本解説では、暗号の Python での実装について解説します。具体的には公開鍵暗号として、RSA 暗号と楕円曲線暗号を、共通鍵暗号として、AES 暗号を取り上げます。それぞれの詳細な説明は後述するとして、暗号を実社会に実装する際には、その実装の効率性と安全性を考慮することが重要です。本解説では、スペースの関係で安全な実装についての詳細な議論は行いませんが、現在知られている基本的な実装法について解説します。そこで本章ではまず、暗号の実装の際に重要となる、有限体と有限体上の演算について解説します。

1 群・体について

暗号実装において重要となる概念として、群・体があります。簡単のために厳密な説明を省きますが、イメージをつかんでもらうことを目的として、以下に群・体の説明を行います。

1.1 群

まず群とは、一つの演算が定義された元の集合です。このとき、演算とは、集合の中の 2 つの元を取って、その 2 つの元を結合して、また集合の中の元になるようなものを指します。また、群においては、単位元と呼ばれる、演算においてどの元と結合してもその元自身になるような元が存在し、また、任意の元に対して、逆元と呼ばれる、演算においてその元と結合すると単位元になるような元が存在することが必要です。ある集合が、演算 $\circ$ について群をなすとき、その集合を $< G, \circ >$ と表記します。言葉のみで説明するとわかりにくいので具体例を考えてみます。ただし、$\mathbb{R}, \mathbb{Z}, \mathbb{N}$ はそれぞれ実数、整数、自然数全体の集合を、演算 $+, \cdot$ は通常の加算、乗算を表します。

- $< \mathbb{R}, + >$ は群をなします。単位元は 0 になります。また、任意の実数に対して、その実数の符号を反転したものが逆元になります。
- $< \mathbb{R}, \cdot >$ は群ではありません。なぜなら、単位元は 1 ですが、0 に掛け算を行って 1 になるような元、つまり 0 の逆元が存在しないからです。言い換えれば、$< \mathbb{R} - 0, \cdot >$ は群をなします。
- $< \mathbb{Z}, + >$ は群をなします。単位元は 0 になります。また、任意の整数に対して、その整数の符号を反転したものが逆元になります。
- $< \mathbb{Z}, \cdot >$ は群ではありません。なぜなら、単位元は 1 ですが、整数に掛け算を行って 1 になるような数は整数であらわすことができない、つまり ± 1 以外の元には逆元が存在しないからです。
- $< \mathbb{N}, + >$ は群ではありません。なぜなら、単位元は 0 ですが、自然数に足し算を行って 0 になるような数は負の整数であり、自然数であらわすことができない、つまり 0 以外の元には逆元が存在しないからです。

1.2 体

次に体について説明します。群では一つの集合に対して、一つの演算を定義しましたが、体では二つの演算を定義します。G が $\circ_1$ について群をなし、G^* が $\circ_2$ について群をなすとき、G は演算 $\circ_1, \circ_2$ について体をなし $< G, \circ_1, \circ_2 >$ のようにあらわします。ここで、G^* は G から $< G, \circ_1 >$ の単位元を取り除いたものを指します。これに関しても具体例で考えてみます。

- $< \mathbb{R}, +, \cdot >$ は体となります。$\mathbb{R}^*$ は $\mathbb{R}$ から 0 だけを除いた集合です。

- $< \mathbb{Z}, +, \cdot >$ は体ではありません。なぜなら、$< \mathbb{Z}, + >$ は群をなしますが、$< \mathbb{Z}^*, \cdot >$ は ± 1 以外の元は逆元が存在しないので群をなしません。
- $< \mathbb{N}, +, \cdot >$ は体ではありません。なぜなら、$< \mathbb{N}, + >$ は群をなしません。

