目次

第 1 章

導入

1.1 Coppersmith 法の使い方

本書では、次の CTF の Crypto の問題を解くことを考えます。なお、本書に記載のソースコードは、奥付に記載のサイトで公開しています。実際に動かしてみたい場合はダウンロードしてください。

▼ problem.py

```
 1: from Crypto.Util.number import getStrongPrime, bytes_to_long
 2: from math import gcd
 3:
 4: flag = """
 5: +>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+
 6: |    FLAG{??????????????????????????????}    |
 7: +>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+
 8: """
 9: assert len(flag)==127
10:
11: e = 3
12: while True:
13:   p = getStrongPrime(512)
14:   q = getStrongPrime(512)
15:   if gcd((p-1)*(q-1), e)==1:
16:     break
17: N = p*q
18: d = pow(e, -1, (p-1)*(q-1))
19:
20: m = bytes_to_long(flag.encode())
21: c = pow(m, e, N)
22: assert pow(c, d, N)==m
23:
24: print(f"{N=}")
25: print(f"{e=}")
26: print(f"{c=}")
```

▼ output.txt

```
1: N=13589625288945517930604456185607365717658928206506048598204988402415993544045164826880908327809318637896814111510717502395535228574193151069551356963186558798078011685056294628964750454161622954446021661642391607390251635263422198432947237233789640634198054326390013926519600423293656003195630107003987638201
2: e=3
3: c=10406441788570997475935119621977304834837645531383282920994734644687468303167709926366066046574332126567510144720528636700553658137042970793458863959654236500667586711512107296055101997862842550090932118275034317851780728752907036700520613236218741418917049887641671787198614223984425049602231764312451652 1
```

典型的な RSA の問題で、外側の部分（"\n+>-<+>-..."）をビッグエンディアンで整数にしたものを m_0、求めたいフラグ（"???..."）を整数にしたものを x とすると、

$$\left(m_0 + 256^{49}x\right)^e \equiv c \mod N$$

を x について解く問題になります。係数 256^{49} は、フラグが末尾から 49 文字目にあるからです。

求めたい x の値は N に比べて小さいです。もし $m_0 = 0$ ならば、$\left(m_0 + 256^{49}x\right)^e$ は N が超えないので、この問題を解くのは簡単です。単に、mod N を無視して、3 乗根を求めれば良いです。実際は、m_0 によって $\left(m_0 + 256^{49}x\right)^e$ が N より大きくなるので、そう簡単にはいきません。

Coppersmith 法はこのような状況でも答えを求めてくれます。SageMath[*1]では、PolynomialRing の small_roots メソッドとして実装されています。

▼ solve.sage

```
1: N = 13589625288945517930604456185607365717658928206506048598204988402415993544045164826880908327809318637896814111510717502395535228574193151069551356963186558798078011685056294628964750454161622954446021661642391607390251635263422198432947237233789640634198054326390013926519600423293656003195630107003988763820 1
2: e = 3
3: c = 10406441788570997475935119621977304834837645531383282920994734644687468303167709926366066046574332126567510144720528636700553658137042970793458863959654236500667586711512107296055101997862842550090932118275034317851780728752965207036700520613236218741418917049887641671787198614223984425049602231764312451652 1
4:
5: m0 = """
6: +>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+
7: |    FLAG{""" + "\0"*25 +       """}    |
```

[*1] https://www.sagemath.org/

```
 8: +>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+
 9: """
10: m0 = int.from_bytes(m0.encode(), "big")
11:
12: PR.<x> = PolynomialRing(Zmod(N))
13: f = (m0 + 256^49*x)^e - c
14: f /= (256^49)^e
15:
16: flag = f.small_roots()[0]
17: flag = int(flag).to_bytes(25, "big").decode()
18: print(flag)
```

方程式を $f(x) \equiv 0$ の形にするために、c を移項しています。また、Coppersmith 法では最高次（この場合は x^e）の係数が 1 でなければいけないので、まず全体を x^e の係数で割っています。

