第16章 古典密码分析

16.1 基本原理

古典密码是指代换、置换密码或其简单变形，例如仿射密码、多表代换密码、棋盘密码、移位密码等。为了方便讨论，在此给出一些常见古典密码算法的形式化描述。不做特别说明，用$\mathcal{M}$表示明文空间，$\mathcal{C}$表示密文空间，$\mathcal{K}$表示密钥空间，$E_k(x)$表示利用加密密钥k加密明文x，$D_k(y)$表示利用解密密钥k解密密文y。

16.1.1 代换密码

一个代换密码是指从明文空间到密文空间的一个双射，或者说置换π：$\mathcal{M} \to \mathcal{C}$。这个定义也适合一般的分组密码，但是古典密码通常研究比较简单的单字母代换或者几个字母组合的代换。

例 16-1 仿射密码。将明密文大写英文字母A~Z依次编码为0~25，记$\mathcal{M} = \mathcal{C} = \mathbb{Z}_{26}$，密钥为一对参数$(k, a)$，记$\mathcal{K} = \mathbb{Z}_{26}^* \times \mathbb{Z}_{26}$，对于任意密钥$(k, a) \in \mathcal{K}$，

$$E_{(k,a)}(x) = kx + a(\text{mod } 26)$$
$$D_{(k,a)}(y) = k^{-1}(y - a) \text{ (mod } 26)$$

当$(k, a) = (1,3)$时，即为熟知的凯撒密码，对应的置换π为：

0	1	2	3	…	22	23	24	25
3	4	5	6	…	25	0	1	2

其逆置换π^{-1}为：

0	1	2	3	…	22	23	24	25
23	24	25	0	…	19	20	21	22

依照上面的定义，若取密钥$(k, a) = (3,13)$，明文为"GOTO"，由于′G′、′O′、′T′在$\mathbb{Z}_{26}$编码中依次为 6，14，19，经加密

$$E_{(3,13)}(6) \equiv 3 \times 6 + 13 \equiv 5(\text{mod } 26)$$
$$E_{(3,13)}(14) \equiv 3 \times 14 + 13 \equiv 3(\text{mod } 26)$$
$$E_{(3,13)}(19) \equiv 3 \times 19 + 13 \equiv 18(\text{mod } 26)$$

而 5，3，18 解码后依次为′F′、′D′、′S′，所以密文为"FDSD"。

如果给定密文"FDSD"，因为$3^{-1} \equiv 9(\text{mod } 26)$，所以，经解密

$$D_{(3,13)}(5) \equiv 9 \times (5 - 13) \equiv 6(\text{mod } 26)$$
$$D_{(3,13)}(3) \equiv 9 \times (3 - 13) \equiv 14(\text{mod } 26)$$
$$D_{(3,13)}(18) \equiv 9 \times (18 - 13) \equiv 19(\text{mod } 26)$$

对 6，14，19 解码后得到明文"GOTO"。

例 16-2 简单替换密码。将明文字母替换为与之唯一对应且不同的密文字母，记$\mathcal{M} = \mathcal{C} = \mathbb{Z}_{26}$，密钥为一个随机置换$\pi$，$\mathcal{K}$为所有置换$\pi$的集合。

$$E_k(x) = \pi_k(x)$$
$$D_k(y) = \pi_k^{-1}(y)$$

若选取随机置换π作为密钥

x	A	B	C	D	E	F	G	H	I	J	K	L	M

$\pi(x)$	P	H	Q	G	I	U	M	E	A	Y	L	N	O
x	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
$\pi(x)$	F	D	X	J	K	R	C	V	S	T	Z	W	B

若明文为"GOTO"，加密时，依据上述置换，'G'被替换为'M'，'O'被替换为'D'，'T'被替换为'C'，所以密文为"MDCD"。

解密时，使用密钥的逆置换即可将密文重新替换回明文，得到明文"GOTO"。

埃特巴什密码就是简单替换密码的一种特例，它的密钥，即置换π为

x	A	B	C	D	E	F	G	H	I	J	K	L	M
$\pi(x)$	Z	Y	X	W	V	U	T	S	R	Q	P	O	N
x	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
$\pi(x)$	M	L	K	J	I	H	G	F	E	D	C	B	A

可见，它的加密置换就是英文字母序的倒序。

简单替换密码与仿射密码之间的主要区别在于其明密文置换π不是通过简单的移位、仿射生成的，而是完全随机的，这也使得其密钥空间$\mathcal{K}$大小高达26!，远大于仿射密码的26×26，破译难度大大提升。

例 16-3 维吉尼亚密码。仿射密码与简单替换密码都属于单表代换密码，而维吉尼亚密码则属于多表代换密码。明文、密文、密钥均为长度为 m 的大写字母组合，记$\mathcal{M} = \mathcal{C} = \mathcal{K} = (\mathbb{Z}_{26})^m$，对于任意密钥$(k_1, k_2, \dots, k_m) \in \mathcal{K}$，

$$E_{(k_1,k_2,\dots,k_m)}(x) \equiv (x_1 + k_1, x_2 + k_2, \dots, x_m + k_m) \pmod{26}$$
$$D_{(k_1,k_2,\dots,k_m)}(y) \equiv (y_1 - k_1, y_2 - k_2, \dots, y_m - k_m) \pmod{26}$$

若取$m = 6$，密钥字为"CIPHER"，将其在$\mathbb{Z}_{26}$中编码为(2,8,15,7,4,17)，明文为"SAFECRYPTOSYSTEM"，按类似于分组密码的思路，将明文中每连续 6 个字母为一组，分别使用密钥进行加密

$$E_{(2,8,15,7,4,17)}(x) \equiv (18 + 2, 0 + 8, \dots, 12 + 7) \pmod{26} \equiv (20, 8, \dots, 19)$$

明文	S	A	F	E	C	R	Y	P	T	O	S	Y	S	T	E	M
x_i	18	0	5	4	2	17	24	15	19	14	18	24	18	19	4	12
k_i	2	8	15	7	4	17	2	8	15	7	4	17	2	8	15	7
y_i	20	8	20	11	6	8	0	23	8	21	22	15	20	1	19	19
密文	U	I	U	L	G	I	A	X	I	V	W	P	U	B	T	T

最后，将分组加密后的密文拼接在一起，得到"UIULGIAXIVWPUBTT"。

解密时，同样将密文中的每连续 6 个字母为一组，分别使用密钥进行解密，即按位置y_i减掉对应密钥k_i，最后将分组解密后的明文拼接在一起，重新得到明文。

可见，多表代换密码与单表代换密码主要区别在于，多表代换密码在进行代换时，一个字母可能会被映射为m个字母中的某一个，而非确定的某个字母。

例 16-4 希尔密码。明文、密文为长度为 m 的大写字母组合，记$\mathcal{M}=\mathcal{C}=(\mathbb{Z}_{26})^m$，$\mathcal{K}$为定义在$\mathbb{Z}_{26}$上的$m\times m$可逆矩阵集合，对于任意明文$x=(x_1,x_2,\dots,x_m)\in\mathcal{M}$，密钥$k\in\mathcal{K}$

$$E_k(x)=kx$$
$$D_k(x)=k^{-1}x$$

其中，上述运算皆为模 26 意义下的，k^{-1}代表k的逆矩阵，只有矩阵k的行列式与 26 互质，矩阵k才是可逆的。

若取密钥

$$k=\begin{bmatrix}2&4&5\\9&2&1\\3&17&7\end{bmatrix}$$

明文为"ATT"，由于′A′、′T′、′T′在$\mathbb{Z}_{26}$编码中依次为 0，19，19，经加密

$$E_k(x)=\begin{bmatrix}2&4&5\\9&2&1\\3&17&7\end{bmatrix}\begin{bmatrix}0\\19\\19\end{bmatrix}=\begin{bmatrix}171\\57\\456\end{bmatrix}(mod\ 26)\equiv\begin{bmatrix}15\\5\\14\end{bmatrix}$$

得到密文矩阵[15,5,14]，其中 15，5，14 解码后依次为′P′、′F′、′O′，所以密文为"PFO"。

解密时，首先需要计算k的逆矩阵k^{-1}

$$k^{-1}=|k|^{-1}\times adj(k)=489^{-1}\times\begin{bmatrix}-3&57&-6\\-60&-1&43\\147&-22&-32\end{bmatrix}(mod\ 26)$$
$$\equiv 5\times\begin{bmatrix}23&5&20\\18&25&17\\17&4&20\end{bmatrix}(mod\ 26)\equiv\begin{bmatrix}11&25&22\\12&21&7\\7&20&22\end{bmatrix}$$

其中，$|k|$代表矩阵k的行列式，$adj(k)$代表矩阵k的伴随矩阵，上述运算全部在$\mathbb{Z}_{26}$模 26 的意义下进行。

对于密文"PFO"，在$\mathbb{Z}_{26}$编码中依次为 15，5，14，经解密

$$D_k(x)=\begin{bmatrix}11&25&22\\12&21&7\\7&20&22\end{bmatrix}\begin{bmatrix}15\\5\\14\end{bmatrix}=\begin{bmatrix}598\\383\\513\end{bmatrix}(mod\ 26)=\begin{bmatrix}0\\19\\19\end{bmatrix}$$

重新求得明文矩阵[0,19,19]，即明文"ATT"。

例 16-5 普莱费尔密码。明文、密文为长度为 m 的大写字母组合，记$\mathcal{M}=\mathcal{C}=(\mathbb{Z}_{26})^m$，$\mathcal{K}$为满足特定规律的$5\times5$大写字母矩阵集合。

生成一个密钥k时，首先随机选取一个大写字母串，去除其中重复出现的字母后，将剩下的字母从左到右、从上到下依次放入大小为5×5的矩阵内，矩阵剩下的空间则依次填充$A-Z$中未出现过的字母，在上述过程中，将I和J视作同一字母。最终产生的5×5字母矩阵，即为密钥k。

加密前，首先将明文每两个字母为一组进行分组。分组时，若某一组为两个相同字母，则在这两个相同字母之间插入一个字母X，随后重新进行分组。分组完成后，若最后剩余一个字母，则加入一个字母X作为最后一组。

加密时，对每组字母分别进行加密，首先找到此组内两个字母在矩阵内的坐标$(x_1,y_1),(x_2,y_2)$，依据下述原则在矩阵k中定位出另外两个字母即为密文

- 若$x_1\neq x_2$且$y_1\neq y_2$，则取$(x_1,y_2),(x_2,y_1)$
- 若$x_1=x_2$且$y_1\neq y_2$，则取$(x_1,(y_1+1)\ mod\ 5),(x_2,(y_2+1)\ mod\ 5)$
- 若$x_1\neq x_2$且$y_1=y_2$，则取$\big((x_1+1)\ mod\ 5,y_1\big),\big((x_2+1)\ mod\ 5,y_2\big)$

解密时，同样以两个密文字母为一组，依据下述原则矩阵k中定位出另外两个字母即为明文

- 若$x_1\neq x_2$且$y_1\neq y_2$，则取$(x_1,y_2),(x_2,y_1)$
- 若$x_1=x_2$且$y_1\neq y_2$，则取$(x_1,(y_1-1)\ mod\ 5),(x_2,(y_2-1)\ mod\ 5)$

- 若$x_1 \neq x_2$且$y_1 = y_2$，则取$\big((x_1 - 1)\ mod\ 5, y_1\big), \big((x_2 - 1)\ mod\ 5, y_2\big)$

若以 PLAYFAIR EXAMPLE 为密钥字，按照定义，可以生成密钥矩阵k：

$x \setminus y$	0	1	2	3	4
0	P	L	A	Y	F
1	I/J	R	E	X	M
2	B	C	D	G	H
3	K	N	O	Q	S
4	T	U	V	W	Z

若明文为"$COMMANDY$"，首先将其按照上述定义进行分组，得到

CO、MX、MA、ND、YX

以纵向为x轴，以横向为y轴，依次进行加密，可得

DN、IM、EF、OC、XG

拼接在一起，即可得到密文"$DNIMEFOCXG$"。

除了上述明密文空间均为英文字母或其组合的各类单表代换密码和多表代换密码外，代换密码中还存在一些密文并非字母的特殊密码体制，例如棋盘密码、摩斯密码等。

例 16-6 棋盘密码。也被称为波利比奥斯方阵*密码*。明文为大写字母，记$\mathcal{M} = \mathbb{Z}_{26}$，$\mathcal{K}$为$5 \times 5$的大写字母矩阵集合，其中，将$I$和$J$视作同一字母，密文为矩阵坐标，记$\mathcal{C} = \mathbb{Z}_5 \times \mathbb{Z}_5$。

$$E_k(p) = (p_x, p_y)$$
$$D_k(p_x, p_y) = p$$

若选取以下矩阵为密钥k，明文为"HELLO"。

	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I/J	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

′H′在k中的坐标为(2,3)，因此′H′对应的密文为 23；′E′在k中的坐标为(1,5)，因此′E′对应的密文为 15；依次类推，得到密文"2315313134"。

解密时，以密文中每相邻两个数字为一组坐标，重新从密钥矩阵k定位回明文即可。

若将前述坐标中的数字$1-5$替换为字母"$ADFGX$"，便是知名的 ADFGX 密码。

例 16-7 摩斯密码。也被称为摩斯电码。明文包含大写字母、数字等各种常用字符，密文

是由点信号、长信号组成的状态代码，加解密时使用固定的置换表进行代换

字符	电码	字符	电码	字符	电码
A	• —	N	— •	0	— — — — —
B	— • • •	O	— — —	1	• — — — —
C	— • — •	P	• — — •	2	• • — — —
D	— • •	Q	— — • —	3	• • • — —
E	•	R	• — •	4	• • • • —
F	• • — •	S	• • •	5	• • • • •
G	— — •	T	—	6	— • • • •
H	• • • •	U	• • —	7	— — • • •
I	• •	V	• • • —	8	— — — • •
J	• — — —	W	• — —	9	— — — — •
K	— • —	X	— • • —		
L	• — • •	Y	— • — —		
M	— —	Z	— — • •		

在加密一串明文时，通常会使用′/′等符号来分隔不同的密文。例如，若取明文"$HELLO$"，对照上述置换，即摩斯电码表，可以得到密文

$$\bullet\bullet\bullet\bullet/\bullet/\bullet — \bullet\bullet/\bullet — \bullet\bullet/— — —$$

在实际应用中，还常常使用二进制即 0 代表点信号、1 代表长信号，这同样也是摩斯密码的一种表示形式。

16.1.2 置换密码

由前文可知，代换密码是依据置换π对明文字母进行代换，来完成加密的。而置换密码则是依据置换π来打乱明文中的字母顺序，达到加密的目的。

在一般的置换密码体制中，明密文为同一串长度为m的大写字母组合，记$\mathcal{M} = \mathcal{C} = (\mathbb{Z}_{26})^m$，$\mathcal{K}$为定义在集合$\{1,2,\dots,m\}$上的置换集合，对于任意密钥$k$，即置换$\pi$，

$$E_\pi(x_1, x_2, \dots, x_m) = (x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(m)})$$

$$D_\pi(y_1, y_2, \dots, y_m) = (y_{\pi^{-1}(1)}, y_{\pi^{-1}(2)}, \dots, y_{\pi^{-1}(m)})$$

其中，π^{-1}是置换π的逆置换。

例如，若取$m = 6$，密钥k（即置换π）为

x	1	2	3	4	5	6
$\pi(x)$	3	5	1	6	4	2

若加密明文"$SOMEEXAMPLES$"，首先每$m = 6$个字母为一组进行分组，分为"$SOMEEX$"和"$AMPLES$"，分别进行加密，依据置换π，"$SOMEEX$"被置换为"$MXSEOE$"，"$AMPLES$"被置换为"$PSAEML$"，最后拼接在一起，得到密文"$MXSEOEPSAEML$"。

解密时，将置换π中的映射关系对调，即可得到逆置换π^{-1}

y	1	2	3	4	5	6
$\pi^{-1}(y)$	3	6	1	5	2	4

依据逆置换π^{-1}，可以将密文重新置换回明文"$SOMEEXAMPLES$"。

此外，还有一些依据特定规则进行置换操作的特殊置换密码体制，例如栅栏密码、列移位密码等。

例 16-8 栅栏密码。将明文按每m个字母为一组，分为n组，记$\mathcal{M} = \mathcal{C} = (\mathbb{Z}_{26})^{mn}$

$$E\left(x_{(1,1)},x_{(1,2)},..,x_{(1,m)},x_{(2,1)},\dots,x_{(n,m)}\right)=\left(x_{(1,1)},x_{(2,1)},..,x_{(n,1)},x_{(1,2)},x_{(2,2)},\dots,x_{(n,m)}\right)$$
$$D\left(y_{(1,1)},y_{(1,2)},..,y_{(1,n)},y_{(2,1)},\dots,y_{(m,n)}\right)=\left(y_{(1,1)},y_{(2,1)},..,y_{(m,1)},y_{(1,2)},y_{(2,2)},\dots,y_{(m,n)}\right)$$

换句话说，加密时，需要依次取出每组的第一个字母，拼接在一起，再依次取出每组的第二个字母，依旧拼接在一起，依此类推，最终得到密文。解密时，则采取与加密时相反的分组方式，将密文每n个字母为一组，分为m组，按同样的置换方法重新得到明文。

例如，若明文为"$THEREISACIPHER$"，取$m=2$，则$n=7$，得到各个分组"TH"、"ER"、"EI"、"SA"、"CI"、"PH"、"ER"。

加密时，取出每组的第一个字母，拼接在一起，得到"$TEESCPE$"，随后，取出每组的第二个字母，再拼接在一起，得到"$HRIAIHR$"，将它们连接在一起，便得到了密文"$TEESCPEHRIAIHR$"。

解密时，则以 7 个字母为一组，"$TEESCPE$"、"$HRIAIHR$"，按上述定义进行置换、拼接，得回明文"$THEREISACIPHER$"。

例 16-9 列移位密码。将明文按每m个字母为一组，分为n组，记$\mathcal{M}=\mathcal{C}=(\mathbb{Z}_{26})^{mn}$，密钥是长度为$m$的一串大写字母，记$\mathcal{K}=(\mathbb{Z}_{26})^{m}$。

为便于说明，取$m=7$，则$n=5$，明文

"$THEQUICKBROWNFOXJUMPSOVERTHELAZYDOG$"

将明文填入$n\times m$即5×7的矩阵中

T	H	E	Q	U	I	C
K	B	R	O	W	N	F
O	X	J	U	M	P	S
O	V	E	R	T	H	E
L	A	Z	Y	D	O	G

加密时，使用长度为 7 的随机密钥"$HOWAREU$"，按"$HOWAREU$"各字母的字母序进行编号，即$'A'=1$、$'E'=2$、$'H'=3$、$'O'=4$、$'R'=5$、$'U'=6$、$'W'=7$，作为序号，结合它们在"$HOWAREU$"内出现的位置，作为列号。将上述矩阵内的各列按序号拼接在一起，即依次取出上述矩阵中的第 4（$'A'$）列、第 6（$'E'$）列、第 1（$'H'$）列、第 2（$'O'$）列、第 5（$'R'$）列、第 7（$'U'$）列、第 3（$'W'$）列，将它们拼接在一起，得到密文

"$QOURY\ INPHO\ TKOOL\ HBXVA\ UWMTD\ CFSEG\ ERJEZ$"

解密时，将密文按每n个字母为一组，分为m组，再次填入一个$n\times m$的矩阵中

Q	I	T	H	U	C	E
O	N	K	B	W	F	R
U	P	O	X	M	S	J
R	H	O	V	T	E	E
Y	O	L	A	D	G	Z

随后，以密钥字"$HOWAREU$"中各字母的字母序作为列号，在"$HOWAREU$"内出现的位置作为序号，重新将矩阵内的各列按序号拼接在一起，即可得回明文。

16.1.3 古典密码分析方法

在进行密码分析时，首先需要确定攻击模型，随后依据模型的已知信息来展开分析有以下几种常见的攻击模型：

唯密文攻击：分析者仅已知密文串y。

已知明文攻击：分析者已知明文串x及其对应密文串y。

选择明文攻击：分析者可以自己进行加密操作，可以选择任意一个明文串x，并可以获得其对应的密文串y。

选择密文攻击：分析者可以自己进行解密操作，可以选择任意一个密文串y，并可以获得其对应的明文串x。

对于古典密码分析，本章只讨论已知信息最少的唯密文攻击。其分析的常用思路为：尽可能地穷举遍历密钥空间$\mathcal{K}$，使用不同的密钥对密文进行解密，得到可能的明文，若明文长度较短，可以进行人工主观判断，找出可能性最大的明文即可。

若明文长度较长，可以使用词频分析法自动化分析，例如四字母分析法等。词频分析法的基本原理是使用统计学方法自动化判断明文文本的可读性，即与人为英文文本的相似性，最终选取可读性最强的可能明文。

四字母分析法即为一种度量随机文本与英语文本相似性的方法。将文本拆分为连续的“四元组”序列，例如，文本"$ATTACK$"中的四元组序列为 ATTA、TTAC 和 TACK。

要想通过文本的四元组序列来确定其与英语文本的相似度，首先需要知道英语文本中经常出现哪些四元组。为此，需要选取大量的可读英语文本，并统计其内各个四元组的出现次数，然后分别除以选取文本中四元组的总数，便得到了各个四元组的出现概率。

例如，若统计托尔斯泰的著作《战争与和平》中四元组的出现概率，去除书中的所有空格和标点符号后，总计约有 2,500,000 个四元组，经进一步分类整理，可以得到各个四元组的出现次数和概率对数

四元组	**出现次数**	log (P)
AAAA	1	-6.40018764963
QKPC	0	-9.40018764963
YOUR	1132	-3.34634122278
TION	4694	-2.72864456437
ATTA	359	-3.84509320105
……		

显然，部分四元组的出现频率要远高于其它四元组，这便可以表示可读文本中的一些共性，用于确定随机文本与英文文本的相似度。例如，如果文本中包含"$QPKC$"，它很可能不是明文文本，相反，如果它包含"$TION$"，则很可能可以确定为明文文本。其中，虽然"$QPKC$"出现次数为 0，但其概率对数并不是无限小，这是因为需要规定一个概率下限，否则会对后续计算产生不必要的影响。

