (112, 115), (59, 2), (12, 100), (126, 102), (43, 46), (14, 112)
, (119, 117), (21, 5), (109, 100), (8, 34), (108, 9), (0, 82), (39, 118),
(120, 115), (124, 50), (101, 79), (42, 88), (41, 110), (113, 53), (77, 49)
, (116, 52), (100, 24), (104, 90), (49, 30), (10, 39), (58, 28), (11, 87),
(6, 27), (30, 111), (25, 38), (22, 12), (20, 64), (79, 120), (115, 107), (
45, 99), (63, 22), (19, 64), (28, 86), (51, 83), (18, 20), (37, 89), (37,
38), (18, 107), (51, 44), (28, 41), (19, 63), (63, 105), (45, 28), (115, 2
0), (79, 7), (20, 63), (22, 115), (25, 89), (30, 16), (6, 100), (11, 40),
(58, 99), (10, 88), (49, 97), (104, 37), (100, 103), (116, 75), (77, 78),
(113, 74), (41, 17), (42, 39), (101, 48), (124, 77), (120, 12), (39, 9), (
0, 45), (108, 118), (8, 93), (109, 27), (21, 122), (119, 10), (14, 15), (4
3, 81), (126, 25), (12, 27), (59, 125), (112, 12), (3, 36), (76, 35), (88,
63), (31, 1), (91, 126), (65, 89), (106, 71), (24, 99), (75, 39), (118, 32
), (121, 20), (89, 121), (48, 119), (34, 73), (107, 118), (5, 126)]
```

이 곡선상 총 114개의 점이 존재한다. 이산체 타원곡선으로 표현
개인키 하나를 선택하면 대응하는 점 하나를 찾아 공개치로 사용한다
1로 정하면 (5,1) 마지막인 114로 정하면 (5, 126) 이런 식으로 나온다