SageMath の実行には Docker Hub のイメージを使うのが簡単でしょう。Docker がインストールされた Windows なら、ソースコードをカレントディレクトリに置いて、次のようにして実行できます。Linux ならば、`%CD%` を `$PWD` にします。

```
>docker run --rm -it -v "%CD%:/host" sagemath/sagemath sage /host/solve.sage
C0pPersm17h_c@n_br3ak_R5A
```

この Coppersmith 法がどのようなアルゴリズムなのかを解説して実装し、SageMath などを使わずにこの問題を解こうというのが本書のテーマです。

1.2 本書の構成

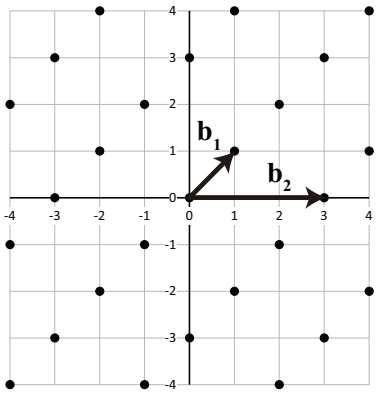
Coppersmith 法は「格子」と「LLL 基底簡約」を使ったアルゴリズムです。第 2 章で格子と LLL 基底簡約を解説し、第 3 章で Coppersmith 法のアルゴリズムを紹介します。

1.3 参考文献

1. Coppersmith, Don. "Finding a small root of a univariate modular equation." *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 1996.
2. 青野 良範, 安田 雅哉. 格子暗号解読のための数学的基礎. 近代科学社, 2019.

本書は参考文献 [1] の内容を解説するものです。格子と LLL 基底簡約については参考文献 [2] を元にしています。これら以外の参考文献については、適宜脚注に出典を記載します。

この格子に含まれる格子点です。一方、$(2,\ 1) = 1 \times \mathbf{b}_1 + \frac{1}{3} \times \mathbf{b}_2$ は、係数が整数ではなく、整数の係数で表わす他の方法も無いので、この格子には含まれません。



▲図 2.1　格子の例

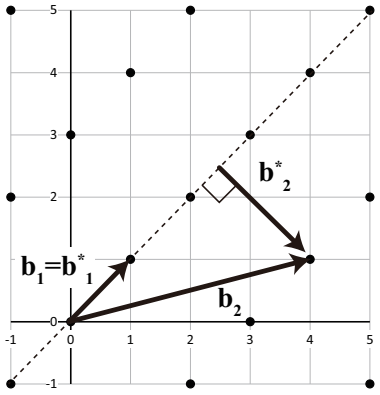
同じ格子を張る基底は複数存在します。例えば、$\mathbf{b}'_1 = (1,\ 1)$ と $\mathbf{b}'_2 = (2,\ -1)$ も図 2.1 と同じ格子を張ります。ある基底ベクトルの整数倍を他の基底ベクトルに加えたベクトルの組は同じ格子を張ります。$\mathbf{b}'_1 = \mathbf{b}_1, \mathbf{b}'_2 = \mathbf{b}_2 - \mathbf{b}_1$ です。

問題に応じて基底を構成し、次章で解説する LLL 基底簡約などの手法を用いて同じ格子を張る別の基底を探し、その別の基底から求める答えを導くというのが、格子を用いて Crypto の問題を解く基本的な流れです。

基底ベクトルの個数 n と各ベクトルの要素数 m は等しい必要はありません。$n < m$ の場合は、後述の LLL 基底簡約をそのまま適用することができます。$n > m$ となる場合、基底ベクトルは必ず一次従属（あるベクトルを他のベクトルの実数倍の和で表わすことができる）になります。$n > m$ かどうかに関わらず、基底ベクトルが一次従属のとき、LLL 基底簡約では一手間が増えます。本書では $n = m$ で基底ベクトルが一次独立である（一次従属ではない）場合のみを扱います。

$n = m$ で基底ベクトルが一次独立な格子について、空間全体を、各部分空間がちょうど 1 個の格子点を含むように、分割することを考えます。次の平行体（平行四辺形を多次元に拡張したもの）は格子点の一つである原点のみを含み、各基底ベクトルの整数倍ずつずらしていくことで、空間全体に敷き詰めることができます。

$$\left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i \;\middle|\; x_i \in \mathbb{R},\ 0 \leq x_i < 1 \right\}$$