获得英文文本的四元组统计数据后，便可以通过这些数据，计算随机文本是英语文本的概率了。按序提取出文本中的所有四元组后，将各个四元组在英文文本中的出现概率相乘，即为文本是英语文本的概率。

例如，文本"$ATTACK$"，其是英语文本的概率

$$P(ATTACK) = P(ATTA) \times P(TTAC) \times P(TACK)$$

$$P(ATTA) = \frac{count(ATTA)}{N}$$

但在实际计算中，多个概率浮点数相乘可能会导致精度下溢，因此一般对出现的概率取对数，按概率对数进行计算

$$log\big(P(ATTACK)\big) = log\big(P(ATTA)\big) + log\big(P(TTAC)\big) + log\big(P(TACK)\big)$$

这个概率对数被称为一段文本的“适应度”，即表示该文本与英文文本的相似度，数字越大表示它越有可能是英文，而数字越小则意味着它越不可能是英文。因此，在使用词频分析法

进行古典密码分析时，可以有意构造适应度最大的解密文本，其极有可能就是所求明文。

16.2 示例分析

如无特别说明，本篇章所有示例代码均在 Sagemath 环境下运行，Sagemath 是一个免费开源的数学计算系统。

下面让通过几个示例来进一步体会四字母分析法的原理和使用方法。

在实际应用中，由于统计量较大，一般通过程序自动化统计英文文本词频，并对统计后的数据进行预处理，以便快速计算指定文本的适应度

```
from math import log10

class ngram_score(object):
    def __init__(self,ngramfile,sep=' '):
        ''' 从 ngramfile 文件中读取四元组及其计数 '''
        self.ngrams = {}
        for line in open(ngramfile,"r"):
            key,count = line.split(sep)
            self.ngrams[key] = int(count)
        self.L = len(key)
        self.N = sum(self.ngrams.values())
        ''' 计算并缓存概率对数 '''
        ngkeys=list(self.ngrams.keys())
        for key in ngkeys:
            self.ngrams[key] = log10(float(self.ngrams[key])/self.N)
        self.floor = log10(0.01/self.N)

    def score(self,text):
        ''' 计算文本 text 与英文文本的相似度 '''
        score = 0
        ngrams = self.ngrams.__getitem__
        for i in range(len(text)-self.L+1):
            if text[i:i+self.L] in self.ngrams: score += ngrams(text[i:i+self.L])
            else: score += self.floor
        return score
```

其中，ngramfile 代表词频统计表文件名，格式为每行一个四元组及其出现次数，形如

```
TION 13168375
NTHE 11234972
THER 10218035
THAT 8980536
……
```

16.2.1 简单替换密码分析

题目要求：已知简单替换密码体制的密文序列，试求其对应明文

UNGLCKVVPGTLVDKBPNEWNLMGVMTTLTAZXKIMJMBBANTLCMOMVTNAAM
ILVTMCGTHMKQTLBMVCMXPIAMTLBMVGLTCKAUILEDMGPVLDHGOMIZWNLMGB
ZLGKSMAZBMKOMKTWNLMGBZKTLCKAMHMIMDMVGBZLXBLCSAZTBMMOMTVP
GMOMVKJLTQPXCBPNEJLBBLUILVDKJKZ

题目分析：依据 18.1.3 节中的基本原理，需要尽可能地覆盖密钥空间，选用疑似密钥尝试对上述密文进行解密，并通过四字母分析法对解密出的可能明文文本进行评估，得到最接近英语文本的可读明文。

依据 18.1.1 节**例 16-2** 中简单替换密码的定义，其密钥为一个随机置换π，一种比较简单的遍历思路为以英文字母序为初始序列，随后不断打乱该序列，把该序列作为置换密钥π。

解题代码：

```
import random
from ngram_score import ngram_score
# 参数初始化
ciphertext ='UNGLCKVVPGTLVDKBPNEWNLMGVMTTLTAZXKIMJMBBANTLCMOMVTNAAMILVTMCGTHMKQTLBMVCMXPIAMTLBMVGLTCKAUILEDMGPVLDHGOMIZWNLMGBZLGKSMAZBMKOMKTWNLMGBZKTLCKAMHMIMDMVGBZLXBLCSAZTBMMOMTVPGMOMVKJLTQPXCBPNEJLBBLUILVDKJKZ'
parentkey = list('ABCDEFGHIJKLMNOPQRSTUVWXYZ') # 置换密钥
key = {'A':'A'}
# 读取四元组统计信息
fitness = ngram_score('english_quadgrams.txt')
parentscore = -99e9
maxscore = -99e9

while 1:
    # 随机打乱置换密钥中的元素
    random.shuffle(parentkey)
    # 缓存密文:明文映射
    for i in range(len(parentkey)):
        key[parentkey[i]] = chr(ord('A')+i)
    # 解密
    decipher = ciphertext
    for i in range(len(decipher)):
        decipher = decipher[:i]+key[decipher[i]]+decipher[i+1:]
    # 计算相似度
    parentscore = fitness.score(decipher)
    # 更新最大相似度并输出当前密钥和明文
    if parentscore > maxscore:
        maxscore = parentscore
        print ('Currrent Key: ', parentkey)
        decipher = ciphertext
        for i in range(len(decipher)):
            decipher = decipher[:i]+key[decipher[i]]+decipher[i+1:]
        print ('Plaintext: ', decipher, maxscore)
```

其中，$ngram_score$为前述预处理代码文件$ngram_score.py$，$english_quadgrams.txt$是提前生成好的词频统计表文件。

上述示例代码的思路是比较简单、比较清晰的，但是其本质是在完全随机地选取置换π，这固然会导致破解效率极低，无法快速接近正确的密钥。可以尝试对其进行优化，在更新最大相似度$maxscore$前，可以把当前密钥作为父密钥，对当前密钥进行多次微调，例如只交换置换序列中的两个字母，围绕当前密钥产生一些关联度较高的其它可能明文，取其中文本适应度最高的$parentscore$作为代表，这样，使遍历密钥空间时的目标对象由点变为面，可以有效提高遍历效率。

优化代码：

```
count = 0
while count < 2000: # 最多尝试 2000 次
    a = random.randint(0,25)
    b = random.randint(0,25)
    # 随机交换父密钥中的两个元素生成子密钥，并用其进行解密
    parentkey[a],parentkey[b]= parentkey[b],parentkey[a]
    key[parentkey[a]],key[parentkey[b]] = key[parentkey[b]],key[parentkey[a]]
    decipher = ciphertext
    for i in range(len(decipher)):
        decipher = decipher[:i]+key[decipher[i]]+decipher[i+1:]
    score = fitness.score(decipher)
    # 此子密钥代替其对应的父密钥，提高明文适应度
    if score > parentscore:
```

```
        parentscore = score
        count=0
    else:
        # 还原
        parentkey[a],parentkey[b]=parentkey[b],parentkey[a]
        key[parentkey[a]],key[parentkey[b]]=key[parentkey[b]],key[parentkey[a]]
        count +=1
```

结合上述代码，可以在很短的时间（约 8 轮）内得到明文

```
BUTICANNOTSINGALOUDQUIETNESSISMYFAREWELLMUSICEVENSUMMERINS
ECTSHEAPSILENCEFORMESILENTISCAMBRIDGETONIGHTVERYQUIETLYITAKEMYL
EAVEASQUIETLYASICAMEHEREGENTLYIFLICKMYSLEEVESNOTEVENAWISPOFCLO
UDWILLIBRINGAWAY
```

16.2.2 维吉尼亚密码分析

题目要求：已知维吉尼亚密码体制的密文序列，且密钥长度不超过 8，试求其对应明文

```
YRAAHYHBIUWGRWBYCHCMHKXKUVRQNFSPWULNRMPQYHBMQDWKLNMBJ
CKUOEJENVLDYLPWCLDAYOUFQOXFAFVLCRMPVZQQQMNSDLCIPWPCHDLYJWKL
PKMZQRQQMTAAVDMLYJQVWYVCOHRUDMUEZCWOPJPVVJSVJEZFYOCMQWWLR
ETAHFDNHYZSRGVMLAHFWRIJKJVLRSNAWKZSPJXSKHXXFKIJDXHWAOIV
```

题目分析：同样采用四字母分析法的思路，尽可能地遍历密钥空间，逐个进行解密，选取与文本适应度最高的可疑明文。

由于密钥的长度未知，因此需要依次尝试1~8的密钥长度。同时，若密钥长度为 8 时，密钥空间大小高达$26^8 \approx 2 \times 10^{11}$，完全遍历的时间消耗过大。

但是可以采取与 18.2.1 节中相类似的优化思路，在指定的密钥长度下，把一个随机密钥作为父密钥，随后选取父密钥中的一个随机位置，依次从$'A'$到$'Z'$改变该位置上的值，作为新的子密钥，若该子密钥能够解密取得更高适应度的可能明文，则把该子密钥作为新的父密钥，否则该位置恢复原值；再次随机选取父密钥中的一个位置，重复前述过程一定轮次。

上述方法的本质依旧是在围绕各个父密钥向周边扩展，使遍历的点变为面，提高遍历的覆盖率和效率，最终选取出一个明文适应度最高的父密钥。

解题代码：

```
import random
from ngram_score import ngram_score
# 参数初始化
ciphertext
='YRAAHYHBIUWGRWBYCHCMHKXKUVRQNFSPWULNRMPQYHBMQDWKLNMBJCKUOEJENVLDYLPWCLDAYOUFQOXFAFVLCRMPV
ZQQQMNSDLCIPWPCHDLYJWKLPKMZQRQQMTAAVDMLYJQVWYVCOHRUDMUEZCWOPJPVVJSVJEZFYOCMQWWLRETAHFDNHYZS
RGVMLAHFWRIJKJVLRSNAWKZSPJXSKHXXFKIJDXHWAOIV'
# 读取四元组统计信息
fitness = ngram_score('english_quadgrams.txt')
maxscore = -99e9
# 大写字母在 Z26 上的减法
def sub(c,m):
    return chr((ord(c)-ord('A')-m)%26+ord('A'))

for k in range (1,9): # 密钥长度 k
    parentscore = -99e9
    key=list(range(k))
    j=0
    while j<10*k:
        pos=randint(0,k-1) # 选取随机位置
        j+=1
        for ch in range(26):
```

```
            temp = key[pos]
            key[pos] = ch
            decipher = ciphertext
            for i in range(len(decipher)): # 解密
                decipher = decipher[:i]+sub(decipher[i],key[i%k])+decipher[i+1:]
            score = fitness.score(decipher)
            # 此子密钥代替父密钥，提高明文适应度
            if score > parentscore:
                parentscore = score
            else:
                # 还原密钥
                key[pos]=temp
    # 输出该密钥和明文
    if parentscore > maxscore:
        maxscore = parentscore
        print ('Currrent Key: ',key)
        print ('Iteration total:', j)
        decipher = ciphertext
        for i in range(len(decipher)):
            decipher = decipher[:i]+sub(decipher[i],key[i%k])+decipher[i+1:]
        print ('Plaintext: ', decipher, maxscore)
        sys.stdout.flush()
```

可以很快取得密钥"$KEYWORD$"，以及对应明文

ONCETHEREWASATRUELOVEATMYHANDBUTIDIDNOTCHERISHITIDIDNOTREALIZEITUNTILITWASGONETHEREISNOTHINGMOREMISERABLETHANITIFGODCANGIVEMEACHANCETORESTARTIWILLTELLTHEGIRLILOVEYOUIFIHAVETOADDADEADLINETOTHELOVEIHOPEITWILLBETENTHOUSANDYEARS

16.2.3 希尔密码分析

题目要求：已知希尔密码体制的密文序列，且密钥矩阵为三维矩阵，试求其对应明文

RYLLAFFSOJJEYVSBYWGDEEKCKUISULIEFVXVZKHBXMVPHMIBQJZSEIXTMNUUIOHPGVFFVYTSUNUWSGLJTVPMXSGWMDJJEZRZIEEBHLTJFDFFXVJOCOJGNQJZVOUGMXHEQBCTVWZBHLGGSTRCSKUGDEIJMWYGJWCFSVVWZJALXZRSVYHAFTDDYJUXNCNBUBZXFFVYTSTGATRPTMWHQCCAMTIZPEMPDZDWRZRZIEEBHLKPINJR

题目分析：希尔密码的破解依旧与 18.2.1 节、18.2.2 节的思路相类似，通过随机产生父密钥，在各个父密钥周围扩展子密钥，选取可以产生相对更高适应度的子密钥作为新的父密钥，进行多次迭代，尽可能大地覆盖密钥空间。此处给出一种可行方法，遍历时每三轮为一组，每组的第一轮随机生成新的父密钥矩阵、最后一轮调换矩阵列顺序，每轮更新指定一列的值，在前述每一次更新操作后都用当前矩阵来进行解密操作，计算所得可能明文的适应度，选取较高者作为替换当前父密钥。

解题代码：

```
1. import random
2. from ngram_score import ngram_score
3. # 参数初始化
4. ciphertext
='RYLLAFFSOJJEYVSBYWGDEEKCKUISULIEFVXVZKHBXMVPHMIBQJZSEIXTMNUUIOHPGVFFVYTSUNUWSGLJTVPMXSGWM
DJJEZRZIEEBHLTJFDFFXVJOCOJGNQJZVOUGMXHEQBCTVWZBHLGGSTRCSKUGDEIJMWYGJWCFSVVWZJALXZRSVYHAFTDD
YJUXNCNBUBZXFFVYTSTGATRPTMWHQCCAMTIZPEMPDZDWRZRZIEEBHLKPINJR'
5. maxscore = -99e9
6. # 定义 Z26 运算的代数结构
7. R = Zmod(26)
8. # 定义 Z26 上的 3*3 矩阵
```

```
MR = MatrixSpace(R,3,3)
key = MR()
# 读取四元组统计信息
fitness = ngram_score('english_quadgrams.txt')
# 将密文转化为三维向量数组 vcode
vcode = [0]*(len(ciphertext)//3)
for i in range(len(ciphertext)//3):
    vcode[i]=vector([R(ord(ciphertext[3*i])-ord('A')),R(ord(ciphertext[3*i+1])-ord('A')),R(ord(ciphertext[3*i+2])-ord('A'))])
# 缓存 Z26 整数和字符之间的对应关系
z2chr = {}
for i in range(26):
    z2chr[R(i)]=chr(ord('A')+i)

#利用 3*3 的解密密钥 key 解密
def hill(ciphertext, key):
    cipher=''
    for i in range(len(ciphertext)//3):
        v = vcode[i]*key
        cipher = cipher+z2chr[v[0]]+z2chr[v[1]]+z2chr[v[2]]
    return cipher

for k in range (15):
    parentscore = -99e9
    pos=k%3
    # 每 3 轮更换随机父密钥
    if pos==0:
        for i in range(3):
            for j in range(3):
                key[i,j]=R(randint(0,25))
    # 尝试更新第 pos 列的值
    for item1 in range(26):
        for item2 in range(26):
            for item3 in range(26):
                if gcd([item1,item2,item3,26])!=1: # 不互质
                    continue
                temp1,temp2,temp3=key[0,pos],key[1,pos],key[2,pos]
                key[0,pos],key[1,pos],key[2,pos]=R(item1),R(item2),R(item3)
                decipher = hill(ciphertext, key)
                score = fitness.score(decipher)
                if score > parentscore: #此子密钥代替父密钥，提高明文适应度
                    parentscore = score
                else: # 还原密钥
                    key[0,pos],key[1,pos],key[2,pos]=temp1,temp2,temp3
    # 每 3 轮中的最后一轮尝试调换部分列
    if pos==2:
        for s in [[(0,1)],[(1,2)],[(0,2)],[(0,1),(0,2)],[(0,2),(0,1)]]:
            for t in s:
                key.swap_columns(t[0],t[1])
            decipher=hill(ciphertext, key)
            score = fitness.score(decipher)
            if score > parentscore: #此子密钥代替父密钥，提高明文适应度
                parentscore = score
            else: # 还原密钥
                for t in s[::-1]:
                    key.swap_columns(t[0],t[1])
    # 输出该密钥和明文
    if parentscore > maxscore:
        maxscore = parentscore
        print ('Currrent Key^-1: ', key)
        decipher = hill(ciphertext, key)
```

```
69.         print ('Plaintext: ', decipher, maxscore)
70.         sys.stdout.flush()
```

其中，$Zmod$函数用于定义一个整数域，$MatrixSpace$函数用于声明一个定义在指定数域下的矩阵空间，gcd为最大公约数计算函数，均为 Sagemath 环境提供的数学计算函数。

可以得到密钥k的逆矩阵，即解密密钥k^{-1}

$$\begin{bmatrix} 6 & 23 & 22 \\ 18 & 8 & 17 \\ 11 & 5 & 4 \end{bmatrix}$$

解密得到明文

FOURSCOREANDSEVENYEARSAGOOURFATHERSBROUGHTFORTHONTHISCONT
INENTANEWNATIONCONCEIVEDINLIBERTYANDDEDICATEDTOTHEPROPOSITIONTH
ATALLMENARECREATEDEQUALNOWWEAREENGAGEDINAGREATCIVILWARTESTIN
GWHETHERTHATNATIONORANYNATIONSOCONCEIVEDANDSODEDICATEDCANLON

16.3 举一反三

古典密码体制主要包含代换密码、置换密码和其它特殊密码，大多呈现一定的规律，因此即使是在已知信息最少的唯密文攻击场景下，也都可以较为轻易地进行密码分析，安全性很弱，现代密码体制中已经很少使用类似的思路。

在进行古典密码体制的分析时，一般可以从指定密码体制的密钥生成规律、加密密文特征出发，思考如何高效的穷举密钥、如何进一步提高当前密钥对应明文的适应度，推荐的思路是以随机的父密钥为中心，通过微调来扩展出子密钥，以覆盖父密钥周边的密钥空间，选取其中明文适应度最高的密钥作为新的父密钥继续扩展，范围性地覆盖密钥空间，可有效提高穷举效率和正确率。

在分析明文适应度时，理论上来讲，单字母频率分析、二元组分析、三元组分析也可以达到计算文本适应度的目的，但从实践检验的效果上来看，效率和正确率的表现均不如四元组分析法，同时若选取更多元素，例如五元组分析、六元组分析，几乎仅仅平添了效率消耗，并没有体现出更多优势，因此一般选取四元组进行词频分析。

但值得注意的是，对于部分特别构造出来的密文来说，仅依靠一般的词频分析法可能无法完成密文破解，因而也产生了一些针对特定古典密码体制的预分析方法，例如针对维吉尼亚密码分析的卡西斯基试验、重合指数法等，可以自行了解。

第17章 序列密码分析

17.1 基本原理

序列密码也称为流密码，它是对称密码算法的一种。序列密码具有实现简单、便于硬件实施、加解密处理速度快、没有或只有有限的错误传播等特点，因此在实际应用中，特别是专用或机密机构中保持着优势，典型的应用领域包括无线通信、外交通信。

17.1.1 序列密码

在前面研究的密码体制中，连续的明文元素是使用相同的密钥K来加密的，即密文串使用如下方法得到：

$$y = y_1 y_2 \cdots = e_K(x_1)e_K(x_2)\cdots$$

这种类型的密码体制通常称为分组密码。

另一种被广泛使用的密码体制称为序列密码，也称流密码，其基本思想是产生一个密钥流$z = z_1 z_2 \cdots$，然后使用它根据下述规则来加密明文串$x = x_1 x_2 \cdots$：

$$y = y_1 y_2 \cdots = e_{z_1}(x_1)e_{z_2}(x_2)\cdots$$

最简单的序列密码是其密钥流直接由初始密钥使用某种特定算法变换得来，密钥流和明文串是相互独立的。这种类型的序列密码称为“同步”序列密码，正式定义如下：

定义 17-1 同步序列密码是一个六元组$(\mathcal{P},\mathcal{C},\mathcal{K},\mathcal{L},\mathcal{E},\mathcal{D})$和一个函数g，并且满足如下条件：

1. $\mathcal{P}$是所有可能明文构成的有限集。
2. $\mathcal{C}$是所有可能密文构成的有限集。
3. 密钥空间$\mathcal{K}$为一有限集，由所有可能密钥构成。
4. $\mathcal{L}$是一个称之为密钥流字母表的有限集。
5. g是一个密钥流生成器。g使用密钥K作为输入，产生无限长的密钥流$z = z_1 z_2 \cdots$，这里$z_i \in \mathcal{P}$，$i \geq 1$。
6. 对任意的$z \in \mathcal{L}$，都有一个加密规则$e_z \in \mathcal{E}$和对应的解密规则$d_z \in \mathcal{D}$。并且对每个明文$x \in \mathcal{P}$，$e_z: \mathcal{P} \to \mathcal{C}$和$d_z: \mathcal{C} \to \mathcal{P}$是满足$d_z\big(e_z(x)\big) = x$的函数。

利用前文提到的维吉尼亚密码对同步序列密码的定义给出一个解释。假设m为维吉尼亚密码的密钥长度，定义$\mathcal{K} = (\mathbb{Z}_{26})^m$，$\mathcal{P} = \mathcal{C} = \mathcal{L} = \mathbb{Z}_{26}$；定义$e_z(x) = (x + z) mod\ 26$，$d_z(y) = (y - z) mod\ 26$。再定义密钥流$z_1 z_2 \cdots$如下：

$$z_i = \begin{cases} k_i & \text{若 } 1 \leq i \leq m \\ z_{i-m} & \text{若} i \geq m+1 \end{cases}$$

上式中$K = (k_1, k_2, \cdots, k_m)$，这样利用$K$可产生密钥流如下：

$$k_1 k_2 \cdots k_m k_1 k_2 \cdots k_m k_1 k_2 \cdots$$

注：分组密码可看作是序列密码的特殊情况，即对所有的$i \geq 1$，密钥流为一常数$z_i = K$。

如果对所有$i \geq 1$的整数有$z_{i+d} = z_i$，则称该序列密码是具有周期d的周期序列密码。如上面分析的密钥字长为m的维吉尼亚密码可看作是周期为m的序列密码。