2 有限体と有限体上の演算について

ここまで、群や体について説明してきましたが、本解説において利用するのは有限体となります。これは、体のうちその集合に含まれる要素が有限であるものを指します。暗号で使われる有限体は、p を素数として、

$$< \{0, 1, \cdots, p-2, p-1\}, \ + \mod (p), \ \cdot \mod (p) >$$

のように、p 個の要素をもつ体として定義されます。ここでのポイントは、各演算が通常の演算と異なり、通常の演算の後に p で割った余りを取るということです。以降、上記のような有限体を F_p と表記し、各演算の剰余演算については省略して表記することにします。例えば、F_7 における演算を考えてみると以下のようになります。

$3+5=1$	通常の演算では $3+5=8$ となりますが、$8 \mod (7) = 1$ となります。
$3+4=0$	3,4 はそれぞれがもう一方の加法逆元となっていることがわかります。
$3 \cdot 4 = 5$	通常の演算では $3 \cdot 4 = 12$ となりますが、$12 \mod (7) = 5$ となります。
$3 \cdot 5 = 1$	3,5 はそれぞれがもう一方の乗法逆元となっていることがわかります。

では、実装について実際に考えてみましょう。とはいっても、通常の演算の後に剰余演算を行うだけです。一点注意が必要なのが乗法逆元の求め方です。これは、フェルマーの小定理を用いて理解することができます。フェルマーの小定理にによると、非零元 $a \in F_p$ に対し、以下のようになることが知られています。

$$a^{p-1} = 1$$

これを変形してみると、以下のようになり、a の乗法逆元、a^{-1} を求めることができます。

$$\begin{aligned} a^{p-1} &= 1 \\ a \cdot a^{p-2} &= 1 \\ a \cdot a^{-1} &= 1 \end{aligned}$$

よって、$a^{-1} = a^{p-2}$ となります。これば、べき乗剰余演算でも乗法逆元を求められることを意味します。一般には拡張ユークリッドの互除法が使われます。

サンプルコード 1: 有限体上の演算

```
1    #関数の入力は 0 ≤ a, b ≤ p とする。ただし p は素数。
2    def Fp_random(p: int) -> int:
3        return random.randint(0, p - 1) #0 から p − 1 までの整数、すなわち F_p の元をランダム
             に返す
4
5    def Fp_add(a: int, b: int, p: int) -> int:
6        ans = a + b
7        if ans >= p:
```

```
        ans = ans - p #a,b < p ならば、a + b < 2p となるので、p で剰余を取らなくても
            ans>p の時に ans-p とすればよい
    return ans

def Fp_sub(a: int, b: int, p: int) -> int:
    ans = a - b
    if ans < 0:
        ans = ans + p #a,b < p ならば、−p < a − b < p となるので、p で剰余を取らなくても
            ans<0 の時に ans+p とすればよい
    return ans

def Fp_mul(a: int, b: int, p: int) -> int:
    ans = (a * b) % p #通常の乗算の後に p で剰余を取る
    return ans

def Fp_pow(a: int, b: int, p: int) -> int:
    ans = pow(a, b, p) #Python の pow 関数は pow(a, b, p) で a^b mod (p) を計算してくれる
    return ans

def Fp_inv(a: int, p: int) -> int:
    ans = Fp_pow(a, p - 2, p) #フェルマーの小定理より、a^{p-1} = 1 なので、a^{p-2} = a^{-1} と
        なる
    return ans
```

また、上記の関数の使用例を以下に示します。

サンプルコード 2: 有限体上の演算の使用例

```
def sample() -> None:
    # 素数 p を設定
    p = 7
    # 有限体上の元をランダムに作成
    a = Fp_random(p)
    b = Fp_random(p)
    # 有限体上の元の足し算
    c = Fp_add(a, b, p)
    # 有限体上の元の引き算
    d = Fp_sub(a, b, p)
    # 有限体上の元の掛け算
    e = Fp_mul(a, b, p)
    # 有限体上の元の累乗
    f = Fp_pow(a, b, p)
    # 有限体上の元の逆元
    g = Fp_inv(a, p)
    # 逆元が正しく計算できているかを確認
    h = Fp_mul(a, g, p)
    print(h) #1 が表示されれば OK
```