▲図 2.2　Gram-Schmidt の直交化の例

基底ベクトルと同様に、$\mathbf{b}_1^*,\ \mathbf{b}_2^*,\ \ldots,\ \mathbf{b}_n^*$ もこれらのベクトルを行に持つ行列を $\mathbf{B}^*$ で表わすことにします。格子 $\mathcal{L}(\mathbf{B})$ と $\mathcal{L}(\mathbf{B}^*)$ は一般に一致しません。ある基底ベクトルの整数倍を他の基底ベクトルに足し引きしても格子は変化しませんが、$\mu_{i,j}$ が整数とは限らないからです。

Gram-Schmidt の直交化を行う Python のコードは次のようになります。

▼ Gram-Schmidt の直交化

```
 1: from fractions import Fraction
:
 5: # 内積
 6: def prod(u, v):
 7:   return sum(x*y for x, y in zip(u, v))
 8:
 9: # Gram-Schmidtの直交化
10: # GSOベクトルのノルムの2乗 [|b0|^2, |b1|^2, ...] とGSO係数 mu を返す。
11: def GSO(M):
12:   n = len(M)
13:   mu = [[0]*n for _ in range(n)]
14:   B = [m[:] for m in M]
15:   for i in range(n):
16:     mu[i] = [0]*i
17:     for j in range(i):
18:       mu[i][j] = Fraction(prod(M[i], B[j]), prod(B[j], B[j]))
19:       for k in range(n):
20:         B[i][k] -= mu[i][j]*B[j][k]
21:   return [prod(b, b) for b in B], mu
```

記号 $\mathbf{B}^*$ がソースコード中で表記できないため、基底行列 $\mathbf{B}$ を M とし、GSO ベクトルの行列 $\mathbf{B}^*$ を B としています。また、$\mathbf{B}^*$ のうち LLL 基底簡約で使うのは $\|\mathbf{b}_i^*\|^2$ だけなので、$\mathbf{B}^*$ ではなくこれらの値を返しています。

ものです。つまり、雑に言えば、$\mathbf{B}$ の各基底ベクトルが完全には直交していない（$\mathbf{B}$ と $\mathbf{B}^*$ が一致しない）ことと、δ の分を除いて、$\mathbf{B}$ の各基底ベクトルはノルムの昇順に並んでいるということになります。

LLL 基底簡約の 2 個の条件や Gram-Schmidt の直交化の定義などから、LLL 簡約された基底 $\mathbf{B}$ は次の性質を持ちます。なお、このタイミングでどのような性質を挙げるかは文献によって異なっており、定番の性質があるわけではありません。ここでは参考文献 [2] に書かれている性質を紹介します。

1. 任意の $1 \leq j \leq i \leq n$ に対して、$\left\|\mathbf{b}_j^*\right\|^2 \leq \alpha^{i-j} \left\|\mathbf{b}_i^*\right\|^2$ が成り立つ。
2. 不等式 $\|\mathbf{b}_1\| \leq \alpha^{\frac{n-1}{4}} \left|\det\left(\mathbf{B}\right)\right|^{\frac{1}{n}}$ が成り立つ。
3. 任意の $1 \leq i \leq n$ に対して、$\|\mathbf{b}_1\| \leq \alpha^{\frac{n-1}{2}} \lambda_i\left(\mathcal{L}\left(\mathbf{B}\right)\right)$ が成り立つ。
4. 不等式 $\prod_{i=1}^{n} \|\mathbf{b}_i\| \leq \alpha^{\frac{n(n-1)}{4}} \left|\det\left(\mathbf{B}\right)\right|$ が成り立つ。

ここで、$\alpha = \frac{4}{4\delta - 1}$ です。パラメタ $\delta = \frac{3}{4}$ のとき $\alpha = \frac{1}{2}$、$\delta = 1$ のとき $\alpha = \frac{3}{4}$ となります。