序列密码通常以二元字符来表示，即$\mathcal{P} = \mathcal{C} = \mathcal{L} = \mathbb{Z}_2$，此时加密解密刚好都可看作模 2 的加法：

$$e_z(x) = (x + z) mod\ 2$$

和

$$d_z(y) = (y+z) mod\ 2$$

如果认为“0”代表布尔值为“假”，“1”代表布尔值为“真”，那么模 2 加法对应于异或运算。这样，加密和解密都可用硬件方式有效地实现。

下面给出一个产生同步密钥流的方法。假设从$(k_1, k_2, \cdots, k_n)$开始，并且$z_i = k_i$，$1 \le i \le m$。利用次数为m的线性递归关系来产生密钥流：

$$z_{i+m} = \sum_{j=0}^{m-1} c_j z_{i+j}\ mod\ 2$$

这里$i \ge 1$，$c_0, c_1, \cdots, c_{m-1} \in \mathbb{Z}_2$是确定的常数。

注：这个递归关系的次数为m，是因为每一个项都依赖于前面m个项；又因为z_{i+m}是前面的项的线性组合，故称其为线性的。注意，不失一般性，取$c_0 = 1$，否则递归关系的次数将为$m-1$。

这里密钥K由$2m$个值$k_1, k_2, \cdots, k_m, c_0, c_1, \cdots, c_{m-1}$组成。如果$(k_1, k_2, \cdots, k_m) = (0,0,\cdots,0)$，则生成的密钥流全为零，当然这种情况是需要避免的，否则明文将与密文相同。另外，如果常数$c_0, c_1, \cdots, c_{m-1}$选择适当的话，则任意非零初始向量$(k_1, k_2, \cdots, k_m)$都将产生周期为$2^m - 1$的密钥流。这种利用“短”的密钥来产生较长的密钥流的方法，正是设计之初所期望的。

例 17-1 设$m = 4$，密钥流按如下线性递归关系产生：

$$z_{i+4} = (z_i + z_{i+1}) mod\ 2\ \ i \ge 1$$

如果密钥流的初始向量不为零，则将获得周期为$2^4 - 1 = 15$的密钥流。例如，若初始向量为$(1,0,0,0)$，则可产生密钥流如下：

$$1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ \cdots$$

任何一个非零的初始向量都将产生具有相同周期的密钥流序列。

这种密钥流产生方法的另外一个诱人之处在于密钥流能使用线性反馈移位寄存器(LFSR)以硬件的方式来有效地实现。使用具有m个级的移位寄存器，向量$(k_1, k_2, \cdots, k_m)$用来初始化移位寄存器，在每一个时间单元，自动完成下列运算：

1. k_1抽出作为下一个密钥流比特
2. $k_2, k_3, \cdots, k_m$分别左移一个级
3. “新”的k_m值由下式“线性反馈”给出：

$$\sum_{j=0}^{m-1} c_j k_{j+1}$$

在任何一个给定的时间点，移位寄存器的m个级的内容是m个连续的密钥流元素，比如在时刻i时是$z_i, z_{i+1}, \cdots, z_{i+m-1}$，在时刻$i+1$时是$z_{i+1}, z_{i+2}, \cdots, z_{i+m}$。

可以看出，线性反馈是通过抽取寄存器的某些级的内容和计算模 2 加法来进行的，下图给出了这个过程的一个解释，其对应的 LFSR 将产生例 17-1 中的密钥流。

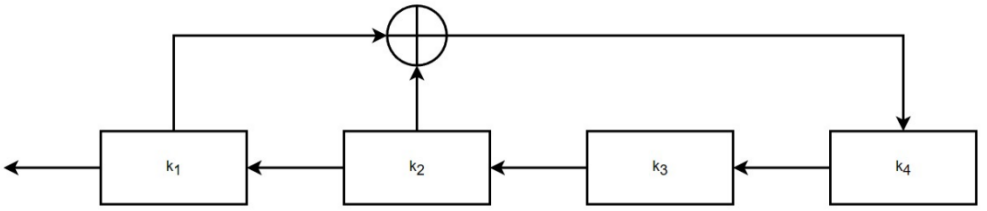


图 17-1 线性反馈移位寄存器

在序列密码中，还有这样一种情况，密钥流z_i的产生不但与密钥K有关，而且还与明文元素$(x_1, x_2, \cdots, x_{i-1})$或密文元素$(y_1, y_2, \cdots, y_{i-1})$有关，这种类型的序列密码称之为异步序列密码。下面给出一个来源于维吉尼亚密码的异步序列密码，称为自动密钥密码。称为“自动密钥”的原因是因为它使用明文来构造密钥流(除了最初始的“原始密钥”外)。当然，由于仅有 26 个可

能的密钥，自动密钥密码是不安全的。

密码体制 17-1 自动密钥密码$\mathcal{P},\mathcal{C},\mathcal{K},\mathcal{L},\mathcal{E},\mathcal{D}$

设$\mathcal{P}=\mathcal{C}=\mathcal{K}=\mathcal{L}=\mathbb{Z}_{26}$，$z_1=K$，定义$z_i=x_{i-1}$，$i\geq 2$。对任意的$0\leq z\leq 25$，$x,y\in\mathbb{Z}_{26}$，定义

$$e_z(x)=(x+z)mod\ 26$$

和

$$d_z(y)=(y-z)mod\ 26$$

例 17-2 假设$K=8$，明文为

$$rendezvous$$

首先将明文转换为整数序列：

17 4 13 3 4 25 21 14 20 18

相应的密钥流是：

8 17 4 13 3 4 25 21 14 20

将对应的元素相加，并通过模 26 约简：

25 21 17 16 7 3 20 9 8 12

字母形式的密文是：

$$ZVRQHDUJIM$$

解密时，Alice 首先转换密文字母为相应的数字串

25 21 17 16 7 3 20 9 8 12

然后计算

$$x_1=d_8(25)=(25-8)mod\ 26=17$$

再计算

$$x_2=d_{17}(21)=(21-17)mod\ 26=4$$

这样一直做下去，每次获得下一个明文字母，用它作为下一个密钥流元素。

17.1.2 LFSR 序列密码分析

在前面介绍的序列密码中，密文是明文和密钥流的模 2 加，即$y_i=(x_i+z_i)mod\ 2$。利用下列线性递归关系从初态$(z_1,z_2,\cdots,z_m)=(k_1,k_2,\cdots,k_m)$产生密钥流：

$$z_{m+i}=\sum_{j=0}^{m-1}c_jz_{i+j}\ mod\ 2\ \ i\geq 1$$

这里$c_0,c_1,\cdots,c_{m-1}\in\mathbb{Z}_2$。

因为这个密码体制中所有运算都是线性的，同前面的希尔密码一样，它容易受到已知明文攻击。假定 Oscar 有了明文串$x_1x_2\cdots x_n$和相应的密文串$y_1y_2\cdots y_n$，那么他能计算密钥流比特$z_i=(x_i+y_i)mod\ 2$，$1\leq i\leq n$。若 Oscar 再知道m的值，那么 Oscar 仅需要计算$c_0,c_1,\cdots,c_{m-1}$的值就能重构整个密钥流。换句话说，他只需要确定m个未知的值。

现在已知，对任何$i\geq 1$，有

$$z_{m+i}=\sum_{j=0}^{m-1}c_jz_{i+j}\ mod\ 2$$

它是m个未知数的线性方程。如果$n\geq 2m$，就有m个未知数的m个线性方程，利用它就可以解出这m个未知数。

m个线性方程可用矩阵形式表示为

$$(z_{m+1},z_{m+2},\cdots,z_{2m})=(c_0,c_1,\cdots,c_{m-1})\begin{pmatrix}z_1 & z_2 & \cdots & z_m\\ z_2 & z_3 & \cdots & z_{m+1}\\ \vdots & \vdots & \ddots & \vdots\\ z_m & z_{m+1} & \cdots & z_{2m-1}\end{pmatrix}$$

如果系数矩阵有逆(模 2)，则可解得

$$(c_0, c_1, \cdots, c_{m-1}) = (z_{m+1}, z_{m+2}, \cdots, z_{2m}) \begin{pmatrix} z_1 & z_2 & \cdots & z_m \\ z_2 & z_3 & \cdots & z_{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ z_m & z_{m+1} & \cdots & z_{2m-1} \end{pmatrix}^{-1}$$

事实上，如果m是产生密钥流的递归次数，那么这个矩阵一定是可逆的。

例 17-3 假设 Oscar 得到密文串

1 0 1 1 0 1 0 1 1 1 1 0 0 1 0

和相应的明文串

0 1 1 0 0 1 1 1 1 1 1 1 0 0 0

那么他能计算出密钥流比特

1 1 0 1 0 0 1 0 0 0 0 1 0 1 0

假定 Oscar 也知道密钥流是使用 5 级 LFSR 产生的，那么他利用前面 10 个比特就可以得到如下的矩阵等式

$$(0,1,0,0,0) = (c_0, c_1, c_2, c_3, c_4) \begin{pmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

容易通过检查两个矩阵的模 2 乘等于单位阵的方式来验证

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

这样就可求得

$$(c_0, c_1, c_2, c_3, c_4) = (0,1,0,0,0) \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix} = (1,0,0,1,0)$$

由此可知，用来产生密钥流的递归公式为

$$z_{i+5} = (z_i + z_{i+3}) mod\ 2$$

17.1.3 B-M 算法分析

根据密码学的需要，对线性反馈移位寄存器(LFSR)主要考虑下面两个问题：

（1）如何利用级数尽可能短的 LFSR 产生周期大、随机性能良好的序列，即固定级数时，什么样的寄存器序列周期最长。这是从密码生成角度考虑，用最小的代价产生尽可能好的、参与密码交换的序列。

（2）当已知一个长为N序列$\underline{a}$时，如何构造一个级数尽可能小的 LFSR 来产生它。这是从密码分析角度来考虑，要想用线性方法重构密钥序列所必须付出的最小代价。这个问题可通过 B-M 算法来解决。

设$\underline{a} = (a_0, a_1, \cdots, a_{N-1})$是$F_2$上的长度为$N$的序列，而$f(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_l x^l$是$F_2$上的多项式，$c_0 = 1$。

如果序列中的元素满足递推关系：

$$a_k = c_1 a_{k-1} + c_2 a_{k-2} + \cdots + c_l a_{k-l}, k = l, l+1, \cdots, N-1$$

则称$\langle f(x), l \rangle$产生二元序列$\underline{a}$。其中$\langle f(x), l \rangle$表示以$f(x)$为反馈多项式的l级线性移位寄存器。

如果$f(x)$是一个能产生$\underline{a}$并且级数最小的线性移位寄存器的反馈多项式，l是该移位寄存器

的级数，则称$\langle f(x), l\rangle$为序列$\underline{a}$的线性综合解。

线性移位寄存器的综合问题可表述为：给定一个N长二元序列$\underline{a}$，如何求出产生这一序列的最小级数的线性移位寄存器，即最短的线性移位寄存器？

注：（1）反馈多项式$f(x)$的次数$\leq l$。因为产生$\underline{a}$且级数最小的线性移位寄存器可能是退化的，在这种情况下$f(x)$的次数$\leq l$；并且此时$f(x)$中的$c_l = 0$，因此在反馈多项式$f(x)$中$c_0 = 1$，但不要求$c_l = 1$。（2）规定：0 级线性移位寄存器是以$f(x) = 1$为反馈多项式的线性移位寄存器，且n长$(n = 1,2,\cdots,N)$全零序列，仅由 0 级线性移位寄存器产生。事实上，以$f(x) = 1$为反馈多项式的递归关系式是：$a_k = 0$，$k = 0,1,\cdots,n-1$。因此，这一规定是合理的。（3）给定一个N长二元序列$\underline{a}$，求能产生$\underline{a}$并且级数最小的线性移位寄存器，就是求$\underline{a}$的线性综合解。利用 B-M 算法可以有效的求出。

B-M 算法具体要点如下：

用归纳法求出一系列线性移位寄存器：

$$\langle f_n(x), l_n\rangle \ \partial^0 f_n(x) < l_n, n = 1,2,\cdots,N$$

每一个$\langle f_n(x), l_n\rangle$都是产生序列$\underline{a}$的前$n$项的最短线性移位寄存器，在$\langle f_n(x), l_n\rangle$的基础上构造相应的$\langle f_{n+1}(x), l_{n+1}\rangle$，使得$\langle f_{n+1}(x), l_{n+1}\rangle$是产生给定序列前$n+1$项的最短移位寄存器，则最后得到的$\langle f_N(x), l_N\rangle$就是产生给定$N$长二元序列$\underline{a}$的最短的线性移位寄存器。

任意给定一个N长序列$\underline{a} = (a_0, a_1, \cdots, a_{N-1})$，按$n$归纳定义

$$\langle f_n(x), l_n\rangle \ n = 0,1,2,\cdots,N-1$$

1. 取初始值：$f_0(x) = 1$，$l_0 = 0$

2. 设$\langle f_0(x), l_0\rangle, \langle f_1(x), l_1\rangle, \cdots, \langle f_n(x), l_n\rangle$ $(0 \leq n \leq N)$均已求得，且$l_0 \leq l_1 \leq \cdots \leq l_n$

记：$f_n(x) = {c_0}^{(n)} + {c_1}^{(n)}x + \cdots + {c_{l_n}}^{(n)}x^{l_n}$，${c_0}^{(n)} = 1$，再计算：

$$d_n = {c_0}^{(n)}a_n + {c_1}^{(n)}a_{n-1} + \cdots + {c_{l_n}}^{(n)}a_{n-l_n}$$

称d_n为第n步差值。然后分两种情形讨论：

（1）若$d_n = 0$，则令：

$$f_{n+1}(x) = f_n(x), l_{n+1} = l_n$$

（2）若$d_n = 1$，则需区分以下两种情形：

①当：$l_0 = l_1 = \cdots = l_n = 0$时，

取：$f_{n+1}(x) = 1 + x^{n+1}, l_{n+1} = n+1$。

②当有$m(0 \leq m \leq n)$，使：$l_m < l_{m+1} = l_{m+2} = \cdots = l_n$。

设：$f_{n+1}(x) = f_n(x) + x^{n-m}f_m(x), l_{n+1} = \max\{l_n, n+1-l_n\}$

最后得到的$\langle f_N(x), l_N\rangle$便是产生序列$\underline{a}$的最短线性移位寄存器。

例 17-4 求产生周期为 7 的m序列(一个周期：0011101)的最短线性移位寄存器。

解：设$a_0a_1a_2a_3a_4a_5a_6 = 0011101$，首先取初值$f_0(x) = 1$，$l_0 = 0$，则由$a_0 = 0$得$d_0 = 1 \cdot a_0 = 0$从而$f_1(x) = 1$，$l_1 = 0$；同理由$a_1 = 0$得$d_1 = 1 \cdot a_1 = 0$从而$f_2(x) = 1$，$l_2 = 0$。由$a_2 = 1$得$d_2 = 1 \cdot a_2 = 1$，从而根据$l_0 = l_1 = l_2 = 0$知$f_3(x) = 1 + x^3$，$l_3 = 3$。

进一步的，计算d_3：$d_3 = 1 \cdot a_3 + 0 \cdot a_2 + 0 \cdot a_1 + 1 \cdot a_0 = 1$，因为$l_2 < l_3$，故$m = 2$，由此$f_4(x) = f_3(x) + x^{3-2}f_2(x) = 1 + x + x^3$，$l_4 = \max\{3, 3+1-3\} = 3$。

计算d_4：$d_4 = 1 \cdot a_4 + 1 \cdot a_3 + 0 \cdot a_2 + 1 \cdot a_1 = 0$，从而$f_5(x) = f_4(x) = 1 + x + x^3$，$l_5 = l_4 = 3$。

计算d_5：$d_5 = 1 \cdot a_5 + 1 \cdot a_4 + 0 \cdot a_3 + 1 \cdot a_2 = 0$，从而$f_6(x) = f_5(x) = 1 + x + x^3$，$l_6 = l_5 = 3$。

计算d_6：$d_6 = 1 \cdot a_6 + 1 \cdot a_5 + 0 \cdot a_4 + 1 \cdot a_3 = 0$，从而$f_7(x) = f_6(x) = 1 + x + x^3$，$l_7 = l_6 = 3$。

这表明，$\langle 1 + x + x^3, 3\rangle$即为产生所给序列一个周期的最短线性移位寄存器。

17.2 示例分析

17.2.1 B-M 算法求解序列最短 LFSR

题目要求：给定目标序列，欲构造一个级数尽可能小的 LFSR 来产生它，求解序列最短 LFSR。

题目示例：已知存在一个$GF(2)$上的 LFSR 序列，该序列的部分连续输出为$1101000101110011100101001011111$,定义该 LFSR 为$x_{n+1}=c_kx_n+c_{k-1}x_{n-1}+\cdots+c_0x_{n-k}$，试写出满足条件且级数最小的一个 LFSR 的参数$c_kc_{k-1}\cdots c_0$(格式：依次输出$c_k,c_{k-1},\cdots,c_0$，例如1001)。

题目分析：

记$\underline{a^{(n)}}=a_0a_1\cdots a_{N-1}$是长为$N$的有限长序列，$f_N(x)=x^n+c_1x^{n-1}+c_2x^{n-2}+\cdots+c_n\in F_q[x]$。如果序列$\underline{a^{(n)}}$满足$a_i+c_1a_{i-1}+c_2a_{2-1}+\cdots+c_na_{i-n}=0$，$n\le i\le N-1$，则称多项式$f_N(x)$可以生成$\underline{a^{(n)}}$，或称$f_N(x)$是$\underline{a}$的特征多项式。

类比于 BM 算法求解序列线性综合解，BM 算法可按照类似流程求解序列最短 LFSR 特征多项式，算法流程如下：

设$\underline{a}\in V(F_q)$，规定$f_0(x)=1$是生成序列$\underline{a}$的前 0 位的最低次多项式，假定$f_0(x),f_1(x),\cdots,f_n(x)$依次是生成$\underline{a}$的前 0 位，前 1 位，…前$n$位的最低次多项式，其次数分别为$l_0,l_1,\cdots,l_n$，下面求$f_{n+1}(x)$：

1. 若$f_n(x)$能生成$\underline{a}$的前$n+1$位，则取$f_{n+1}(x)=f_n(x)$，$l_{n+1}=l_n$
2. 若$f_n(x)$不能生成$\underline{a}$的前$n+1$位

（1）若$l_0=l_1=\cdots=l_n$，则取$f_{n+1}(x)=x^{n+1}-1$，$l_{n+1}=n+1$

（2）若$l_m<l_{m+1}=\cdots=l_n$，则

如果$m-l_m\ge n-l_n$，那么取

$$f_{n+1}(x)=f_n(x)-d_nd_m{}^{-1}x^{(m-l_m)-(n-l_n)}f_m(x)$$

如果$n-l_n>m-l_m$，那么取

$$f_{n+1}(x)=x^{(n-l_n)-(m-l_m)}f_n(x)-d_nd_m{}^{-1}f_m(x)$$

并且总有$l_{n+1}=\max\{l_n,n+1-l_n\}$

注意：此时d_n,d_m都一定不等于 0；在二元域上不等于 0 就一定为 1

在进行计算时，利用下面两个公式考察$f_n(x)$生成的$\widetilde{a_n}$是否与题目中给出的a_n相同，来判断$f_{n+1}(x)$是否与$f_n(x)$相等。

$$f_n(x)=x^{l_n}+c_1x^{l_n-1}+c_2x^{l_n-2}+\cdots+c_{l_n-1}x+c_{l_n}$$

$$\widetilde{a_n}=c_1a_{n-1}+c_2a_{n-2}+\cdots+c_{l_n-1}a_{n-(l_n-1)}+c_{l_n}a_{n-l_n}$$

解题代码：

```
# 判断 f_n[n]是否可以生成 a 的第 n+1 项
def Generable(n, f_n, ln, sequence):
    A = 0                                   # 理论计算值
    if ln == 0:
        A = 0
    else:
        for i in range(ln):                 # Ci = f_n[n][l[n]-i-1]
            A += (f_n[n][ln-i-1]) * (sequence[n-i-1])
    if (A % 2) == sequence[n]:
        return True
    else:
        return False
# BM 算法
```

```
def BM(sequence):
    f_n = [[0 for i in range(len(sequence))] for i in range(len(sequence)+1)]
    f_n[0][0] = 1                                          # 赋初值 f0=1
    l = [0 for i in range(len(sequence)+1)]                  # 记录每一轮最低次多项式次数
    for i in range(len(sequence)):
        if Generable(i, f_n, l[i], sequence):              # 算法流程 1
            for j in range(len(sequence)):
                f_n[i+1][j] = f_n[i][j]
                l[i+1] = l[i]
            continue
        else:                                             # 算法流程 2
            if l_all_equal(l):                             # 2(1)
                f_n[i+1][0] = 1
                f_n[i+1][i+1] = 1
                l[i+1] = i+1
            else:                                         # 2(2)
                j = i
                l[i + 1] = max(l[i], i + 1 - l[i])
                while j <= i:
                    if l[j] < l[i]:
                        m = j
                        break
                    j -= 1
                if (m-l[m] >= i-l[i]):                     # 2(2)(1)
                    for k in range(i-l[i]-m+l[m]+1):
                        f_n[i+1][k] = (f_n[i][k]) % 2
                    for k in range(i-l[i]-m+l[m], l[i+1]+1):
                        f_n[i+1][k] = (f_n[i][k] + f_n[m]
                                       [k-(m-l[m]-i+l[i])]) % 2
                else:                                     # 2(2)(2)
                    for k in range(i-l[i]-m+l[m]+1):
                        f_n[i+1][k] = (f_n[m][k]) % 2
                    for k in range(i-l[i]-m+l[m], l[i+1]+1):
                        f_n[i+1][k] = (f_n[i][k-(i-l[i]-m+l[m])
                                              ] + f_n[m][k]) % 2
    return f_n, l
# 判断 l 列表中的数字是否全部相同
def l_all_equal(l):
    for i in range(len(l) - 1):
        if l[i] != l[i + 1]:
            return False
    return True
# 还原多项式字符串并返回
def print_f(f_n, i):
    result = ''
    j = l[i]
    while j >= 0:
        if f_n[i][j] != 0:
            if j == 0:
                result += '1+'
            else:
                result += 'x^' + str(j) + '+'
        j -= 1
    return result[0:-1]
# 将字符串转为 int 形列表
def Seq_to_list(sequence):
    result = []
    for i in sequence:
        result.append(int(i))
    return result
seq = ['11010001011100111001010010111 1']
for S in seq:
```

```
    f_n, l = BM(Seq_to_list(S))
    print('输入序列：' + S)
    print('最短 LFSR 的特征多项式：' + print_f(f_n, len(S)))
    print('最低级数：' + str(l[len(S)]))
    print(" ")