3 番目の性質の $\lambda_i\left(\mathcal{L}\left(\mathbf{B}\right)\right)$ は、格子 $\mathcal{L}\left(\mathbf{B}\right)$ の i 番目の逐次最小です。ノルムが x 以下の格子点の中から一次独立な格子点を i 個選べるような最小の x が i 番目の逐次最小です。特に、$\lambda_1\left(\mathcal{L}\left(\mathbf{B}\right)\right)$ は $\mathcal{L}\left(\mathbf{B}\right)$ に含まれる原点以外のノルムが最短であるベクトルのノルムになります。

CTF の Crypto の問題を解くときに良く使うのは、2 番目と 3 番目の性質です。Crypto の問題を解くときには、与えられたベクトルの整数係数線形結合のうち原点以外で最短であるものを近似的に求めるツールとして、LLL 基底簡約を使うからです。2 番目の性質では基底行列の行列式に対して、3 番目の性質では真の最短ベクトルに対して、どのくらいの長さのベクトルが求められるかが分かります。なお、基底が与えられたときにノルムが最短のベクトルを求める問題（Shortest Vector Problem, SVP と言います）の効率的な解法は知られていません。

LLL 基底簡約のアルゴリズムは意外とシンプルです。変数 k を 2 で初期化し、$k \leq n$ が満たされている間、$\mathbf{B}^*$ と $\mu_{i,j}$ を更新しながら、次の手順を繰り返します。

1. $j = k - 1$ から 0 について、$\mathbf{b}_i' = \mathbf{b}_i - \lfloor \mu_{i,j} \rceil \mathbf{b}_j$ と更新する。$\lfloor x \rceil$ は x に最も近い整数です。そのような整数が 2 個存在するときは、どちらでも構いません。
2. $\|\mathbf{b}_k^*\|^2 \geq \left(\delta - \mu_{k,k-1}^2\right) \left\|\mathbf{b}_{k-1}^*\right\|^2$ ならば、（Lovász 条件を満たすならば）$k = k + 1$ と更新する。そうでなければ、$\mathbf{b}_{k-1}$ と $\mathbf{b}_k$ を交換し、$k = k - 1$ と更新する。

変数 k の値が増えたり減ったりするので、繰り返しの回数が行列のサイズなどの多項式で抑えられるのか、そもそも繰り返しが停止するのかが気になります。しかし、基底ベクトルの要素が整数であり、$\delta < 1$ を満たすならば、繰り返し回数は行列のサイズと要

```
39:         for l in range(i):
40:           mu[k][l] -= q*mu[i][l]
41:         mu[k][i] -= q
42:
43:     # Lovasz条件
44:     if B[k]>=(delta-mu[k][k-1]**2)*B[k-1]:
45:       k += 1
46:     else:
47:       # M[k] と M[k-1] の交換
48:       M[k], M[k-1] = M[k-1], M[k]
49:       # B の更新
50:       b_old = B[k-1]
51:       B[k-1] = B[k] + mu[k][k-1]**2*b_old
52:       B[k] *= b_old/B[k-1]
53:       # mu の更新
54:       mu_old = mu[k][k-1]
55:       mu[k][k-1] *= b_old/B[k-1]
56:       for i in range(k-1):
57:         t = mu[k][i]
58:         mu[k][i] = mu[k-1][i]
59:         mu[k-1][i] = t
60:       for i in range(k+1, n):
61:         t = mu[i][k]
62:         mu[i][k] = mu[i][k-1] - mu_old*t
63:         mu[i][k-1] = t + mu[k][k-1]*mu[i][k]
64:       k = max(1, k-1)
65:   return M
```

配列 B は、本文中の $\|\mathbf{b}_1^*\|^2$, $\|\mathbf{b}_2^*\|^2$, ..., $\|\mathbf{b}_n^*\|^2$ を格納しています。行列 M を書き換えるたびに Gram-Schmidt の直交化を行うのではなく、B と mu の変化するところだけを上手く更新することで、処理が高速になります。