```

解题结果：序列11010001011100111001010010111 1的最短 LFSR 特征多项式为x^8+x+1，由特征多项式定义可推导出 LFSR 递推公式为$x_{n+1}=x_{n-6}+x_{n-7}\ mod\ 2$，$k=7$，故$c_kc_{k-1}\cdots c_0=00000011$。

17.2.2 streamgame1 解决 LFSR 问题

题目要求：已知反馈函数，输出序列，求逆推出初始状态。题目所涉及 key 文件和 streamgame1.py 文件内容如下：

key 文件如下：（十六进制形式）

55 38 F7 42 C1 0D B2 C7 ED E0 24 3A

streamgame1.py 如下：

```
from flag import flag
assert flag.startswith("flag{")
# 作用：判断字符串是否以指定字符或子字符串开头 flag{
assert flag.endswith("}")
# 作用：判断字符串是否以指定字符或子字符串结尾}，flag{}，6 个字节
assert len(flag) == 25
# flag 的长度为 25 字节，25-6=19 个字节
# 3<<2 可以这么算，bin(3)=0b11 向左移动 2 位变成 1100，0b1100=12(十进制)
def lfsr(R, mask):
    # 将 R 向左移动 1 位，bin(0xffffff)='0b111111111111111111111111'=0xffffff 的二进制补码
    output = (R << 1) & 0xffffff
    i = (R & mask) & 0xffffff  # 按位与运算符&：参与运算的两个值,如果两个相应位都为 1,则该位的结果为 1,否则为 0
    lastbit = 0
    while i != 0:
        lastbit ^= (i & 1)  # 按位异或运算符：当两对应的  二进位相异时，结果为 1
        i = i >> 1
    output ^= lastbit
    return (output, lastbit)
R = int(flag[5:-1], 2)
mask = 0b10100011000100011100
f = open("key", "ab")  # 以二进制追加模式打开
for i in range(12):
    tmp = 0
    for j in range(8):
        (R, out) = lfsr(R, mask)
        tmp = (tmp << 1) ^ out  # 按位异或运算符：当两对应的二进位相异时，结果为 1
    f.write(chr(tmp))  # chr() 用一个范围在 range（256）内的（就是 0～255）整数作参数，返回一个对应的字符。
f.close()

```

题目分析：

通过对加密脚本 streamgame1.py 的理解，可得本题的 LFSR 模型：

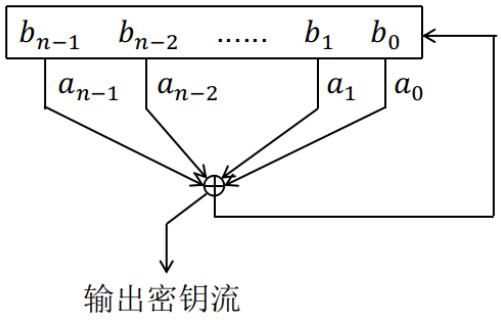


图 17-2 LFSR 模型

其中$a_{n-1}, a_{n-2}, \cdots, a_0$为程序中 mask 的二进制位，当$a_i = 1$时，将$b_i$输入异或运算，否则$b_i$不输入异或运算；根据模型可以得到如下等式：

$$\begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_{n-1} \\ k_n \end{pmatrix} = \begin{pmatrix} b_{n-1} & b_{n-2} & \cdots & b_1 & b_0 \\ b_{n-2} & b_{n-3} & \cdots & b_0 & k_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ b_1 & b_0 & \cdots & k_{n-3} & k_{n-2} \\ b_0 & k_1 & \cdots & k_{n-2} & k_{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix}$$

其中的加法为异或，因为$a_{n-1} = 1$，将上式重写如下：

$$\begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_{n-1} \\ k_n \end{pmatrix} = \begin{pmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_1 \\ b_0 \end{pmatrix} \oplus \begin{pmatrix} b_{n-2} & \cdots & b_1 & b_0 \\ b_{n-3} & \cdots & b_0 & k_1 \\ \vdots & \ddots & \vdots & \vdots \\ b_0 & \cdots & k_{n-3} & k_{n-2} \\ k_1 & \cdots & k_{n-2} & k_{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix}$$

由异或性质：

$$\begin{pmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_1 \\ b_0 \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_{n-1} \\ k_n \end{pmatrix} \oplus \begin{pmatrix} b_{n-2} & \cdots & b_1 & b_0 \\ b_{n-3} & \cdots & b_0 & k_1 \\ \vdots & \ddots & \vdots & \vdots \\ b_0 & \cdots & k_{n-3} & k_{n-2} \\ k_1 & \cdots & k_{n-2} & k_{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix}$$

再将等式“还原”：

$$\begin{pmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_1 \\ b_0 \end{pmatrix} = \begin{pmatrix} k_1 & b_{n-2} & \cdots & b_1 & b_0 \\ k_2 & b_{n-3} & \cdots & b_0 & k_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ k_{n-1} & b_0 & \cdots & k_{n-3} & k_{n-2} \\ k_n & k_1 & \cdots & k_{n-2} & k_{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix}$$

计算的顺序由下至上，即可解出初始状态的所有比特位。

解题代码：

```
from gmpy2 import c_div
def lfsr(R, mask):
    output = (R << 1) & 0xffffff
    i = (R & mask) & 0xffffff
    lastbit = 0
    while i != 0:
        lastbit ^= (i & 1)
        i = i >> 1
    output ^= lastbit
    return (output, lastbit)
def cal(s, mask):
    lm = len(bin(mask))-2
    R = int(s[-1:]+s[:-1], 2)
    ss = ''
```

```
    for j in range(lm, 0, -1):
        (_, tk) = lfsr(R, mask)
        ss = str(tk)+ss
        R = int(s[j-2]+str(tk)+bin(R)[2:].rjust(lm, '0')[1:-1], 2)
    return ss
def solve():
    mask = 0b10100011000100011100
    lm = len(bin(mask))-2
    with open('key', 'rb') as f:
        stream = f.read(c_div(lm, 8))
    s = ''.join([bin(256+ord(it))[3:] for it in stream])
    flag = 'flag{'+cal(s[:lm], mask)+'}'
    return flag
print (solve())

```

解题结果：flag{1110101100001101011}。

17.2.3 oldstreamgame 解决 LFSR 问题

题目要求：已知反馈函数，输出序列，求逆推出初始状态。题目所涉及 key 文件和 oldstreamgame.py 文件内容如下：

key 文件如下：（十六进制形式）

20FDEEF8A4C9F4083F331DA8238AE5ED083DF0CB0E7A83355696345DF44D7C186C1F459BCE135F1DB6C76775D5DCBAB7A783E48A203C19CA25C22F60AE62B37DE8E40578E3A7787EB429730D95C9E1944288EB3E2E747D8216A4785507A137B413CD690C

oldstreamgame.py 如下：

```
flag = "flag{xxxxxxxxxxxxxxx}"
assert flag.startswith("flag{")
assert flag.endswith("}")
assert len(flag)==14
def lfsr(R,mask):
    output = (R << 1) & 0xffffffff
    i=(R&mask)&0xffffffff
    lastbit=0
    while i!=0:
        lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)
R=int(flag[5:-1],16)
mask = 0b10100010000001000000100010010100
f=open("key","w")
for i in range(100):
    tmp=0
    for j in range(8):
        (R,out)=lfsr(R,mask)
        tmp=(tmp << 1)^out
    f.write(chr(tmp))
f.close()

```

题目分析：

程序输出的第 32 个比特是由程序输出的前 31 个比特和初始种子的第 1 个比特来决定的，因此可以知道初始种子的第一个比特，进而可以知道初始种子的第 2 个比特，依次类推。

解题代码：

```
mask = 0b10100010000001000000100010010100
b = ''
N = 32
```

```
with open('key', 'rb') as f:
    b = f.read()
key = ''
for i in range(N // 8):
    t = ord(b[i])
    for j in range(7, -1, -1):
        key += str(t >> j & 1)
idx = 0
ans = ""
key = key[31] + key[:32]
while idx < 32:
    tmp = 0
    for i in range(32):
        if mask >> i & 1:
            tmp ^= int(key[31 - i])
    ans = str(tmp) + ans
    idx += 1
    key = key[31] + str(tmp) + key[1:31]
num = int(ans, 2)
print (hex(num))

```

解题结果：flag{926201d7}。

17.2.4 streamgame3 解决 NLFSR 问题

题目要求：已知多个反馈函数，输出序列，求逆推出初始状态。题目所涉及 output 文件和 streamgame3.py 文件内容如下：

output 输出文件内容较多，不做展示，具体可参考 2018 强网杯 streamgame3 试题。

streamgame3.py 如下：

```
from flag import flag
assert flag.startswith("flag{")
assert flag.endswith("}")
assert len(flag)==24
def lfsr(R,mask):
    output = (R << 1) & 0xffffff
    i=(R&mask)&0xffffff
    lastbit=0
    while i!=0:
        lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)
def single_round(R1,R1_mask,R2,R2_mask,R3,R3_mask):
    (R1_NEW,x1)=lfsr(R1,R1_mask)
    (R2_NEW,x2)=lfsr(R2,R2_mask)
    (R3_NEW,x3)=lfsr(R3,R3_mask)
    return (R1_NEW,R2_NEW,R3_NEW,(x1*x2)^((x2^1)*x3))
R1=int(flag[5:11],16)
R2=int(flag[11:17],16)
R3=int(flag[17:23],16)
assert len(bin(R1)[2:])==17
assert len(bin(R2)[2:])==19
assert len(bin(R3)[2:])==21
R1_mask=0x10020
R2_mask=0x4100c
R3_mask=0x100002
for fi in range(1024):
    print fi
    tmp1mb=""
    for i in range(1024):
```

```
        tmp1kb=""
        for j in range(1024):
            tmp=0
            for k in range(8):
                (R1,R2,R3,out)=single_round(R1,R1_mask,R2,R2_mask,R3,R3_mask)
                tmp = (tmp << 1) ^ out
            tmp1kb+=chr(tmp)
        tmp1mb+=tmp1kb
    f = open("./output/" + str(fi), "ab")
    f.write(tmp1mb)
    f.close()
```

题目分析：

为了使得密钥流输出的序列尽可能复杂，可能会使用非线性反馈移位寄存器，常见的有三种：（1）非线性组合生成器，对多个 LFSR 的输出使用一个非线性组合函数；（2）非线性滤波生成器，对一个 LFSR 的内容使用一个非线性组合函数；（3）钟控生成器，使用一个（或多个）LFSR 的输出来控制另一个（或多个）LFSR 的时钟。

本题为非线性组合生成器，可统计在三个 LFSR 输出不同的情况下，最后生成器的输出，如下：

x_1	x_2	x_3	$F(x_1,x_2,x_3)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

可以发现，该非线性组合生成器输出，与x_1相同的概率为 0.75，与x_2相同的概率为 0.5，与x_3相同的概率为 0.75。这说明输出与x_1和x_3的关联性非常大。因此，可以暴力去枚举x_1和x_3对应对的 LFSR 的输出判断其与非线性组合生成器输出相等的个数，如果大约在 75%的话，就可以认为是正确的。x_2可直接暴力枚举。

解题代码：

```
def lfsr(R, mask):
    output = (R << 1) & 0xffffff
    i = (R & mask) & 0xffffff
    lastbit = 0
    while i != 0:
        lastbit ^= (i & 1)
        i = i >> 1
    output ^= lastbit
    return (output, lastbit)
def single_round(R1, R1_mask, R2, R2_mask, R3, R3_mask):
    (R1_NEW, x1) = lfsr(R1, R1_mask)
    (R2_NEW, x2) = lfsr(R2, R2_mask)
    (R3_NEW, x3) = lfsr(R3, R3_mask)
    return (R1_NEW, R2_NEW, R3_NEW, (x1 * x2) ^ ((x2 ^ 1) * x3))
R1_mask = 0x10020
R2_mask = 0x4100c
R3_mask = 0x100002
n3 = 21
n2 = 19
n1 = 17
def guess(beg, end, num, mask):
    ansn = range(beg, end)
```

```
    data = open('./output/0').read(num)
    data = ''.join(bin(256 + ord(c))[3:] for c in data)
    now = 0
    res = 0
    for i in ansn:
        r = i
        cnt = 0
        for j in range(num * 8):
            r, lastbit = lfsr(r, mask)
            lastbit = str(lastbit)
            cnt += (lastbit == data[j])
        if cnt > now:
            now = cnt
            res = i
            print (now, res)
    return res
def bruteforce2(x, z):
    data = open('./output/0').read(50)
    data = ''.join(bin(256 + ord(c))[3:] for c in data)
    for y in range(pow(2, n2 - 1), pow(2, n2)):
        R1, R2, R3 = x, y, z
        flag = True
        for i in range(len(data)):
            (R1, R2, R3,
             out) = single_round(R1, R1_mask, R2, R2_mask, R3, R3_mask)
            if str(out) != data[i]:
                flag = False
                break
        if y % 10000 == 0:
            print ('now: ', x, y, z)
        if flag:
            print ('ans: ', hex(x)[2:], hex(y)[2:], hex(z)[2:])
            break
R1 = guess(pow(2, n1 - 1), pow(2, n1), 40, R1_mask)
print (R1)
R3 = guess(pow(2, n3 - 1), pow(2, n3), 40, R3_mask)
print (R3)
R1 = 113099
R3 = 1487603
bruteforce2(R1, R3)

```

解题结果：暴力破解结果为 1b9cb 5979c 16b2f3，flag 为 flag{01b9cb05979c16b2f3}。

17.3 举一反三

序列密码，也称流密码，是对称密码的一种。其与分组密码的显著区别就在于其加密变换的对象比分组的块小，一般是比特。序列密码体制关键就在于其产生密钥序列的方法，也就是密钥序列产生器应具有良好的随机性，让密钥序列不可预测。序列密码分为同步序列密码和自（异）同步序列密码，前者密钥序列独立于明文序列和密文序列，而后者并不独立。

序列密码的密钥序列产生器，其控制状态序列的部分一般利用线性反馈移位寄存器进行实现，一般要求最长周期或m序列产生器实现，这样可为实际应用中更为复杂的非线性组合部分提供统计性能良好的序列。

线性反馈移位寄存器 LFSR 的反馈函数，是简单地对移位寄存器中的某些位进行异或，并将异或的结果填充到 LFSR 的最左端。针对单一的 LFSR，当 LFSR 猜测者已知 LFSR 级数m和至少两倍级数长度的连续明密文对后，便可计算出密钥序列并依照 LFSR 计算原理构造具有m个未知数的m个线性方程，按照矩阵方式求解出密钥序列的递推参数c_i，进而求得 LFSR 的反

馈公式。

在密码学中，针对于已知序列求解如何构造级数尽可能小的 LFSR 来产生它的问题，可利用 B-M 算法进行求解。B-M 算法引入反馈多项式、线性综合解的概念，通过归纳计算的方式推算序列，构建满足序列前i位的生成，直至序列被全部生成，获得级数最小的 LFSR 结果，其结果甚至可以预测序列后续的内容输出。

在一些密码学相关竞赛中，也常有以序列密码或 LFSR 为核心的试题，一般试题目标为已知反馈函数、输出序列，求逆推出初始状态。由于 LFSR 的反馈函数已经给出，针对其计算流程推出还原输入序列的方法是可行的，因为不同 LFSR 的处理过程不同，推算过程也不尽相同。针对较为复杂的情况，如多个 LFSR 的组合运算，往往可采用在 LFSR 计算推算的基础上结合暴力破解、猜测的方法来进行解决。

第18章 大整数分解

18.1 基本原理

在非对称密码体制中，加密算法的安全性基本依赖于特定的难解问题，该问题令使用者可以轻易地通过公钥k完成加密操作E_k，却难以完成解密操作D_k，解密则必须依赖私钥。

例如知名的 RSA 加密算法，RSA 首先随机选择两个不同的大素数p和q，计算$N = p \times q$，再选择一个与$\varphi(\mathrm{N}) = (p-1)(q-1)$互素的小整数$e$，并求得$e$关于$\varphi(\mathrm{N})$的逆元$d$，即

$$ed \equiv 1 \left(mod\ \varphi(N)\right)$$

此时，(N, e)为公钥，(N, d)为私钥，

$$E_{(N,e)}(x) = x^e (mod\ N)$$
$$D_{(N,d)}(y) = y^d (mod\ N)$$

攻击者若想通过公开的公钥计算得到私钥，就首先需要得知p和q的取值，才能够计算出$\varphi(N)$，进而求e关于$\varphi(\mathrm{N})$的逆元。

可见，RSA 的安全性便依赖于大整数分解问题，若可以将大整数N分解为素数乘积$p \times q$，便可以仅通过公钥直接对密文进行解密。在本章中，将对大整数分解问题展开研究。

18.1.1 素性检测

在研究大整数分解的方法之前，首先需要知道如何判断一个整数是否为素数，即素性检测，这是后续研究的基础。素性检测方法可以分为确定性素数检测和概率性素数检测：确定性素数检测可以百分之百地确定一个大整数是否为素数，但检测的效率极低，学界暂无足够高效的确定性素数检测方法；概率性素数检测则与之相反，仅以一定的高概率保证被检测数的素性，并不能确保被检测数一定符合检测结果，但检测效率要远高于确定性素数检测，巨大的性能差距使得概率性素数检测要比确定性素数检测更为常用。

确定性素数检测：试除法。依据素数的定义，只含有 1 和该数字本身两个因数，即不能被除了 1 和它本身以外的其它整数整除。

那么，对于$\forall N \in \mathbb{Z}^+$，只需要逐一验证$N$能否被$2 \sim N-1$中的任意一个整数整除即可，若无法被前述任何整数整除，则N是素数；否则不是。

同时，依据整除相关定理

$\forall n \in \mathbb{Z}^+$, 若$n = xy, x \le y$则$x \le \sqrt{n}$ $\Rightarrow$ 如果对$\forall i \le \sqrt{n}$，$i \in \mathbb{Z}^+$，满足$i \nmid n$，则n为素数

因此，在实际应用中，只需验证至$\sqrt{N}$即可。但即便如此，试除法也需要进行至多$\sqrt{N}$次的试除运算，对于 RSA 中至少 1024 比特的密钥长度来说，这一时间消耗是无法接收的。

概率性素数检测：Miller Rabin 法。Miller Rabin 检测法的基本思路是采用概率性正确的快速测试方法，对一个大整数进行多次测试，逐次降低误判率，直至使得误判率降低至可接受的范围，即可作为结论。

Miller Rabin 检测法选取的快速检测方法是基于费马小定理和二次探测定理的。

费马小定理：若p是素数，a为正整数，且a和p互质，则：$a^{p-1} \equiv 1\ (mod\ p)$。

逆命题 18-1：若正整数p满足$a^{p-1} \equiv 1\ (mod\ p)$，则$p$为素数。

此命题固然是一个假命题，但从统计学的角度上来说，依旧可以通过它来进行较大正确概率下的素性判断。

注意，不能仅通过**命题 18-1** 来进行素性测试，例如，该命题是不适用于大部分的 Carmichael 数的，这使得正整数n在特意构造的情况下，该测试并不适用，因此需要进一步引入二次探测定理。

二次探测定理：若p是素数，$p \neq 2$，则关于x的同余方程$x^2 \equiv 1 \ (mod \ p)$，在$[1,p)$上的解仅有$x = 1$及$x = p - 1$。

当正整数n为大于 2 的偶数时，必然为合数，在此不作讨论。因此可以构造出奇数m有

$$n - 1 = 2^k \times m$$

则**命题 18-1** 可以进一步写作

$$a^{n-1} = a^{2^k \times m} \equiv 1 \ (mod \ n)$$

依据二次探测定理，假设n是素数，则此时有

$$a^{2^{k-1} \times m} = 1 \text{ 或 } n - 1 \ \Rightarrow \ a^{2^{k-1} \times m} \equiv 1 \ (mod \ n)$$

$$\Rightarrow \ a^{2^{k-2} \times m} = 1 \text{ 或 } n - 1 \ \Rightarrow \ a^{2^{k-2} \times m} \equiv 1 \ (mod \ n)$$

$$\Rightarrow \cdots$$

$$\Rightarrow \ a^{2^{k-i} \times m} = 1 \text{ 或 } n - 1 \ \Rightarrow \ a^{2^{k-i} \times m} \equiv 1 \ (mod \ n)$$

直至$a^m \equiv 1 \ (mod \ n)$。在变量i递增的过程中，若正整数n可以连续多次满足上述等式，那就可以大概率地确定n为素数。统计结果表明，即使在考虑到特殊构造正整数n的情况下，前述每次判断的错误率也大概为0.25，i次运算后的错误率大概是0.25^i，很快便可以降低至一个可以接受的错误概率。

18.1.2 传统大整数分解算法

***Pollard P − 1*算法。**对于大整数n的素因子p，$p - 1$一定是一个合数，可以被分解为

$$p - 1 = b_1^{k_1} \cdot b_2^{k_2} \cdot \ldots \cdot b_m^{k_m}$$

其中，b_i均为素因子，可以取得一个最小的正整数$B \leq \max(b_i^{k_i})$，使$p - 1 \mid B!$，若此正整数B较小，则称$p - 1$是B平滑的。此时，可以使用$Pollard \ P - 1$算法对大整数n进行快速因数分解。

依据费马小定理，对于$\forall a \in \mathbb{Z}p$，有

$$a^{B!} \equiv a^{p-1} \equiv 1 \ (mod \ p)$$

即

$$p \mid a^{B!} - 1$$

则有

$$p \mid \gcd(a^{B!} - 1, n)$$

根据以上结论，可以选取固定的a，在实际应用中一般取 2，随后枚举B的取值，不断计算$a^{B!} - 1$与n的最大公因数d，一旦出现d满足$1 < d < n$，则d便是n的一个因子，推广至 RSA 下的公钥破解，即为$p = d$。

算法执行流程：

```
1. a=2
2. for i in range(2, B+1):
3.     a = power_mod(a, i, n)
4.     d = gcd(a-1,n)
5.     if 1<d and d<n:
6.         print(d)
7.         break
```

例 18-1 在尝试分解正整数15770708441时，该正整数的实际分解结果为

$$15770708441 = 135979 \times 115979$$

其中，$135979 - 1 = 135978 = 2 \times 3 \times 131 \times 173$，即$p - 1$是173平滑的，通过指定参数$B \geq 173$，即可使用$Pollard \ P - 1$算法在 173 次运算内计算出15770708441的分解。

显然，$Pollard\ P-1$算法只适用于$p-1$是平滑数，即存在较小素因子的情况，因此，现代RSA 密钥生成器一般会保证$\frac{p-1}{2}$也是素数，使$p-1$存在足够大的素因子，以抵御$Pollard\ P-1$算法攻击。

$William\ P+1$算法。$Pollard\ P-1$算法是针对$p-1$是平滑数的情况进行分析，而$William\ P+1$算法则是针对$p+1$是平滑数的情况，即可以将$p+1$分解为小素数的乘积

$$p-1=b_1^{k_1}\cdot b_2^{k_2}\cdot\ldots\cdot b_m^{k_m}$$

同样的，b_i均为素因子，可以取得一个最小的正整数$B\leq\max(b_i^{k_i})$，使$p+1\mid B!$。

选取一个大于 2 的整数 A 构建卢卡斯序列

$$V_0=2,V_1=A,V_i=A\cdot V_{i-1}-V_{i-2}$$

对于大整数n及它的一个素因子p，有

$$p\mid\gcd(V_M-2,n)$$

其中，要求M为$p-\left(\frac{A^2-4}{p}\right)$的倍数，$\left(\frac{A^2-4}{p}\right)$代表$\frac{A^2-4}{p}$的雅可比符号。

而只有当$\left(\frac{A^2-4}{p}\right)$为$-1$时，才会体现出$p+1$是平滑数所带来的计算优势，此时$M$的取值为$B!$，否则该算法将退化为$Pollard\ P-1$算法的慢速版本。

上述卢卡斯序列可以使用此流程快速计算：

```
def Lucas(A, M, n):
    v1, v2 = A, (A**2 - 2) % n
    for bit in bin(M)[3:]:
        if bit == "0":
            v1, v2 = (v1**2 - 2) % n, (v1*v2 - A) % n
        else:
            v1, v2 = (v1*v2 - A) % n, (v2**2 - 2) % n
    return v1

V_M=Lucas(A, M, n)
```

其中，$bin(M)$用于取M自最高非零位至最右位的二进制序列。

借助上述卢卡斯序列快速计算算法，$William\ P+1$算法执行流程为：

```
v=A
for i in range(2, B+1):
    v = Lucas(v,i,n)
    d = gcd(v-2,n)
    if 1<d and d<n:
        print(d)
        break
```

$Pollard\ \rho$算法。对于大整数n的素因子p，构造数对(x,x')，其中$x\neq x'$，满足

$$x\equiv x'(mod\ p)$$

随后计算

$$\mathrm{d}=\gcd(x-x',n)$$

若d满足$1<d<n$，则d便是n的一个因子，推广至 RSA 下的公钥破解，即为$p=d$。

为了能够快速构建出关于p同余的(x,x')，采用 Floyd 判环算法，首先构造递推序列

$$x_{i+1}=f(x_i)=({x_i}^2+c)\ mod\ n$$

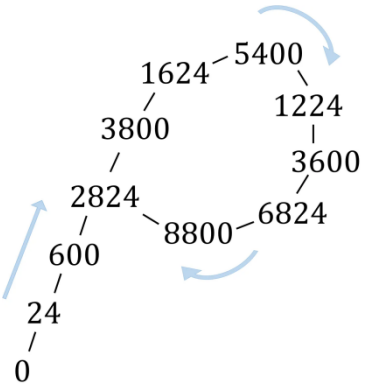
$$令\ x=x_i,x'=x_{2i}$$

也就是说，x在以一半的递推速度重新经过x'的递推路径，由于运算始终是在$\mathbb{Z}_n$域内进行，这使得x与x'终会发生碰撞，即二者关于n的一个因子p同余，达到目的。

例 18-2 若指定$x_0 = 0$、$c = 24$，分解整数$n = 9400$，依据上述递推式，可以得到

$$x_1 = ({x_0}^2 + 24)\ mod\ 9400 = 24$$
$$x_2 = ({x_1}^2 + 24)\ mod\ 9400 = 600$$
$$x_3 = ({x_2}^2 + 24)\ mod\ 9400 = 2824$$
$$\cdots\cdots$$

经过多次迭代计算后，可以发现$x_3 \equiv x_{11} \equiv 2824\ mod\ 9400$，即发生了碰撞，而接下来的迭代将陷入一个循环，使得整个迭代路径的形状形似字母ρ，这也是为何该算法被称为$Pollard\ \rho$算法。



通过不断尝试计算$\gcd(x_{2i} - x_i, n)$，最终可以分解得到$n = 2^3 \times 5^2 \times 47$。

算法执行流程：

```
def f(x, n, c):
    return (x**2 + c) % n

x1 = 2
x2 = f(x1, n, C)
d = gcd(x1-x2, n)
while d == 1:
    x1 = f(x1, n, C)
    x2 = f(f(x2,n,C), n, C)
    d = gcd(x1-x2, n)
    if d == n:
        print('fail')
print(d)
```

该算法同样存在较为显然的缺陷，其只能适用于存在较小素因子的大整数n，否则时间消耗将极大，因此，RSA 在生成公钥时，通常会选取较为接近的两个大素数作为p和q，用于抵御$Pollard\ \rho$算法攻击。

连分式算法。为了描述连分式算法，首先需要引入$M.Kraitchik$因式分解格式和分解基算法。$M.Kraitchik$因式分解格式由如下几步组成：

(1) 生成一系列同余式$u \equiv v\ mod\ n, u \neq v$。

(2) 给出上述同余中u和v的部分或完全分解。

(3) 从同余式中挑选出一个子集$u_i \equiv v_i\ mod\ n, i = 1,2,\cdots$，使$\prod_i u_i$和$\prod_i v_i$为平方数。令$s = \sqrt{\prod_i v_i}$，$t = \sqrt{\prod_i u_i}$，则$t^2 \equiv s^2\ mod\ n$。

(4) 计算$(t \pm s, n)$，若它不等于n，则找到了n的一个非平凡因子。否则，另找一组同余式，重复上述过程。

对上述$M.Kraitchik$格式的一个重要改进是所谓的“分解基”算法，大部分现代因子分解法均沿用了这个基本方法。在$M.Kraitchik$格式中，选取$u = x^2$（x随机选取或由某公式生成），v取为u对n的最小绝对剩余，$|v| < n/2$。分解基算法预先取定一个中等大小的y（y远小于n）。

令$B=\{P_1,P_2,\cdots,P_k\}$，其中$P_1=-1$，$P_2,\cdots,P_k$为不超过y的所有素数，称B为分解基。一个整数x，若x^2对n的最小绝对剩余可以表示为B中某些数之积，则称x为B^-数。用某种方式生成$u_i=x_i{}^2$，分解$v_i=u_i \bmod n$(最小绝对剩余)，若x_i为B^-数，则v_i可以表示成$v_i=\prod_{j=1}^{k}P_j{}^{\alpha_{ij}}$，则这个$x_i$及$v_i$是所需要的，保留下来。若$x_i$不是$B^-$数，则抛弃$x_i$及$v_i$。一旦生成足够多的$B^-$数$x_i$，便可以从中挑出一些，使对应的$v_i$相乘为平方数。

事实上，若$v_i=\prod_{j=1}^{k}P_j{}^{\alpha_{ij}}$，令$\alpha_i=(\alpha_{i1},\alpha_{i2},\cdots,\alpha_{ik})$，把$\alpha_i$看成是$Z_2^k$中的向量，这里$Z_2^k$为$Z_2$上的$k$维线性空间，这些$v_i$相乘为平方数的充要条件是相应的向量之和$\sum\alpha_i$为$Z_2^k$中的零向量。由于$Z_2^k$中任何$k+1$个向量必定线性相关，所以最多只要生成$k+1$个$B^-$数，便可挑出一些$v_i$，使其相乘为平方数，令$s^2$为这些$v_i$之积，而$t$为对应的$x_i$之积，则$t^2\equiv s^2 \bmod n$。于是，至少有一半把握可获得$n$的一个非平凡因子。若不幸得到的是$n$的平凡因子，则重复上述过程或从已经分解的$v_i$中挑选另一组使其积为平方数，生成$t$和$s$，进行上述测试。若这样的过程进行$r$次，找到$n$的一个非平凡因子的概率大于等于$1-1/2^r$。

给定一个实数x，用如下方法构造它的连分式展开。令$a_0=[x]$，$x_0=x-a_0$；$a_1=[1/x_0]$，$x_1=1/x_0-a_1$；一般地，若$x_{i-1}\neq 0$，则令$a_i=[1/x_{i-1}]$，$x_i=1/x_{i-1}-a_i$，$i>1$。若进行到某步时$x_i=0$，则过程停止，否则，继续往下做。于是得到x的连分式展开

$$x=a_0+\cfrac{1}{a_1+\cfrac{1}{a_2+\cdots+\cfrac{1}{a_i+x_i}}}$$

通常记为

$$x=a_0+\frac{1}{a_1+}\,\frac{1}{a_2+}\cdots\frac{1}{a_i+x_i}$$

若x为有理数，上述过程在有限步之后结束，若x为无理数，则过程直至无穷。设x为无理数，并设

$$\frac{b_i}{c_i}=a_0+\frac{1}{a_1+}\,\frac{1}{a_2+}\cdots\frac{1}{a_{j-1}+}\,\frac{1}{a_j}$$

其中b_i/c_i为即约分数，称为x的连分式的第i阶逼近分数。

命题 18-2 有

(1) $b_i=a_ib_{i-1}+b_{i-2}$，$b_0=a_0$，$b_1=a_0a_1+1$；$c_i=a_ic_{i-1}+c_{i-2}$，$c_0=1$，$c_1=a_1$，$i=2,3,\cdots$。

(2) $b_ic_{i-1}-b_{i-1}c_i=(-1)^{i-1},i\geq 1$。

(3) 设x为大于1的实数，则对所有i，$\left|b_i{}^2-x^2c_i{}^2\right|<2x$。

在上述**命题 18-2** 中取$x=\sqrt{n}$，则有$\left|b_i{}^2-nc_i{}^2\right|<2\sqrt{n}$，即$b_i{}^2$对$n$的最小绝对剩余的绝对值始终小于$2\sqrt{n}$，所以$b_i{}^2$是$M.Kraitchik$算法中$u$的一个很好的选取方法，将其应用于$M.Kraitchik$的格式中，是连分式法的最初形式。

但真正把上述方法精细化且引入了分解基技巧的是$M\cdot A\cdot Morrison$和$J\cdot Brillhart$，$Morrison$和$Brillhart$的连分式分解法为：

首先选取一个恰当的正整数y，取$B=\{p_1,p_2,\cdots,p_k\}$，其中$p_1=-1$，$p_2,\cdots,p_k$为不超过y的并且使n为它们的二次剩余的所有素数。令$b_{-1}-1=0$，$b_0=a_0=\sqrt{n}$，$x_0=\sqrt{n}-a_0$，计算$v_0=b_0{}^2-n$，然后对$i=1,2,\cdots$，计算

(1) $a_i=[1/x_{i-1}]$，$x_i=1/x_{i-1}-a_i$。

(2) $b_i\equiv a_ib_{i-1}+b_{i-2} \bmod n$。

(3) $v_i\equiv b_i{}^2 \bmod n$，取最小绝对剩余。

(4) 计算v_i对B的分解式，对不能完全分解的v_i弃之。当计算出至少$k+1$个完全无解的v_i之后，停止上述过程。

(5) 按分解基算法的格式找出v_i的一个子集使其相乘为平方数，并构选相应的的t和s，计

算$(t \pm s, n)$，若它为n的非平凡因子，到此结束，分解成功，否则，继续生成新的b_i及上述过程。

整体算法执行流程如下，此处为按照单一v_i是否为平方数来判断结束条件：

```
seq_P, seq_Q = 0, 1
a0 = floor(sqrt(n))
seq_a = [a0]
seq_p = [0,a0]
i = 1
while 1:
    # P_k,Q_k 序列
    seq_P = seq_a[0]*seq_Q-seq_P
    seq_Q = divmod(n-seq_P**2,seq_Q)[0]
    t = (seq_P+sqrt(n))/seq_Q
    seq_a[0] = floor(t)
    # p_k 序列
    if i == 1:
        seq_p.append(a0*seq_a[0]+1)
    else:
        seq_p[0] = seq_p[1]
        seq_p[1] = seq_p[2]
        seq_p[2] = seq_a[0]*seq_p[1]+seq_p[0]
    # 分解因式
    if i%2 == 0 and seq_Q.is_square():
        s = int(sqrt(seq_Q))
        factor = [gcd(seq_p[1]-s,n),gcd(seq_p[1]+s,n)]
        if factor[0] != 1 and factor[1] != 1:
            print(factor[0], factor[1])
            break
    i += 1
```

例 18-3 接下来，通过一个例子来体会上述过程，试分解$n = 4399$。

按照连分式计算的过程，可以求得a_i, b_i, v_i，$i = 0,1,2,\cdots$，数据如下：

i	0	1	2	3
a_i	66	3	12	1
b_i	66	199	2454	2653
v_i	-43	10	-115	9

一般的，需构建分解基并按分解基算法的格式找出v_i的一个子集使其相乘为平方数。此处可以看到，$v_3 = 9$恰好是平方数，因而可以得到$v_3 \equiv {b_3}^2 \ mod\ n$，通过计算$\gcd(\sqrt{v_i} \pm b_3, n)$可以求得$n = 4399$的两个非平凡因子为53和83。

Dixon 随机平方算法。该算法基于费马分解法的思想，即假定可以找到一对正整数$a \neq \pm b (mod\ N)$，满足$a^2 \equiv b^2 (mod\ n)$，则$\gcd(a \pm b, n)$便是n的一个非平凡因子，从而达到分解大整数n的目的。

随机平方算法使用一个较小的因子基B，因子基是b个小素数的集合。选取几个整数z，使得$z^2\ mod\ n$的所有素因子都在因子基B中，并且，保证这些z^2相乘后，其乘积分解后的每个因子基内的素因子都恰好出现偶数次，此时，便可以建立起一个满足期望的同余方程$a^2 \equiv b^2 (mod\ n)$，有望得出n的分解。

该算法的关键在于如何选取满足要求的整数z。记$B = \{p_1, p_2, \dots, p_b\}$，假设已经得到多于$b$个不同的同余方程

$$z_i^2 \equiv p_1^{\alpha_{(1,i)}} \times p_2^{\alpha_{(2,i)}} \times \dots \times p_b^{\alpha_{(b,i)}} \ (mod\ n)$$

构造$\mathbb{Z}_2$上的向量

$$a_i = \left(\alpha_{(1,i)}, \alpha_{(2,i)}, \dots, \alpha_{(b,i)}\right) mod\ 2$$

通过高斯消元法，可以找到向量集a的子集，使得这些向量在$\mathbb{Z}_2$上的和为$(0,0,\dots,0)$，即线性相关，此时说明该子集对应的z_i集合的乘积会使B内所有素因子的幂次α恰好为偶数，满足前

述要求，可以构建出同余方程$a^2 \equiv b^2 (mod\ n)$。而且，只要选取的z足够“随机”，且个数多于$b+1$，那么最终导致$a \equiv \pm b (mod\ N)$的概率最多为50%，是足够高效的。

最后，有关整数z的选取。为了使得z^2可以在B内完全分解，比较常用的一种做法是，使用形如$i + \lceil \sqrt{kn} \rceil$及其附近的整数，这些整数经过平方、模$n$运算后，一般会比较小，更容易使其素因子完全落在$B$内；还有一种常见做法是，选取形如$\lfloor \sqrt{kn} \rfloor$及其附近的整数，这些整数经过平方、模$n$运算后，会比$n$小一点，代表着$-z^2\ mod\ n$是比较小的，此时将$-1$添加到$B$中，便会很容易地使其素因子完全落在$B$内。

例 18-4 接下来通过一个示例体会上述过程，试分解$n = 1829$

取$B = \{-1,2,3,5,7,11,13\}$，计算可知

$$\sqrt{n} = 42.77, \sqrt{2n} = 60.48, \sqrt{3n} = 74.07, \sqrt{4n} = 85.53$$

取整数$42, 43, 61, 74, 85, 86$，有

$$z_1^2 \equiv 42^2 \equiv -65 \equiv -1 \times 5 \times 13\ mod\ n$$
$$z_2^2 \equiv 43^2 \equiv 20 \equiv 2^2 \times 5\ mod\ n$$
$$z_3^2 \equiv 61^2 \equiv 63 \equiv 3^2 \times 7\ mod\ n$$
$$z_4^2 \equiv 74^2 \equiv -11 \equiv -1 \times 11\ mod\ n$$
$$z_5^2 \equiv 85^2 \equiv -91 \equiv -1 \times 7 \times 13\ mod\ n$$
$$z_6^2 \equiv 86^2 \equiv 80 \equiv 2^4 \times 5\ mod\ n$$

对应着$\mathbb{Z}_2$上的六个向量，这虽然没有达到足够的$b+1=8$个向量的数量，但在当前情况下已经足够使用了。

$$a_1 = (1,0,0,1,0,0,1)\ mod\ 2 = (1,0,0,1,0,0,1)$$
$$a_2 = (0,2,0,1,0,0,0)\ mod\ 2 = (0,0,0,1,0,0,0)$$
$$a_3 = (0,0,2,0,1,0,0)\ mod\ 2 = (0,0,0,0,1,0,0)$$
$$a_4 = (1,0,0,0,0,1,0)\ mod\ 2 = (1,0,0,0,0,1,0)$$
$$a_5 = (1,0,0,0,1,0,1)\ mod\ 2 = (1,0,0,0,1,0,1)$$
$$a_6 = (0,4,0,1,0,0,0)\ mod\ 2 = (0,0,0,1,0,0,0)$$

不难发现

$$a_2 + a_6 = (0,0,0,0,0,0,0)$$
$$a_1 + a_2 + a_3 + a_5 = (0,0,0,0,0,0,0)$$

分别可以构造出同余等式

$$(43 \times 86)^2 \equiv 2^2 \times 5 \times 2^4 \times 5\ (mod\ 1829)\ 即\ 40^2 \equiv 40^2\ (mod\ 1829)$$
$$(42 \times 43 \times 61 \times 85)^2 \equiv (2 \times 3 \times 5 \times 7 \times 13)^2 (mod\ 1829)\ 即\ 1459^2 \equiv 901^2 (mod\ 1829)$$

此时，成功构建出了满足要求的数对，随后即可直接计算

$$\gcd(1459 + 901, 1829) = 59$$

成功得到$n = 1829$的一个非平凡因子59。

18.1.3 现代大整数分解算法

椭圆曲线分解算法。根据通过上一节中关于$Pollard\ P-1$算法的描述，推断若N没有$p-1$光滑的素因子p，$Pollard\ P-1$算法无法实现对N的快速分解。为了弥补$Pollard\ P-1$算法所存在的缺陷，可以用域F_p上的随机椭圆曲线产生的群代替乘法群Z_p，因为椭圆曲线上的群的阶可以因曲线的不同而改变，若相应群的阶不够光滑，则可重新选择曲线。应用域F_p上椭圆曲线的点所产生的群实现对大整数N的因子分解，可消除$p-1$算法的部分弱点，增加成功分解的可能性。

由于p是未知的，不能直接在域F_p上定义椭圆曲线，而应在环Z/NZ上给出椭圆曲线E，但在实际的应用中只需使用E上的一个子集即可。

若N与2,3互素，$a, b \in Z/NZ$且$(4a^3 + 27b^2, N) = 1$，方程$y^2 = x^3 + ax + b$定义Z/NZ上的

椭圆曲线$E_{a,b}$，取$E_{a,b}$上的子集$V_N=\{(x,y,1)|x,y\in Z/NZ\}\cup\{\partial\}$，其中$\partial=(0,1,0)$。

在V_N上定义椭圆曲线的特殊“加法运算”，设$P,Q\in V_N$，$R=P+Q$：若$P=\partial$，则$R=Q$；若$Q=\partial$，则$R=P$；而当$P\neq\partial,Q\neq\partial$时，记$P=(x_1,y_1,1),Q=(x_2,y_2,1)$，计算$\gcd(x_1-x_2,N)$：

(1) 若$\gcd(x_1-x_2,N)$是N的非平凡因子，则计算停止，分解成功；

(2) 若$\gcd(x_1-x_2,N)=1$，计算$\lambda=(y_1-y_2)(x_1-x_2)^{-1}$，$x_3=\lambda^2-x_1-x_2$，$y_3=\lambda(x_1-x_3)-y_1$，此时，$R=P+Q=(x_3,y_3,1)\in V_N$；

(3) 若$\gcd(x_1-x_2,N)=N$，即$x_1=x_2$，计算$\gcd(y_1+y_2,N)$：

(3.1) 若$\gcd(y_1+y_2,N)$是N的非平凡因子，则计算停止，分解成功；

(3.2) 若$\gcd(y_1+y_2,N)=N$，即$y_1=-y_2$，则$R=\partial$；

(3.3) 若$\gcd(y_1+y_2,N)=1$，计算$\lambda=(3{x_1}^2+a)(y_1+y_2)^{-1}$，$x_3=\lambda^2-x_1-x_2$，$y_3=\lambda(x_1-x_3)-y_1$，此时，$R=P+Q=(x_3,y_3,1)\in V_N$。

由以上的“加法”运算定义可知，计算的结果或是得到N的一个非平凡因子，或是得到$R=P+Q\in V_N$。

在应用上述原理分解大整数N时，常适当选择相应的椭圆曲线上的一个点P，通过计算kP以实现对N的因子分解。

首先，确定Z/NZ上的椭圆曲线：$y^2=x^3+ax+b$，其中$(4a^3+27b^2,N)=1$，然后，选取适当的点$P=\{\alpha,\beta,1\}$，使得$\beta^2=\alpha^3+a\alpha+b(mod\ N)$，进一步地，计算$kP$。