2.4 LLL 基底簡約の使用例

CTF の Crypto の問題を解くときには、LLL 基底簡約を行った後の $\mathbf{b}_1$ を良く使うと前述しましたが、Coppersmith 法では異なります。そこで、Crypto 問で良くある LLL 基底簡約の使い方の例をここで紹介します。

■問題:

整数の組 $\mathbf{x} = \{50,\ 137,\ 38,\ 197,\ 120,\ 206\}$ の中から、和が $S = 295$ になるようにいくつかの整数を選べ。

これは部分和問題と呼ばれる問題です。この問題を効率的に解くことができれば、

第 3 章

Coppersmith 法

本章では本書のタイトルでもある Coppersmith 法の解説をします。Coppersmith 法は、与えられた合同方程式と同じ解を持つ（合同方程式ではない）方程式を作ることによって、合同方程式を解く手法です。

3.1 Coppersmith 法とは

Coppersmith 法は、パラメタ $\epsilon > 0$ を取り、k 次の合同方程式

$$p(x) = x^k + a_{k-1}x^{k-1} + \cdots + a_2x^2 + a_1x + a_0 \equiv 0 \mod N$$

の小さな解 x_0 $\left(|x_0| < \frac{1}{2}N^{\frac{1}{k}-\epsilon}\right)$ を求める手法です。多項式 $p(x)$ は、最高次の係数が 1 である（モニック多項式である）必要があります。解の条件から分かるように、ϵ が小さいほど大きな解が得られますが、代わりに計算量が増えます。SageMath の実装のデフォルト値は $\epsilon = \frac{1}{8}$ です[*1]。

3.2 Coppersmith 法の実装

文章だけでは分かりづらいので、具体的な例を挙げつつ、Coppersmith 法を解説していくことにします。例として、以下の合同方程式を解きます。

$$p(x) = x^2 + 59x + 78 \equiv 0 \mod 100$$

[*1] https://doc.sagemath.org/html/en/reference/polynomial_rings/sage/rings/polynomial/polynomial_modn_dense_ntl.html#sage.rings.polynomial.polynomial_modn_dense_ntl.small_roots

この $p(x)$ は 2 次式なので、$k = 2$ です。この合同方程式は解 $x_0 = 2$ を持ちます。パラメタ $\epsilon = \frac{1}{8}$ とします。

まずは、X と h を次のように決定します。

$$\begin{aligned} X &= \frac{1}{2}N^{\frac{1}{k}-\epsilon}, \\ h &\geq \max\left(\frac{7}{k},\ \frac{k+\epsilon k-1}{\epsilon k^2}\right) \end{aligned}$$

例のパラメタに対しては、$X = 2.8117\ldots$ であり、$h \geq \max\left(\frac{7}{2},\ \frac{5}{2}\right) = \frac{7}{2}$ なので、$h = 4$ とします。

パラメタ X は、Coppersmith 法で求められる解の絶対値の上限です。パラメタ h をこのように定めることで、$h - 1 \geq (hk-1)\left(\frac{1}{k} - \epsilon\right)$ と $hk \geq 7$ が成り立ちます。これは、あとで作る格子の基本平行体の体積、すなわち基底行列の行列式の絶対値、を大きくするために必要な条件です。

▼ パラメタの決定

```
67: # p(x) = Σ[0<=i<=k]A[i]*x^i = 0 mod N について、
68: # |x0|<(1/2)*N^(1/k)-e を満たす解 x0 を返す。
69: def coppersmith(A, N, e=Fraction(1, 8)):
70:   # パラメタの決定
71:   k = len(A)-1
72:   h = math.ceil(max(7/k, (k+e*k-1)/(e*k**2)))
73:   X = int(N**(1/k-e)/2)
74:   print(f"{k=}")
75:   print(f"{h=}")
76:   print(f"{X=}")
```