在上述算法实现地过程中，涉及到一些主要的运算或算法：求两个整数的最大公因子的$Euclidean$算法；求乘法群Z_N^*中的元素的逆元的扩展$Euclidean$算法；椭圆曲线$E_{a,b}$上的子集V_N中额点的加法运算、倍乘运算。通过计算kP能否实现对N的因子分解的目的，关键在于前述的a,α,β的选取是否适当，若最初选取的a,α,β未能实现对N的因子分解，则需重新选取a,α,β直到成功分解N为止。

整体算法执行流程如下：

```
# r 表示随机选择多少条曲线，B 表示上界
def factor_ec(n,r,B):
    F=Zmod(n)
    x=0
    for i in range(r):
        # y^2=x^3+ax+b
        x=randint(0,n)
        y=randint(0,n)
        a=randint(0,n)
        b=(y^2-x^3-a*x)%n
        E = EllipticCurve(F, [a, b])
        P = E(x,y)
        for j in range(2,B+1):
            c=j
            Q=E([0,1,0])
            while c>0:
                if c%2==1:
                    d=gcd(Q[0]-P[0],n)
                    if d>1 and d!=n:
                        return d
                    Q=Q+P
                P=P+P
                c=c//2
            # Q=j*P
            P=Q
```

例 18-5 接下来，通过一个例子来体会上述过程，试分解$N=253$。

记初始选取点为P_0，则算法目标为通过计算kP_0的过程，判断相加点P,Q的横坐标x_1,x_2的算术关系即$\gcd(x_1-x_2,N)$为N的非平凡因子，以确定(x_1-x_2)在$mod\ N$上无逆元，此时$P+Q$无法通过上述方法计算得出。此处，kP_0计算考虑$k=B!$的情况，在c循环内部，P每次循环加

倍，Q记录点加结果，当c最低位为1时，将执行$P+Q$计算，此时判断$\gcd(x_1-x_2,N)$情况。

已知$N=253$，假设选取的椭圆曲线方程为$y^2=x^3+144x+133$，初始点$P_0=(223,162,1)$。

当计算$2!P_0$时：初始$P=P_0$，$Q=\partial$，当P加倍至$2P_0=(237,193,1)$时，此时$Q=\partial$，将执行$P+Q$计算，此时判断$\gcd(x_1-x_2,N)$不为N的非平凡因子，最终$Q=2!P_0$，P记录为$2P_0$；

当计算$3!P_0$时：初始$P=2P_0$，$Q=\partial$，当P尚未加倍时，此时$Q=\partial$，将执行$P+Q$计算，此时判断$\gcd(x_1-x_2,N)$不为非平凡因子，当P加倍至$4P_0=(110,208,1)$时，此时$Q=2P_0$，将执行$P+Q$计算，此时判断$\gcd(x_1-x_2,N)$不为非平凡因子，最终$Q=3!P_0$，P记录为$6P_0$；

当计算$4!P_0$时：初始$P=6P_0$，$Q=\partial$，当P加倍至$24P_0=(220,188,1)$时，此时$Q=\partial$，将执行$P+Q$计算，此时判断$\gcd(x_1-x_2,N)=\gcd(220,253)=11$为非平凡因子，计算结束。

由此，11为$N=253$的一个非平凡因子，容易求得其另一个非平凡因子为23。

二次筛法。二次筛法的基本算法依赖于构造以下方程的解，其中N是待分解的大整数：

$$A^2\equiv B^2\ mod\ N \qquad (1)$$

如果$A\not\equiv B\ mod\ N$且$A\not\equiv -B\ mod\ N$，则$(A+B,N)$和$(A-B,N)$是N的因子。

在单多项式版本的二次筛法中，使用以下单个多项式生成一组N的二次剩余：

$$Q(x)=(x+\lceil\sqrt{N}\rceil)^2-N\equiv H^2\ mod\ N \qquad (2)$$

由此可见，如果一个素数$p|Q(x)$，那么$p|Q(x+kp)$对所有$k\in Z$都成立。因此，当求解出$Q(x)\equiv 0\ mod\ p$时，多项式的值可以用于筛法分解。$Q(x)$的潜在因子p正是需要满足勒让德符号$(N/p)=1$的素数，以单位-1来表示因子的符号。

算法执行流程如下：

(1) 选择一个因子基$FB=\{p_i|(N/p_i)=1,p_i\ prime,i=1,\dots,F\}$，其中$F$选取某个合适值，$p_0=1$表示符号；

(2) 对于所有的$p_i\in FB$，求解二次方程$Q(x)\equiv 0\ mod\ p_i$。每个p_i会有两个根r_1和r_2；

(3) 对于适当的M，在区间$[-M,M]$上将筛选数组初始化为0；

(4) 对于所有的$p_i\in FB$，将$[\log(p_i)]$添加到筛选数组的$r_1,r_1\pm p_i,r_1\pm 2p_i$ …以及$r_2,r_2\pm p_i,r_2\pm 2p_i$ …处；

(5) $Q(x)$的值在$[-M,M]$上近似为$M\sqrt{N}$，因此将每个筛分位置与$[\log(N)/2+\log(M)]$进行比较。完全因子剩余的相应筛分值接近此值。针对这些，通过除法构造精确的因式分解。分解很少被找到，以至于做这个除法的时间可以忽略不计。在做除法时，不需要检查因子基中的所有素数。如果x是筛选数组中的位置，只需要计算$R\equiv x\ mod\ p$。只有当R等于两个根中的一个时才能进行多精度除法；

$$Q(x)=\prod_{i=0}^{F}p_i^{a_i},p_i\in FB \qquad (3)$$

令v_j为${H_j}^2=Q(x)$的指数$[\alpha_{j1}\ \alpha_{j2}\ \alpha_{j3}\ \cdots\ \alpha_{jF}]$的对应向量；

(6) 收集$F+1$因式分解。然后在通过消去$v_j\ mod\ 2$所形成的矩阵上找到一组剩余，其乘积通过高斯消去是$GF(2)$上的一个平方。这在指数上构建了一个$mod\ 2$的线性依赖项，并且在该依赖项中向量的乘积上形成了一个平方。然后构造同余式(1)式就很简单了。

这种方法的主要困难在于必须获得与因子基中素数数目相等的完全因子剩余。为了得到足够的因式分解式，M必须非常大，并且剩余的大小与M呈线性增长。

$Peter\ Montgomery$提出了一种解决该问题的方法：简单地使用多重多项式来生成剩余，并在一个小得多的区间内筛选每个多项式。利用多重多项式可以使筛分间隔保持较小，从而使剩余更容易分解。它允许在单多项式版本中使用不到总筛长十分之一的方式找到足够的因子剩余。其改变多项式的代价也很小。

有关$Q(x)=Ax^2+Bx+C$中系数的选择。为了使$Q(x)$生成二次剩余，要求

$$B^2 - 4AC = N$$

由于表达式最后同余于$0\ or\ 1\ mod\ 4$，这意味着如果$N \equiv 3\ mod\ 4$，则必须预先乘以一个小常数k，使得$kN \equiv 1\ mod\ 4$。一般来说，这也是一件好事，因为它通常可以使得在小素数中找到较为丰富的因子基。在$[-M, M]$上保持$Q(x)$的值较小，在某种适当的意义上会十分有利。有多种有效方法可以做到这一点，例如：

$$(a)\ Minimize\ sup|Q(x)|\ over\ [-M, M]$$

$$or$$

$$(b)\ Minimize\ \int_{-M}^{M} |Q(x)|dx$$

$$or \qquad (4)$$

$$(c)\ Minimize\ \int_{-M}^{M} Q^2(x)dx$$

$$subject\ to$$

$$(d)\ B^2 - 4AC = kN\ and\ A, B, C \in Z$$

很容易看出$(4a)$和$(4b)$本质上是相等的。抛物线底的长度是$2M$，它的面积与它的高成正比。放松整数约束，解决上面的每个拉格朗日乘数问题，会发现它们都产生了本质上相同的结果。$(4a)$的准确答案是

$$\begin{aligned} A &= W_1\sqrt{kN}/M, \\ B &= 0, \qquad (5) \\ C &= W_2 M\sqrt{kN}, \end{aligned}$$

其中

$$W_1 = \sqrt{2}/2\ and\ W_2 = -1/2\sqrt{2}.$$

不同最小化问题的结果之间的唯一区别是常数W_1和W_2的微小变化。

$Q(x)$在$[-M, M]$上的最大值为$M\sqrt{kN}/2\sqrt{2}$，比式(2)和$Sandia$的特殊q多项式提高了$\sqrt{8}$。

$B = 0$直接从对称性考虑而来，但$(4a)$和$(4c)$对A和C给出了类似的结果，因为约束$B^2 - 4AC = kN$对$Q(x)$的形状极具约束力。最简单的理解方式是，在根处其斜率为$\pm\sqrt{kN}$。事实上，的确希望抛物线变得平整，但对判别式的约束意味着曲线必须有一定的“陡度”。因此，除了上下平移抛物线以外，没有其他可做的办法。

选择A，B和C的一个简单方法来自快速求模平方根方法。满足$(4d)$则必须有

$$B^2 \equiv kN\ mod\ 4A \qquad (6)$$

设$A = D^2$，$(D/kN) = 1$，$D \equiv 3\ mod\ 4$，以及$A \approx \sqrt{kN/2}/M$。希望D是素数，因为如果因子基种的素数能够整除A，那么$Q(x) \equiv 0\ mod\ p$只有一个根，且在$[-M, M]$上$p|Q(x)$的概率从$2/p$下降到$1/p$。D只是一个可能的素数，但对于实际目的而言已经足够。或者，可以选择D作为不在因子基中的素数的乘积，且必须知道其分解才能够求解式(6)。实际的算法中将D作为一个可能的素数。为了找到系数，计算

$$h_0 \equiv (kN)^{(D-3)/4}\ mod\ D, \qquad (7a)$$

$$h_1 \equiv kNh_0 \equiv (kN)^{(D+1)/4}\ mod\ D. \qquad (7b)$$

然后

$${h_1}^2 \equiv kN(kN)^{\frac{D-1}{2}}\ mod\ D \equiv kN\ mod\ D \quad since\ (D/kN) = 1. \qquad (8)$$

令

$$h_2 = (2h_1)^{-1}\left[\frac{kN - {h_1}^2}{D}\right]\ mod\ D. \qquad (9)$$

现在则有

$$B \equiv h_1 + h_2 D\ mod\ A \qquad (10)$$

和

$$B^2 \equiv {h_1}^2 + 2h_1h_2D + {h_2}^2D^2 \equiv kN \bmod A. \qquad (11)$$

因为B一定是奇数，如果是偶数，就用A减去B。

$(2h_1)^{-1} \bmod D$的值很容易得到，因为$h_0 \equiv {h_1}^{-1} \bmod D$已经计算。则有

$$C = \left[\frac{B^2 - kN}{4A}\right]. \qquad (12)$$

实际上，没有必要实际计算(12)，因为实际上并不需要c的值。但是，它可以用来检验其他的计算。计算并保存$1/2D \bmod kN$的值以供以后使用。这将使得在找到因式分解式时能快速计算$Q(x)$。

由于选择D的方式，也有

$$Q(x) \equiv H^2 \equiv \left(\frac{2Ax + B}{2D}\right)^2 \bmod kN. \qquad (13)$$

需要注意的是，如果x位于$Q(x)$的实根之间，那么$Q(x)$是负的，必须用kN减去它的值。

求系数的开销主要是D上的可能素数和剩余检验，以及h_1和$1/2h_1 \bmod D$和$1/2D \bmod kN$的计算。然而，要做的总计算量很小。

最后，$Q(x) \bmod p_i, p_i \in FB$的根是

$$\left(-B \pm \sqrt{kN}\right)(2A)^{-1} \bmod p_i, \qquad (14)$$

改变多项式的大部分开销发生在计算式(14)中。计算(14)的开销主要是$(1/2A) \bmod p_i$的计算，这必须对因子基中的所有素数进行计算。即使有一个有效的算法来做这件事，比如扩展的欧几里得算法，当改变多项式时，通常必须执行数千次。

综上所述，二次筛法实际常用的基本步骤为：

(1) 选择一个乘数k，使$kN \equiv 1 \bmod 4$，且kN富含小的二次剩余。一般倾向于使得$kN \equiv 1 \bmod 8$，因为只有在这种情况下才有$2 \in FB$；

(2) 选择因子基的大小F，筛间隔的长度为$2M + 1$，以及较大的素数公差T；

(3) 计算测试值

$$\left[\log\left(\frac{M\sqrt{kN/2}}{pmax^T}\right)\right],$$

其中$pmax$是因子基中最大的素数。如果$T \le 2$，那么当筛分中的值超过这个值时，对应的$Q(x)$值将被完全分解。如果$T > 2$，那么$Q(x)$值可能不能完全分解。然而，这些部分的因式分解在算法后续过程中也可以起到一定作用；

(4) 计算因子基FB，针对所有的$p_i \in FB$计算$\sqrt{kN} \bmod p_i$。针对所有的$p_i \in FB$计算$[\log(p_i)]$；

(5) 在上一步骤中找到的许多因子分解式在因子基上是分解不完全的，同时，值得注意的是，如果$Q(x)$被分解为

$$Q(x) = \prod_i {p_i}^{a_i} L, \qquad L > 1, \qquad (15)$$

那么，当发现两个或多个L值相同的实例出现时，可以将式(15)对应的实例相乘。这在等式右边产生了L^2的因子，可以把这个结果保留到矩阵约简步骤中。注意，L不一定是素数：只需要两个或两个以上就可以匹配。通过使用L的值作为键值对(15)的所有实例进行排序来搜索匹配项。一般称(15)为$large\ prime$因数分解。步骤(2)中T的值可以控制L的大小，选择保留所有值在$pmax^T$以下的L，其中$pmax$是因子基中最大的素数。这可以使运行时间减少一半以上。

令FF为在因子基上完全分解的剩余数值，令FT为以F为因子基大小的因子分解总数。那么

$$R = \frac{FT}{F + FT - FF}. \qquad (16)$$

同时，使用该式子来确定何时找到了足够的因式分解式；

(6) 矩阵约简。最后，收集所有找到的因式分解式，并在$GF(2)$上约简矩阵。对于每一个线性依赖关系S，有

$$P_1 \equiv \prod_j H_j \ mod\ kN \quad for\ j \in S,$$

$$P_2 \equiv \prod_j {p_i}^{\sum_j V_{ji}/2} \ mod\ kN \quad for\ all\ p_i \in FB\ and\ \ j \in S. \tag{17}$$

如果$P_1 \not\equiv P_2\ mod\ kN$且$P_1 \not\equiv -P_2\ mod\ kN$，那么$(P_1 + P_2, kN)$和$(P_1 - P_2, kN)$是$N$的因子。

18.2 示例分析

18.2.1 2017 SECCON very smooth

题目要求：给出一个 HTTPS 加密的流量包 s.pcap，试解密以获取其中的 flag。

题目分析：使用 binwalk、wireshark 等工具提取出该流量包内使用的三个证书，随后通过 openssl 命令查看证书内容，可以发现它们都是 RSA 证书，且公钥参数n一致，皆为

Modulus=D546AA825CF61DE97765F464FBFE4889AD8BF2F25A2175D02C8B6F2AC0C5C27B67035AEC192B3741DD1F4D127531B07AB012EB86241C09C081499E69EF5AEAC78DC6230D475DA7EE17F02F63B6F09A2D381DF9B6928E8D9E0747FEBA248BFFDFF89CDFAF4771658919B6981C9E1428E9A53425CA2A310AA6D760833118EE0D71

可见，该模数n为 1024 比特，十进制值即

149767527975084886970446073530848114556615616489502613024958495602726912268566044330103850191720149622479290535294679429142532379851252608925587476670908668848275349192719279981470382501117310509432417895412013324758865071052169170753552224766744798369054498758364258656141800253652826603727552918575175830897

同时，根据题目名称“very smooth”，可以猜测n的素因子p可能为平滑数。

尝试使用$Pollard\ P-1$算法进行分解

```
n =
149767527975084886970446073530848114556615616489502613024958495602726912268566044330103850191720149622479290535294679429142532379851252608925587476670908668848275349192719279981470382501117310509432417895412013324758865071052169170753552224766744798369054498758364258656141800253652826603727552918575175830897

i=1
a=2
while 1:
    i = i+1
    a = power_mod(a, i, n)
    d = gcd(a-1,n)
    if 1<d and d<n:
        print(d, ' * ', n/d)
        break
```

很快就可以分解出素因子$n = p \times q =$

11807485231629132025602991324007150366908229752508016230400000000000000000000000000000000000000000000000000000000000000000000000000000000001 $\times$ 12684117323636134264468162714319298445454220244413621344524758865071052169170753552224766744798369054498758364258656141800253652826603727552918575175830897

接下来，尝试使用$William\ P+1$算法进行分解

```
def Lucas(A, M, n):
    v1, v2 = A, (A**2 - 2) % n
```

```python
    for bit in bin(M)[3:]:
        if bit == "0":
            v1, v2 = (v1**2 - 2) % n, (v1*v2 - A) % n
        else:
            v1, v2 = (v1*v2 - A) % n, (v2**2 - 2) % n
    return v1

n = 14976752797508488697044607353084811455661561648950261302495849560272691226856604433010385019172014962247929053529467942914253237985125260892558747667090866884827534919271927998147038250111731050943241789541201332475886507105216917075355222476674479836905449875836425865614180025365282660372755291857517583089 7
A = 3

i=1
v=A
while 1:
    i = i+1
    v = Lucas(v,i,n)
    d = gcd(v-2,n)
    if 1<d and d<n:
        print(d, ' * ', n/d)
        break
```

同样可以很快得出相同的分解答案。

接下来，依据上述参数n、p、q以及$e = 65537$，可以轻易地计算出私钥参数d

```python
p=11807485231629132025602991324007150366908229752508016230400000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
q=12684117323636134264468162714319298445454220244413621344524758865071052169170753552224766744798369054498758364258656141800253652826603727552918575175830897

e=65537
phi_N=(p-1)*(q-1)
d=inverse_mod(e, phi_N)
print(d)
```

最后，利用 Crypto.PublicKey 等 Python 工具包，将上述私钥参数打包生成 PEM 格式的私钥文件，将该私钥文件导入 wireshark 中，打开原始流量包，即可得到明文以及 flag。

18.2.2 非平滑因子大整数分解

题目要求：试分解整数$n = 2000000000000000000080400000000000000000003933$。

题目分析：首先，尝试使用$Pollard\ P-1$、$William\ P+1$、$Pollard\ \rho$等传统分解算法来进行分解，发现长时间都无法分解得到n的素因子p，可见$p-1$、$p+1$都不是平滑度足够小的平滑数。那么接下来，考虑使用二次筛法等现代分解算法来进行求解。

解题代码：

```python
def list_prod(a): return reduce(lambda x, y: x * y, a)


def mpqs(n):
    root_n = floor(sqrt(n))
    root_2n = floor(sqrt(n+n))

    # 素数基的上界
    jy = 10
    # 平滑度约束
```

```
    bound = int(RDF(jy*log(n, 10)**2))

    prime = []
    par_prime = {}
    mod_root = []
    log_p = []
    num_prime = 0
    hit_par_prime = 0
    used_prime = {}

    # 从 2 开始，寻找小的素数列表，(n/p)=1,bcs p|y^2-n
    p = 2
    while p < bound or num_prime < 3:

        # 勒让德符号
        if p > 2:
            leg = legendre_symbol(n, p)
        else:
            leg = n & 1

        if leg == 1:
            prime += [p]
            mod_root += [mod(n, p).sqrt().lift()]
            log_p += [RDF(log(p, 10))]
            num_prime += 1
        elif leg == 0:
            print('通过试除法得到素因子: ', p)
            return p
        p = next_prime(p)

    # x 取值范围
    x_max = num_prime*10
    # f(x)取值范围
    m_val = (x_max * root_2n) >> 1
    # 降低阈值
    thresh = RDF(log(m_val, 10) * 0.735)  # 阈值对数

    # 去掉贡献小的小素数
    min_prime = next_prime(int(thresh*3))
    while legendre_symbol(n, min_prime) != 1:
        min_prime = next_prime(min_prime)
    if min_prime > bound:  # 计算出错
        return -1
    pos_min_prime = prime.index(min_prime)

    fudge = sum(log_p[i] for i, p in enumerate(prime) if p < min_prime)/4
    thresh -= fudge
    num_poly = 0
    root_A = floor(sqrt(root_2n / x_max))

    # 筛选平滑数
    mt = matrix(ZZ, 0, num_prime+1)
    xlist = []
    rowcount = 0
    factor = 1
    while factor == 1 or factor == n:
        # 寻找整数 A，使:
        # A 约为 sqrt(2*n) / x_max
        # A 是一个完全平方数
        # sqrt(A)是素数，且 n 是 sqrt(A)的二次残差
        while True:
```

```
            root_A = next_prime(root_A)
            leg = legendre_symbol(n, root_A)
            if leg == 1:
                break
            elif leg == 0:
                print('筛选平滑数时发现素因子:', root_A)
                return root_A

        A = root_A * root_A

        # 寻找整数 B，使:
        # B*B 是一个模 n 的二次残差，使 B*B-A*C = n
        b = mod(n, root_A).sqrt().lift()
        tmp = (b + b).inverse_mod(root_A)
        B = (b + (n - b*b) * tmp) % A
        # B*B-A*C = n 即 C = (B*B-n)/A
        C = (B*B - n) / A
        num_poly += 1

        # 筛选(-x_max,xmax)范围内的素因子
        sums = [0.0]*(2*x_max)
        sums_dict = {}
        for i in range(pos_min_prime+1, num_prime):
            p = prime[i]
            logp = log_p[i]

            if A % p == 0:
                continue
            inv_A = A.inverse_mod(p)

            a = ((mod_root[i] - B) * inv_A) % p
            b = (-(mod_root[i] + B) * inv_A) % p

            k = 0
            # 每个循环更改一对值
            while k < x_max:
                # a+kp
                if k+a < x_max:
                    sums[k+a] += logp
                    if sums[k+a] > thresh:
                        sums_dict[k+a] = 1
                if k+b < x_max:
                    sums[k+b] += logp
                    if sums[k+b] > thresh:
                        sums_dict[k+b] = 1
                if k:
                    # a-kp
                    x1 = k-a+x_max
                    x2 = k-b+x_max

                    sums[x1] += logp
                    if sums[x1] > thresh:
                        sums_dict[x1] = 1
                    sums[x2] += logp
                    if sums[x2] > thresh:
                        sums_dict[x2] = 1
                k += p

        # 检查平滑性
        factor = 1
        for i in sums_dict:

```

```
            if factor != 1 and factor != n:
                break

            x = x_max-i if i > x_max else i
            # 由于 B*B-n = A*C
            # (A*x+B)^2 - n = A*A*x*x+2*A*B*x + B*B - n
            #               = A*(A*x*x+2*B*x+C)
            # 等价于
            # (A*x+B)^2 = A*(A*x*x+2*B*x+C) (mod n)
            # 由于 A 为完全平方数，因此 A 无需被筛选
            sieve_val = A*x*x + 2*B*x + C
            row = vector(ZZ, num_prime+1)

            if sieve_val < 0:
                # 第一列用来表示正负
                row[0] = 1
                sieve_val = -sieve_val

            j = 0
            while j < num_prime and sieve_val != 1:
                while sieve_val % prime[j] == 0:
                    row[j+1] += 1
                    sieve_val = sieve_val//prime[j]
                j += 1

            # 完全分解成列表中的素数的乘积
            if sieve_val == 1:
                xlist.append((root_A, A*x+B))
            else:
                if sieve_val not in par_prime.keys():
                    par_prime[sieve_val] = (root_A, A*x+B, row)
                    continue
                else:
                    hit_par_prime += 1
                    xlist.append((root_A*par_prime[sieve_val][0]*sieve_val,
(A*x+B)*par_prime[sieve_val][1]))
                    row = row+par_prime[sieve_val][2]
            for j in range(1, len(row)):
                if row[j] != 0:
                    used_prime[j] = 1

            # 插入到矩阵最后一行
            mt = mt.stack(row)
            rowcount += 1

            # GF(2)上寻找线性相关组
            if rowcount > len(used_prime):
                ker = mt[0:rowcount].change_ring(GF(2)).left_kernel()
                s = ker.dimension()
                t = 1
                while t < s:
                    left = 1
                    right = 1
                    res = list(map(lift, ker[t]))
                    coef = mt.linear_combination_of_rows(res)
                    for k in range(rowcount):
                        if res[k] == 1:
                            left = (left*xlist[k][1]) % n
                            right = (right*xlist[k][0]) % n
                    for k in range(1, num_prime):
                        if coef[k] != 0:
```

```
                            right = (right*(power_mod(prime[k-1], coef[k]//2, n))) %
n
                    t += 1

                    factor = gcd(left-right, n)
                    if factor == 1 or factor == n:
                        continue
                    else:
                        break
        return factor


n = 200000000000000000008040000000000000003933
factor = mpqs(n)
print(n, ' = ', factor, '*', n/factor)
```

使用二次筛法，在几十秒的 CPU 时间内便可以得到分解结果

$$n = 10000000000000000000057 \times 20000000000000000000069$$

18.3 举一反三

作为非对称密码体制中的重要难解问题，大整数分解问题长期受到学界的重点关注。在本章中，首先讨论了有关大整数的素性判断问题，包括确定性素数检测方法和概率性素数检测方法，二者的主要区别在于前者效率低但结果确定、后者则效率高但结果不完全可信，其中，概率性素性检测处理除所讲解到的 Miller-Rabin 检测法外，还有 AKS 检测法、Baillie-PSW 检测法等等，它们皆实现了错误率可接受的多项式时间素数检测。

早至上个世纪初期，数学家们便陆续发明了一系列著名的传统大整数分解算法，包括但不限于$Pollard\ P-1$算法、$William\ P+1$算法、$Pollard\ \rho$算法、连分式算法、随机平方算法，以及最简单的试除法等等，它们作为现代分解算法的先驱，提供了许多作为基础的原始思路，但它们大多含有明显的缺陷，需要素因子p满足一定的条件，才能体现出足够高的分解效率。而现代 RSA 密钥生成器则会针对各个传统大整数分解算法的特点，加以优化，以提高 RSA 密钥参数的安全性。

20 世纪末期以来，当下广泛应用的现代大整数分解算法被逐步发展起来，以椭圆曲线分解法、二次筛法、数域筛法为主，构成了大整数分解问题的三驾马车，它们也是当今最有效的三类大整数分解算法。

进入 21 世纪，RSA 官方发布了 8 个难解模数挑战，分别为 RSA-576、RSA-640、RSA-704、RSA-768、RSA-896、RSA-1024、RSA-1536 和 RSA-2048，其中 RSA-d中的d代表该模数的比特长度，自 2003 年 RSA-576 被成功分解以来，至 2009 年，RSA-768 也已被成功分解，而这些分解都是利用数域筛法完成的。最新的一项大整数分解记录发生在 2019 年，795 比特的 RSA-240（十进制位数）模数被开源软件团队 CADO-NFS 成功分解，使用的也是基于数域筛法的改进分解算法。

第19章 离散对数问题

19.1 基本原理

在整数中，离散对数是一种基于同余运算和原根的一种对数运算。离散对数在一些特殊情况下可以快速计算，然而，通常没有非常具有效率的方法来计算它们。公钥密码学中部分重要算法的基础，是假设寻找离散对数的问题解，在仔细选择过的群中，并不存在有效率的求解算法。

19.1.1 离散对数

在密码学中，许多公钥密码体制都是基于离散对数问题而构建的，其安全性也依赖于离散对数问题的难解性。第一个也是最为著名的这类密码体制，是 ElGamal 密码体制。以下将以 ElGamal 密码体制为基础，引入离散对数问题。

ElGamal 密码体制是基于离散对数问题。首先在有限乘法群$(G,\cdot)$中描述这个问题。对于一个n阶元素$\alpha \in G$，定义

$$\langle \alpha \rangle = \{\alpha^i : 0 \le i \le n-1\}$$

容易看到，$\langle \alpha \rangle$是G的一个子群，$\langle \alpha \rangle$是一个n阶循环群。

通常情况下，取G为有限域$\mathbb{Z}_p$(p为素数)的乘法群，α为模p的本原元，此时$n = |\langle \alpha \rangle| = p-1$；另一种情况下，取$\alpha$为乘法群$\mathbb{Z}_p^*$的一个素数阶$q$的元素[其中$p$为素数，并且$p-1 \equiv 0(mod\ q)$]。在$\mathbb{Z}_p^*$中这种元素$\alpha$可以由本原元的$(p-1)/q$次幂得到。

在群$(G,\cdot)$的子群$\langle \alpha \rangle$中定义离散对数问题。

问题 19-1 离散对数

实例：乘法群$(G,\cdot)$，一个n阶元素$\alpha \in G$和元素$\beta \in \langle \alpha \rangle$

问题：找到唯一的整数a，$0 \le a \le n-1$，满足

$$\alpha^a = \beta$$

将这个整数a记为$log_\alpha \beta$，称为β的离散对数。

在密码中主要应用离散对数问题的如下性质：求解离散对数(可能)是困难的，而其逆运算指数运算可以应用平方-乘的方法有效地计算。换句话说，在适当的群G中，指数函数是单向函数。

ElGamal 提出了一个基于$(\mathbb{Z}_p^*,\cdot)$上离散对数问题的公钥密码体制。这个体制表述为密码体制 4-1。

密码体制 19-1 $\mathbb{Z}_p^*$上的 ElGamal 公钥密码体制

设p是一个素数，使得$(\mathbb{Z}_p^*,\cdot)$上的离散对数问题是难处理的，令$\alpha \in \mathbb{Z}_p^*$是一个本原元。令$\mathcal{P} = \mathbb{Z}_p^*$，$\mathcal{C} = \mathbb{Z}_p^* \times \mathbb{Z}_p^*$，定义

$$\mathcal{K} = \{(p,\alpha,a,\beta) : \beta \equiv \alpha^a (mod\ p)\}$$

p,α,β是公钥，a是私钥。

对$K = (p,\alpha,a,\beta)$，以及一个(秘密)随机数$k \in \mathbb{Z}_{p-1}$，定义

$$e_K(x,k) = (y_1, y_2)$$

其中

$$y_1 = \alpha^k\ mod\ p$$

且

$$y_2 = x\beta^k \ mod\ p$$

对$y_1, y_2 \in \mathbb{Z}_p^*$，定义

$$d_K(y_1, y_2) = y_2(y_1{}^a)^{-1} \ mod\ p$$

在 ElGamal 密码体制中，加密运算是随机的，因为密文既依赖于明文x又依赖于 Alice 选择的随机数k。所以，对于同一个明文，会有许多(事实上，有$p-1$个)可能的密文。

ElGamal 密码体制的工作方式可以非正式地描述如下：明文x通过乘以β^k“伪装”起来，产生y_2。值α^k也作为密文地一部分传送。Bob 知道私钥a，可以从α^k计算出β^k。最后用y_2除以β^k除去伪装，得到x。

以下例子可说明在 ElGamal 密码体制中所进行地计算。

例 19-1 设$p = 2579$，$\alpha = 2$。α是模p地本原元。令$a = 765$，所以

$$\beta = 2^{765} \ mod\ 2579 = 949$$

假定 Alice 现在想要传送消息$x = 1299$给 Bob。比如$k = 853$是她选择的随机数。那么她计算

$$y_1 = 2^{853} \ mod\ 2579 = 435$$

和

$$y_2 = 1299 \times 949^{853} \ mod\ 2579 = 2396$$

当 Bob 收到密文$y = (435,2396)$后，他计算

$$x = 2396 \times (435^{765})^{-1} \ mod\ 2579 = 1299$$

正是 Alice 加密的明文。

很显然，如果 Oscar 可以计算$a = log_\alpha \beta$，那么 ElGamal 密码体制就是不安全的，因为那时 Oscar 可以像 Bob 一样解密密文。因此，ElGamal 密码体制安全的一个必要条件，就是$\mathbb{Z}_p^*$上的离散对数问题是难处理的。一般正是这么认为的，当然p要仔细的选取，α是模p的本原元。特别是，对于这种形式的离散对数问题，不存在已知的多项式时间算法。为了防止已知的攻击，p应该至少取 300 个十进制位，$p-1$应该具有至少一个较“大”的素数因子。

19.1.2 离散对数穷举搜索

假定$(G,\cdot)$是一个乘法群，$\alpha \in G$是一个n阶元素。因而离散对数问题可以表达成下面的形式：给定$\beta \in \langle\alpha\rangle$，找出唯一的指数$a$，$0 \le a \le n-1$，使得$\alpha^a = \beta$。

从分析一些基本的算法开始，这些算法可以用于求解离散对数问题。在分析中假定，计算群G中两个元素的乘积需要常数[即$O(1)$]的时间。

首先，注意到离散对数问题可以通过$O(n)$的时间和$O(1)$存储空间穷举搜索解决。只要计算$\alpha, \alpha^2, \alpha^3, \cdots$，直到发现$\beta = \alpha^a$[上述序列中每一项$\alpha^i$通过前一项$\alpha^{i-1}$乘以$\alpha$得到]，因此总时间需要$O(n)$]。

另外一种方法是，预先计算出所有可能的值α^i，并对有序对(i, α^i)以第二个坐标排序列表，然后，给定β，对存储的列表执行一个二分查找，直到找到a使得$\alpha^a = \beta$。这需要$O(n)$时间预先计算α的n个幂，$O(nlogn)$时间对n个元素的排序。如果像通常分析算法那样，忽略掉对数因子，预先计算的时间就是$O(n)$。n个有序元素列表的二分查找时间为$O(logn)$。如果再次忽略对数因子项，可看到，离散对数问题可以用$O(1)$时间，$O(n)$步预先计算和$O(n)$存储空间解决。

以下将介绍几种针对离散对数问题的求解方法，包括$Shanks$算法、$Pollard's\ \rho$离散对数算法、$Pollard's\ kangaroo$算法、$Pohlig-Hellman$算法、指数演算法、$CADO-NFS$计算离散对数，等等。通过穷举搜索解决离散对数问题的方法，将不做详细讲述。

19.1.3 $Shanks$算法

$Shanks$算法，是一种非平凡的时间-存储折中算法，算法具体内容表述如下。

算法 19-1 *SHANKS*

```
def bsgs(a, b, bounds):
    Z = integer_ring.ZZ
    identity=a.parent().one()
    lb, ub = bounds
    if lb < 0 or ub < lb:
        raise ValueError("bsgs() requires 0<=lb<=ub")
    if a.is_zero() and not b.is_zero():
        raise ValueError("no solution in bsgs()")
    ran = 1 + ub - lb   # 区间的长度
    c = (b^-1) * (a^lb)
    m = ran.isqrt() + 1
    table = dict()
    d = c
    for i0 in xsrange(m):
        i = lb + i0
        if identity == d:       # identity == b^(-1)*a^i, 则返回 i
            return Z(i)
        table[d] = i
        d = d*a
    c = c * (d^-1)    # 此处即为 a**(-m)
    d = identity
    for i in xsrange(m):
        j = table.get(d)
        if j is not None:  # 然后 d == b*a**(-i*m) == a**j
            return Z(i * m + j)
        d = c*d
    print("log of %s to the base %s does not exist in %s" % (b, a, bounds))
#算法例子
p=809
R=GF(p)
a=R(3)
b=R(525)
ans = bsgs(a,b,bounds=(1,p))
print(ans)
```

针对上述算法描述，假定记乘法群为G，n为元素α的阶，欲求离散对数$log_\alpha\beta$。如果需要的话，已知$m=\lceil\sqrt{n}\rceil$，遍历j从$0\sim m-1$计算α^{mj}和对m个有序对(j,α^{mj})关于第二个坐标排序得到L_1，这两个步骤可以预先计算(然而，这并不影响渐近的运行时间)。其次，在遍历i从$0\sim m-1$计算$\beta\alpha^{-i}$和对m个有序对$(i,\beta\alpha^{-i})$关于第二个坐标排序得到L_2后，可以看到如果$(j,y)\in L_1$和$(i,y)\in L_2$，则

$$\alpha^{mj}=y=\beta\alpha^{-i}$$

因此

$$\alpha^{mj+i}=\beta$$

反过来，对任意的$\beta\in\langle\alpha\rangle$，有$0\le log_\alpha\beta\le n-1$。用$m$去除$log_\alpha\beta$，就可以将$log_\alpha\beta$表示为形式

$$log_\alpha\beta=mj+i$$

其中$0\le j,i\le m-1$。$j\le m-1$可以从下面得出

$$log_\alpha\beta\le n-1\le m^2-1=m(m-1)+m-1$$

因而找到$(j,y)\in L_1$和$(i,y)\in L_2$将会成功(但是，如果恰巧$\beta\notin\langle\alpha\rangle$，就不会成功)。

很容易实现这个算法，使其运行时间为$O(m)$，存储空间为$O(m)$(忽略对数因子)。以下包含部分算法细节：遍历j从$0\sim m-1$计算α^{mj}可以先计算α^m，然后依次乘以α^m计算其幂。这步总的花费时间为$O(m)$。同样地，遍历i从$0\sim m-1$计算$\beta\alpha^{-i}$和花费的时间为$O(m)$。(j,α^{mj})和$(i,\beta\alpha^{-i})$的排序利用有效的排序算法，花费时间为$O(mlogm)$。最后，做一个对两个表L_1和L_2同时进行的遍历，完成$(j,y)\in L_1$和$(i,y)\in L_2$的寻找，需要的时间为$O(m)$。

以下为一个使用$Shanks$算法的例子。

例 19-2 假定要在$(\mathbb{Z}_{809}^*,\cdot)$中求出$log_3 525$。注意 809 是素数，3 是$\mathbb{Z}_{809}^*$中本原元，这时$\alpha = 3$，$n = 808$，$\beta = 525$和$m = \lceil\sqrt{808}\rceil = 29$。则

$$\alpha^{29}\ mod\ 809 = 99$$

首先，对于$0 \le j \le 28$计算有序对$(j, 99^j\ mod\ 809)$。得到下面的列表

(0,1)	(1,99)	(2,93)	(3,308)	(4,559)
(5,329)	(6,211)	(7,664)	(8,207)	(9,268)
(10,644)	(11,654)	(12,26)	(13,147)	(14,800)
(15,727)	(16,781)	(17,464)	(18,632)	(19,275)
(20,528)	(21,496)	(22,564)	(23,15)	(24,676)
(25,586)	(26,575)	(27,295)	(28,81)	

这些序对排序后产生L_1。

第二个列表包括序对$(i, 525 \times (3^i)^{-1}\ mod\ 809)$，$0 \le i \le 28$。如下所示：

(0,525)	(1,175)	(2,328)	(3,379)	(4,396)
(5,132)	(6,44)	(7,554)	(8,724)	(9,511)
(10,440)	(11,686)	(12,768)	(13,256)	(14,355)
(15,388)	(16,399)	(17,133)	(18,314)	(19,644)
(20,754)	(21,521)	(22,713)	(23,777)	(24,259)
(25,356)	(26,658)	(27,489)	(28,163)	

排序后得到L_2。

现在同时遍历两个列表，发现(10,644)在L_1中，(19,644)在L_1中。所以可以进行计算

$$log_3 525 = (29 \times 10 + 19)\ mod\ 809 = 309$$

这个结果可以通过验证$3^{309} \equiv 525(mod\ 809)$得到检验。

19.1.4 $Pollard's\ \rho$离散对数算法

假定$(G,\cdot)$是一个群，$\alpha \in G$是一个n阶元素，要计算元素$\beta \in \langle\alpha\rangle$的离散对数。由于$\langle\alpha\rangle$是$n$阶循环群，可以把$log_\alpha\beta$看作$\mathbb{Z}_n$中的元素。

$Pollard's\ \rho$算法，通过迭代一个貌似随机的函数f，构造一个序列$x_1, x_2, \cdots$。一旦在序列中得到两个元素x_i和x_j，满足$x_i = x_j$，这里$i < j$，就有希望计算出$log_\alpha\beta$。为了能够节省时间和空间，需要寻求一种与$Pollard's\ \rho$大整数分解算法一样的碰撞$x_i = x_{2i}$。

设$S_1 \cup S_2 \cup S_3$是群G的一个划分，它们的元素个数大致相同。定义函数$f: \langle\alpha\rangle \times \mathbb{Z}_n \times \mathbb{Z}_n \to \langle\alpha\rangle \times \mathbb{Z}_n \times \mathbb{Z}_n$如下：

$$f(x,a,b) = \begin{cases} (\beta x, a, b+1) & x \in S_1 \\ (x^2, 2a, 2b) & x \in S_2 \\ (\alpha x, a+1, b) & x \in S_3 \end{cases}$$

而且，构造的每个三元组(x,a,b)要满足性质$x = \alpha^a\beta^b$。选择初始三元组满足这个性质，比如(1,0,0)。可以看出，如果(x,a,b)满足这个性质，$f(x,a,b)$也满足这个性质。因此，定义

$$(x_i,a_i,b_i) = \begin{cases} (1,0,0) & i = 0 \\ f(x_{i-1},a_{i-1},b_{i-1}) & i \ge 1 \end{cases}$$

比较三元组(x_{2i},a_{2i},b_{2i})和(x_i,a_i,b_i)，直到发现$x_{2i} = x_i$，$i \ge 1$。这时有

$$\alpha^{a_{2i}}\beta^{b_{2i}} = \alpha^{a_i}\beta^{b_i}$$

记$c = log_\alpha\beta$，则下面等式成立

$$\alpha^{a_{2i}+cb_{2i}} = \alpha^{a_i+cb_i}$$

由于α是n阶元素，就有

$$a_{2i} + cb_{2i} \equiv a_i + cb_i (mod\ n)$$

改写后有

$$c(b_{2i} - b_i) \equiv a_i - a_{2i} (mod\ n)$$

如果$\gcd(b_{2i} - b_i, n) = 1$，就可以如下解出$c$：

$$c = (a_i - a_{2i})(b_{2i} - b_i)^{-1} \ mod \ n$$

以下将用一个例子说明上述算法的应用。注意，必须保障$1 \notin S_2$[因为$1 \in S_2$时，对任意的$i \geq 0$都有$x_i = (1,0,0)$]。

例 19-3 整数$p = 809$是素数，可以验证$\alpha = 89$在$\mathbb{Z}_{809}^*$中是$n = 101$阶元素。元素$\beta = 618$在子群$\langle\alpha\rangle$中；计算$log_\alpha \beta$。

假定如下定义S_1, S_2, S_3：

$$S_1 = \{x \in \mathbb{Z}_{809} : x \equiv 1 (mod\ 3)\}$$
$$S_2 = \{x \in \mathbb{Z}_{809} : x \equiv 0 (mod\ 3)\}$$
$$S_3 = \{x \in \mathbb{Z}_{809} : x \equiv 2 (mod\ 3)\}$$

对于$i = 1,2,\cdots$，得到三元组(x_{2i}, a_{2i}, b_{2i})和(x_i, a_i, b_i)的值如下：

i	(x_i, a_i, b_i)	(x_{2i}, a_{2i}, b_{2i})
1	(618,0,1)	(76,0,2)
2	(76,0,2)	(113,0,4)
3	(46,0,3)	(488,1,5)
4	(113,0,4)	(605,4,10)
5	(349,1,4)	(422,5,11)
6	(488,1,5)	(683,7,11)
7	(555,2,5)	(451,8,12)
8	(605,4,10)	(344,9,13)
9	(451,5,10)	(112,11,13)
10	(422,5,11)	(422,11,15)

上述列表中第一个碰撞是$x_{10} = x_{20} = 422$。要解的方程是

$$c = (5 - 11)(15 - 11)^{-1} \ mod \ 101 = (6 \times 4^{-1}) \ mod \ 101 = 49$$

所以，在$\mathbb{Z}_{809}^*$中$log_{89} 618 = 49$。

离散对数的$Pollard's\ \rho$算法由算法 19-2 给出。在该算法中，继续假定$\alpha \in G$具有阶数n，并且$\beta \in \langle\alpha\rangle$。