定義通りに定めると、X は実数となります。実装と例では、扱いやすくするために整数に丸めます。

ここで、整数 i と j について、$x_0^i \left(p\left(x_0\right)\right)^j$ が満たす条件を考えます。合同式 $p\left(x_0\right) \equiv 0 \mod N$ が成り立つことから、適当な整数 y_0 を使って $p\left(x_0\right) = y_0 N$ と書くことができます。よって、$x_0^i \left(p\left(x_0\right)\right)^j = x_0^i y_0^j N^j \equiv 0 \mod N^j$ が成り立ちます。合同式 $p(x) = x^2 + 59x + 78 \equiv 0 \mod 100$ からは、$\left(p(x)\right)^2 = x^4 + 118x^3 + 3637x^2 + 9204x + 6084 \equiv 0 \mod 10000$ が導けます。

サイズ $(2hk-k) \times (2hk-k)$ の基底行列 $\mathbf{M}$ を作ります。基底行列 $\mathbf{M}$ は、4 個の部分に分けられます。左上の $hk \times hk$ の部分は、$M_{i,i} = X^{-i+1}/\sqrt{hk}$ とし、それ以外の要素は 0 です。右上の $hk \times (hk-k)$ の部分は、$1 \leq i \leq hk$ と $1 \leq j \leq h-1$、$1 \leq l \leq k$ に対して、$M_{i,hk+(j-1)k+l}$ を $x^l \left(p(x)\right)^j$ の x^{i-1} の係数とします。左下の $(hk-k) \times hk$ の部分は零行列です。右下の $(hk-k) \times (hk-k)$ の部分は、$1 \leq j \leq h-1$ と $1 \leq l \leq k$ に対して、$M_{hk+(j-1)k+l,hk+(j-1)k+l} = N^j$ とし、それ以外の要素は 0 にします。以降、列数

合同方程式 $p(x) = x^2 + 59x + 78 \equiv 0 \mod 100$ に対する基底行列 $\mathbf{M}$ は次のようになります。ただし、紙面の都合上、本来は $h = 4$ ですが、$h = 3$ に対しての結果を示しています。

$$\mathbf{M} = \begin{pmatrix} 32 & 0 & 0 & 0 & 0 & 0 & 78 & 0 & 6084 & 0 \\ 0 & 16 & 0 & 0 & 0 & 0 & 59 & 78 & 9204 & 6084 \\ 0 & 0 & 8 & 0 & 0 & 0 & 1 & 59 & 3637 & 9204 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 1 & 118 & 3637 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 1 & 118 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 100 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 100 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10000 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10000 \end{pmatrix}$$

最高次の係数が 1 であるという制約から、$1 \leq i \leq hk - k$ に対して、$\mathbf{M}_{k+i,hk+i} = 1$ となります。この部分を使い、行列の基本変形（行の交換と、ある行の整数倍を他の行から引くこと）によって、対角成分の右側の部分を 1 に、右側のそれ以外の要素を 0 にします。こうして得られた行列を $\widetilde{\mathbf{M}}$ とします。

▼ 右上を零行列、右下を単位行列にする

```
100:   # 右上を零行列、右下を単位行列にする。
101:   M = M[h*k:]+M[:h*k]
102:   for i in range(2*h*k-k):
103:     for j in range(h*k, 2*h*k-k):
104:       if j!=i:
105:         t = M[i][j]
106:         for l in range(2*h*k-k):
107:           M[i][l] -= t*M[j][l]
```

解くべきは次の連立方程式です。

$$\begin{aligned}
-1f_1 &\quad -2f_2 &\quad -4f_3 &\quad -8f_4 &\quad -16f_5 &\quad -32 &= 0,\\
-7f_1 &\quad +11f_2 &\quad -3f_3 &\quad +19f_4 &\quad +13f_5 &\quad +51 &= 0,\\
5f_1 &\quad +20f_2 &\quad +30f_3 &\quad -30f_4 &\quad -70f_5 &\quad -430 &= 0,\\
10f_1 &\quad +15f_2 &\quad -65f_3 &\quad +65f_4 &\quad -165f_5 &\quad +65 &= 0,\\
3f_1 &\quad +35f_2 &\quad +1f_3 &\quad -289f_4 &\quad +73f_5 &\quad +335 &= 0
\end{aligned}$$

答えは、$f_1 = -6$, $f_2 = -2$, $f_3 = \frac{17}{2}$, $f_4 = 0$, $f_5 = -\frac{7}{2}$ となり、$f_1,\ f_2,\ \ldots,\ f_6$ に 2 を掛けて係数とし、次の方程式が得られます。

$$p'(x) = -12 - 4x + 17x^2 - 7x^4 + 2x^5 = 0$$

解ける形にはなりましたが、多項式の整数解を求めるのは少々面倒です。解答スクリプトでは、$x < |X|$ を満たす 2 個の値 x_1 と x_2 を選び、$p'(x_1)$ と $p'(x_2)$ の符合が異なるならば二分探索で $p'(x_0) = 0$ となる x_0 を求め、元の合同方程式の解となるかを確認する、ということを繰り返すようにしました。適切な数値精度で近似したニュートン法を用いるなどするべきかもしれません。

▼ $p'(x) = 0$ を解く

```
146:   # ΣF[i]*x^i = 0 を解く。
147:   p = lambda x: F[0]+sum(F[i]*x**i for i in range(1, n))
148:   while True:
149:     x1 = random.randint(-X, X+1)
150:     x2 = random.randint(-X, X+1)
151:     if p(x1)*p(x2)<0:
152:       while abs(x1-x2)>1:
153:         m = (x1+x2)//2
154:         if p(m)*p(x1)>=0:
155:           x1 = m
156:         else:
157:           x2 = m
158:       if p(x1)==0:
159:         # 元の方程式の解となっていることも確認する。
160:         if (A[0]+sum(A[i]*x1**i for i in range(1, k+1)))%N==0:
161:           x0 = x1
162:           break
163:   return x0
```

なお、$x < |X|$ で $p(x) \equiv 0 \mod N$ を満たすような x は常に $p'(x) = 0$ を満たしますが、逆は成り立たないことに注意が必要です。例の $p'(x) = 0$ は、$p(x) \equiv 0 \mod N$ の解でもある $x = 2$ の他に、$x = -1$ という解も持ちます。これは $(-1,\ 1,\ -1,\ 1,\ -1,\ 1)$ が $\frac{4}{25}\mathbf{b_1'} + \frac{3}{25}\mathbf{b_2'}$ と表わせるからです。係数が非整数なので格子点ではないですが、行列 $\mathbf{B'}$

```
152:       while abs(x1-x2)>1:
153:         m = (x1+x2)//2
154:         if p(m)*p(x1)>=0:
155:           x1 = m
156:         else:
157:           x2 = m
158:       if p(x1)==0:
159:         # 元の方程式の解となっていることも確認する。
160:         if (A[0]+sum(A[i]*x1**i for i in range(1, k+1)))%N==0:
161:           x0 = x1
162:           break
163:   return x0
164:
165: N = 135896252889455179306044561856073657176589282065060485982049884024159935440451648268809083278093186378968141115107175023955352285741931510695513569631865587980780116850562946289647504541616229544460216616423916073902516352634221984329472372337896406341980543263900139265196004232936560031956301070039887638201
166: e = 3
167: c = 104064417885709974759351196219773048348376455313832829209947346446874683031677099263660660465743321265675101447205286367005536581370429707934588639596542365006675867115121072960551019978628425500909321182750343178517807287529652070367005206132362187414189170498876416717871986142239844250496022317643124516521
168:
169: m0 = """
170: +>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+
171: |    FLAG{""" + "\0"*25 +       """}    |
172: +>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+>-<+
173: """
174: m0 = int.from_bytes(m0.encode(), "big")
175:
176: # A[i]: (m0 + x * 256^49)^e - c の x^i の係数
177: A = [
178:   m0**3 - c,
179:   3 * m0**2 * 256**49,
180:   3 * m0 * (256**49)**2,
181:   (256**49)**3,
182: ]
183:
184: # A[3]を1にする。
185: for i in range(4):
186:   A[i] = A[i] * pow((256**49)**3, -1, N) % N
187:
188: flag = coppersmith(A, N)
189: print(flag.to_bytes(25, "big").decode())
```