当$\gcd(b' - b, n) > 1$时，即$\gcd(b' - b, n) = d$，容易证明同余方程$c(b' - b) \equiv a - a' \ (mod\ n)$恰好有$d$个解。假如$d$不是很大的话，可以直接算出同余方程的$d$个解并检验哪个解是正确的。

在函数f的随机性的合理假设下，可以期望在n阶循环群中用算法的$O(\sqrt{n})$次迭代计算离散对数。

算法 19-2 *Pollard's ρ*离散对数算法

```
from sage.rings.integer import Integer
from sage.rings.finite_rings.integer_mod_ring import IntegerModRing
def rho(a, base, ord=None, hash_function=hash):
    partition_size = 20
    memory_size = 4
    ord = base.multiplicative_order()
    ord = Integer(ord)
    if not ord.is_prime():
        raise ValueError("for Pollard rho algorithm the order of the group must be prime")
    mut = hasattr(base, 'set_immutable')
    isqrtord = ord.isqrt()
    reset_bound = 8 * isqrtord  # 采取一定保证
    I = IntegerModRing(ord)
    for s in range(10):  # 避免无限循环
        # 随机步长配置
        m = [I.random_element() for i in range(partition_size)]
        n = [I.random_element() for i in range(partition_size)]
        M = [(base ^ (Integer(m[i]))) * (a ^ (Integer(n[i])))
             for i in range(partition_size)]
```

```
        ax = I.random_element()
        x = base ^ Integer(ax)
        if mut:
            x.set_immutable()
        bx = I(0)
        sigma = [(0, None)] * memory_size
        H = {}  # 记录
        i0 = 0
        nextsigma = 0
        for i in range(reset_bound):
            # 随机步长，需要一个有效 hash
            s = hash_function(x) % partition_size
            x, ax, bx = ((M[s] * x), ax + m[s], bx + n[s])
            if mut:
                x.set_immutable()
            # 寻找碰撞
            if x in H:
                ay, by = H[x]
                if bx == by:
                    break
                else:
                    res = sage.rings.integer.Integer((ay - ax) / (bx - by))
                    if (base ^ res) == a:
                        return res
                    else:
                        break
            # 需要记录数值
            elif i >= nextsigma:
                if sigma[i0][1] is not None:
                    H.pop(sigma[i0][1])
                sigma[i0] = (i, x)
                i0 = (i0 + 1) % memory_size
                nextsigma = 3 * sigma[i0][0]  # 3 是一个经验性的值
                H[x] = (ax, bx)
    raise ValueError("Pollard rho algorithm failed to find a logarithm")
#算法例子
p=809
R=GF(p)
a=R(89)
b=R(618)
ans = rho(b,a)
print(ans)
```

19.1.5 *Pollard's kangaroo*算法

给定循环群G，群中元素g,h，大整数N，计算n使得

$$h = g^n (0 \le n \le N)$$

*Pollard's kangaroo*算法解决上述离散对数问题需要$2\sqrt{N}$次运算和附加的少量存储空间。算法的运行过程如下：

选择一个正整数的小集合$S = (s_1, s_2, \cdots, s_k)$作为*kangaroo*的跳跃步集合，集合中元素的均值m大小与$\sqrt{N}$相当。从群G中随机均匀选取一些元素构成可区分集合$D = \{g_1, g_2, \cdots, g_t\}$，集合$D$的大小约为$|D|/|G| = c/\sqrt{N}$，其中$c$为常数且$c \gg 1$。

定义一个从群G到集合S的随机映射$f: G \to S$。

一只*kangaroo*跳跃就对应一个G中的序列：

$$g_{i+1} = g_i \cdot g^{f(g_i)}, i = 0,1,2,\cdots$$

从给定的g_0开始。同时定义一个序列d_i，令$d_0 = 0$，

$$d_{i+1} = d_i + f(g_i), i = 0,1,2,\cdots$$

因此d_i是$kangaroo$前i次跳跃的距离和，那么，

$$g_i = g_0 \cdot g^{d_i}, i = 0,1,2,\cdots$$

$Pollard's\ kangaroo$是基于随机步的算法，每只$kangaroo$每次跳跃一步。定义两只$kangaroo$分别为$tame\ kangaroo$和$wild\ kangaroo$，记为T和W。T从区间中点(即$g^{N/2}$)开始向区间右侧跳跃，W从群元素h开始(即未知离散对数的群元素)向区间右侧跳跃，两者使用同样的随机步集合。在串行计算机上对T和W交替操作，无论何时只要$kangaroo$的跳跃落地点的群元素属于可区分点集合D，以三元组的形式$(T_i/W_i, d_i, T/W)$记录此时的落地点$T_i(W_i)$值和d_i值，以及类型标志$T(W)$。并将三元组以$T_i(W_i)$值为索引存入索引表或二叉树中，当某个$T_i(W_i)$值被不同类型(T或者W)$kangaroo$访问时即发生碰撞，算法终止。若用$d_i(T)$和$d_j(W)$分别表示发生碰撞时T和W各自的跳跃距离和，则可求出$n = N/2 + d_i(T) - d_j(W)$。当无法发生碰撞时，可尝试改变$S$集合或者$f$映射重新执行算法。

下面给出一个简单例子对$Pollard's\ kangaroo$算法进行解释。

例 19-4 设$p = 29$，$\alpha = 2$。p是素数，α是模p的本原元素，设$\beta = 18$，要计算$a = log_\alpha \beta$。

若已知a在$mod\ p$范围内的边界是[2,27]，则区间长度$width = 27 - 2 = 25$。预先构建集合大小为k的随机步集合$\{(1,2),(4,16),(2,4)\}$，其中每一项$(r, base^r)$表示随机步长为r和α^r，k是满足$2^k \geq \sqrt{width} + 1$的最小正整数。此外，定义$hash$函数，从$G$中元素映射到随机步集合下标$r$。

记两只$kangaroo$分别为T和W。T的起始点为G中元素$\alpha^{\left\lfloor \frac{2+27}{2} \right\rfloor}(mod\ 29) = 28$，$W$的起始点为$\beta = 18$。假定依照$hash$函数，$T$和$W$从起始点开始的跳跃步骤和距离计算如下：

i	0	1	2	3	4	5
$(T_i, d_i(T))$	(28,0)	(13,4)	(5,8)	(10,9)	(15,13)	(1,14)
$(W_i, d_i(W))$	(18,0)	(7,1)	(25,5)	(23,9)	(17,10)	(5,11)

可以发现，$T_2 = W_5$，发生碰撞，此时可计算$n = \left\lfloor \frac{2+27}{2} \right\rfloor + d_2(T) - d_5(W) = 11$，即得到$log_\alpha \beta = log_2 18 = 11$。

下面将给出经典$Pollard's\ kangaroo$算法效率的粗略分析。记$Mul(k)$表示模k比特数的一次完全乘法代价，$|f|$表示计算一次f映射的时间代价。则算法中计算一只$kangaroo$的一次跳跃需要的时间代价是$Mul(\|p\|) + |f|$。记位于前面的$kangaroo$为F，后面为B。算法可以分为三步：

Stage1. B赶上F的起始点。此时B跳跃次数的期望值为$N/(4m)$，所以两只的跳跃总次数为$N/(2m)$。

Stage2. B与F的跳跃点重合。由于平均步长为m，简单的认为F路径中的跳跃点包含了所有点的$1/m$。所以该步中的跳跃总次数为$2m$。

Stage3. B继续跳跃直至遇到可区分点集合D中的某点。记θ为某一群元素属于可区分集合D的概率，则该步中的跳跃总次数为$2/\theta$。

根据上面的分析，算法执行流程中需要的总跳跃步数为：$N/(2m) + 2m + 2/\theta$。取$\theta = clogN/\sqrt{N}$，常数$c > 0$，平均情况下的跳跃步数为$(2 + 1/clogN)\sqrt{N} = (2 + O(1))\sqrt{N}$。所以，对于经典$Pollard's\ kangaroo$算法，总的运行时间代价为$(Mul\|p\| + |f|) \times (2 + 1/clogN)\sqrt{N}$。

算法 19-3 $Pollard's\ kangaroo$算法

```
def func_lambda(a, base, bounds, hash_function=hash):
    from sage.rings.integer import Integer
    lb, ub = bounds
    if lb < 0 or ub < lb:
        raise ValueError("discrete_log_lambda() requires 0<=lb<=ub")
    # 检查可变性
    mut = hasattr(base, 'set_immutable')
    width = Integer(ub - lb)
    N = width.isqrt() + 1
    M = dict()
```

```
    for s in range(10):  # 避免无限循环
        # 随机步长配置
        k = 0
        while 2**k < N:
            r = sage.misc.prandom.randrange(1, N)
            M[k] = (r, (base ^ r))
            k += 1
        # 第一个随机步长
        H = (base ^ ub)
        c = ub
        for i in range(N):
            if mut:
                H.set_immutable()
            r, e = M[hash_function(H) % k]
            H = (H * e)
            c += r
        if mut:
            H.set_immutable()
        mem = set([H])
        # 第二个随机步长
        H = a
        d = 0
        while c - d >= lb:
            if mut:
                H.set_immutable()
            if ub >= c - d and H in mem:
                return c - d
            r, e = M[hash_function(H) % k]
            H = (H * e)
            d += r
    raise ValueError("Pollard Lambda failed to find a log")
#算法例子
p=809
R=GF(p)
a=R(3)
b=R(525)
ans = func_lambda(b,a,(200,600))
print(ans)
```

19.1.6 $Pohlig-Hellman$算法

下面将研究$Pohlig-Hellman$算法。假定

$$n=\prod_{i=1}^{k}{p_i}^{c_i}$$

其中p_i是不同的素数。值$a=log_\alpha\beta$是模n(唯一)确定的。首先知道，如果能够对每个i，$1\le i\le k$，计算出$a\ mod\ {p_i}^{c_i}$，就可以利用中国剩余定理计算出$a\ mod\ n$。所以假设q是素数，

$$n\equiv 0(mod\ q^c)$$

且

$$n\not\equiv 0(mod\ q^{c+1})$$

说明如何计算

$$x=a\ mod\ q^c$$

其中$0\le x\le q^c-1$。把x以q的幂表示为

$$x=\sum_{i=0}^{c-1}a_iq^i$$

其中对于$0\le i\le c-1$，$0\le a_i\le q-1$。还有，可以把a表示为

$$a = x + sq^c$$

s是某一整数。因而就有

$$a = \sum_{i=0}^{c-1} a_i q^i + sq^c$$

算法的第一步是计算a_0。算法中利用的主要事实是以下等式

$$\beta^{n/q} = \alpha^{a_0 n/q} \quad (21.1)$$

式(21.1)的证明如下：

$$\begin{aligned}
&\beta^{n/q} = (\alpha^a)^{n/q} \\
&= \left(\alpha^{a_0 + a_1 q + \cdots + a_{c-1} q^{c-1} + sq^c}\right)^{n/q} \\
&= (\alpha^{a_0 + Kq})^{n/q} && \text{其中}K\text{是整数} \\
&= \alpha^{a_0 n/q} \alpha^{Kn} \\
&= \alpha^{a_0 n/q}
\end{aligned}$$

有了式(21.1)，确定a_0就很简单了。比如，可以计算

$$\gamma = \alpha^{n/q}, \gamma^2, \cdots,$$

直到对某个$i \le q-1$

$$\gamma^i = \beta^{n/q}$$

这时，$a_0 = i$。

如果$c = 1$，事情已经解决。否则$c > 1$，继续确定$a_1, \cdots, a_{c-1}$。这可以与计算a_0类似的方式进行。记$\beta_0 = \beta$，对于$1 \le j \le c-1$，定义

$$\beta_j = \beta \alpha^{-(a_0 + a_1 q + \cdots + a_{j-1} q^{j-1})}$$

把式(21.1)推广为：

$$\beta_j^{\,n/q^{j+1}} = \alpha^{a_j n/q} \quad (21.2)$$

可以看到，在$j = 0$时，式(21.2)归结为式(21.1)。

式(21.2)的证明类似于式(21.1)的证明：

$$\begin{aligned}
&\beta_j^{\,n/q^{j+1}} = \left(\alpha^{a-(a_0 + a_1 q + \cdots + a_{j-1} q^{j-1})}\right)^{n/q^{j+1}} \\
&= \left(\alpha^{a_j q^j + \cdots + a_{c-1} q^{c-1} + sq^c}\right)^{n/q^{j+1}} \\
&= \left(\alpha^{a_j q^j + K_j q^{j+1}}\right)^{n/q^{j+1}} && \text{其中}K_j\text{是整数} \\
&= \alpha^{a_j n/q} \alpha^{K_j n} \\
&= \alpha^{a_j n/q}
\end{aligned}$$

所以，给定β_j，能够从式(21.2)直接计算出a_j。

为了使算法的描述完整，可以看到，当a_j已知的情况下，β_{j+1}能够由β_j通过简单的递归关系计算出：

$$\beta_{j+1} = \beta_j \alpha^{-a_j q^j} \quad (21.3)$$

所以，交替利用式(21.2)和式(21.3)，可以计算出$a_0, \beta_1, a_1, \beta_2, \cdots, \beta_{c-1}, a_{c-1}$。

$Pohlig - Hellman$算法可表述为算法 19-4。总结一下该算法的运算，α是乘法群G的一个n阶元素，q是素数

$$n \equiv 0 (mod\ q^c)$$

且

$$n \not\equiv 0 (mod\ q^{c+1})$$

算法计算出了$a_0, \cdots, a_{c-1}$，其中

$$log_\alpha \beta\ mod\ q^c = \sum_{i=0}^{c-1} a_i q^i$$

算法 19-4 $Pohlig-Hellman$算法

```
from sage.groups.generic import bsgs
# Pohlig-Hellman 法
def pohlig_hellman_DLP(g, y, p):
    crt_moduli = []
    crt_remain = []
    for q0, r in factor(p-1):
        q = q0 ^ r
        x = bsgs(pow(g,(p-1)//q,p), pow(y,(p-1)//q,p),(1,p))
        crt_moduli.append(q)
        crt_remain.append(x)
    x = crt(crt_remain, crt_moduli)
    return x
#算法例子
g = 2
y = 18
p = 29
x = pohlig_hellman_DLP(g, y, p)
print(x)
print(pow(g, x, p) == y)
```

下面使用一个例子对$Pohlig-Hellman$算法加以说明。

例 19-5 设$p=29$，$\alpha=2$。p是素数，α是模p的本原元素，有

$$n=p-1=28=2^2\times 7^1$$

设$\beta=18$，要计算$a=lb18$。首先计算$a \bmod 4$，随后计算$a \bmod 7$。

应用$Pohlig-Hellman$，先选择$q=2$和$c=2$。得到$a_0=1$和$a_1=1$。所以，$a\equiv 3(mod\ 4)$。

其次对于$q=7$，$c=1$应用$Pohlig-Hellman$。算出$a_0=4$，所以，$a\equiv 4(mod\ 7)$。

最后应用中国剩余定理求解方程组

$$a\equiv 3(mod\ 4)$$
$$a\equiv 4(mod\ 7)$$

得到$a\equiv 11(mod\ 28)$。即算得在$\mathbb{Z}_{29}$中$lb18=11$。

考察$Pohlig-Hellman$算法的复杂度。不难看出，直接实现算法的时间为$O(cq)$。然而，这可以改进，注意到每次计算满足$\delta=\alpha^{in/q}$的值i，可以视为解一个特殊的离散对数问题。特别地，$\delta=\alpha^{in/q}$当且仅当

$$i=log_{\alpha^{n/q}}\delta$$

元素$\alpha^{n/q}$的阶是q，所以每个i可以用$O(\sqrt{q})$时间(利用$Shanks$算法)计算。这样，原$Pohlig-Hellman$算法的复杂度可以降到$O(c\sqrt{q})$。

19.1.7 指数演算法

前述介绍的算法可以应用到任何群。以下将介绍的指数演算法，该方法非常特殊：它用于计算$\mathbb{Z}_p^*$中的离散对数这种特定的情形，其中p是素数，α是模p的本原元素。在这种特定的情形，指数演算法比前述介绍的算法要快。

指数演算法计算离散对数，主要是模仿了许多最好的因子分解算法。这个方法使用了一个因子基。因子基是由一些“小”素数组成的集合$\mathcal{B}$。假设$\mathcal{B}=\{p_1,p_2,\cdots,p_B\}$。第一步(预处理步)是计算因子基中$B$个素数的离散对数。第二步，利用这些离散对数，计算所要求的离散对数。

假设C比B稍大；比如$C=B+10$。在预计算阶段，构造C个模p的同余方程，它们具有下述形式：

$$\alpha^{x_j}\equiv {p_1}^{a_{1j}}{p_2}^{a_{2j}}\cdots {p_B}^{a_{Bj}}(mod\ p)$$

$1\le j\le C$。它们等价于

$$x_j\equiv a_{1j}log_\alpha p_1+\cdots+a_{Bj}log_\alpha p_B(mod\ p-1)$$

$1\le j\le C$。给定B个“未知量”$log_\alpha p_i(1\le j\le B)$的$C$个同余方程，希望存在模$p-1$下的

唯一解。如果是这样的话，就可以算出因子基元素的离散对数。

如何产生C个期望的同余方程呢？一个基本的方法是，随机地取一个数x，计算$\alpha^x \bmod p$，确定是否$\alpha^x \bmod p$的所有因子在$\mathcal{B}$中(例如，可以利用试除法)。

假设预计算步骤已经顺利实现。利用$Las\ Vegas$型的随机算法计算所求的离散对数。选择一个随机数$s(1 \le s \le p-2)$，计算

$$\gamma = \beta\alpha^s \bmod p$$

现在试图在因子基$\mathcal{B}$上分解γ。如果成功，就得到如下的同余方程：

$$\beta\alpha^s \equiv {p_1}^{c_1}{p_2}^{c_2}\cdots{p_B}^{c_B}(mod\ p)$$

等价于

$$log_\alpha\beta + s = c_1 log_\alpha p_1 + \cdots + c_B log_\alpha p_B (mod\ p-1)$$

由于上式中除$log_\alpha\beta$外，其余的项都已知，容易解出$log_\alpha\beta$。

算法 19-5 指数演算法

```
 1. def is_Bsmooth(b, n):
 2.     factors = list(factor(int(n)))
 3.     if len(factors) != 0 and factors[-1][0] <= b:
 4.         return True, dict(factors)
 5.     else:
 6.         return False, dict(factors)
 7. def find_congruences(B, g, p, congruences=[]):
 8.     unique = lambda l: list(set(l))
 9.     bases = []
10.     max_equations = prime_pi(B)
11.     while True:
12.         k = randint(2, p-1)
13.         ok, factors = is_Bsmooth(B, pow(g,k,p))
14.         if ok:
15.             congruences.append((factors, k))
16.             if len(congruences) >= max_equations:
17.                 break
18.     bases = unique([base for c in [c[0].keys() for c in congruences] for base in
c])
19.     return bases, congruences
20. def to_matrices(R, bases, congruences):
21.     M = [[c[0][base] if base in c[0] else 0 \
22.             for base in bases] for c in congruences]
23.     b = [c[1] for c in congruences]
24.     return Matrix(R, M), vector(R, b)
25. def index_calculus(g, y, p, B=None):
26.     R = IntegerModRing(p-1)
27.     if B is None:
28.         B = ceil(exp(0.5*sqrt(2*log(p)*log(log(p)))))
29.     bases = []
30.     congruences = []
31.     for i in range(100):
32.         bases, congruences = find_congruences(B, g, p, congruences)
33.         M, b = to_matrices(R, bases, congruences)
34.         try:
35.             exponents = M.solve_right(b)
36.             break
37.         except ValueError:
38.             # 矩阵方程无解
39.             continue
40.     else:
41.         return None
42.     # a*g^y mod p
43.     while True:
44.         k = randint(2, p-1)
45.         ok, factors = is_Bsmooth(B, (y * pow(g,k,p)) % p)
```

```
        if ok and set(factors.keys()).issubset(bases):
            print('found k = {}'.format(k))
            break
    print('bases:', bases)
    print('q:', factors.keys())
    dlogs = {b: exp for (b,exp) in zip(bases, exponents)}
    x = (sum(dlogs[q] * e for q, e in factors.items()) - k) % (p-1)
    if pow(g, x, p) == y:
        return x
    return None
#算法例子
g = 5
y = 9451
p = 10007
x = index_calculus(g, y, p)
print(x)
print(pow(g, x, p) == y)
```

下面是一个特制的例子，用以说明算法的两个步骤。

例 19-6 整数$p = 10007$是素数。假定$\alpha = 5$是本原元素用做模p的离散对数的基。假定取$\mathcal{B} = \{2,3,5,7\}$作为因子基。当然$log_5 5 = 1$，因此，有三个因子基元素的对数要确定。

4063，5136 和 9865 属于需要的“幸运”指数。

当$x = 4063$时，计算

$$5^{4063}\ mod\ 10007 = 42 = 2 \times 3 \times 7$$

产生同余方程

$$log_5 2 + log_5 3 + log_5 7 \equiv 4063(mod\ 10006)$$

类似地，由于

$$5^{5136}\ mod\ 10007 = 54 = 2 \times 3^3$$

和

$$5^{9865}\ mod\ 10007 = 189 = 3^3 \times 7$$

进一步得到两个同余方程：

$$log_5 2 + 3log_5 3 \equiv 5136(mod\ 10006)$$

和

$$3log_5 3 + log_5 7 \equiv 9865(mod\ 10006)$$

现在有了具有三个未知量地三个同余方程，并且模 10006 具有唯一的解。即$log_5 2 = 6578$，$log_5 3 = 6190$和$log_5 7 = 1301$。

现在，假设要求$log_5 9451$。选择“随机”指数$s = 7737$，计算

$$9451 \times 5^{7763}\ mod\ 10007 = 8400$$

因为$8400 = 2^4 \times 3^1 \times 5^2 \times 7^1$在$\mathcal{B}$上完全分解，得到

$$\begin{aligned} log_5 9451 &= (4log_5 2 + log_5 3 + 2log_5 5 + log_5 7 - s)\ mod\ 10006 \\ &= (4 \times 6578 + 6190 + 2 \times 1 + 130 - 7736)\ mod\ 10006 \\ &= 6057 \end{aligned}$$

检查$5^{6057} \equiv 9451(mod\ 10007)$，得以验证。

人们对于各种版本的指数演算算法进行过启发式分析。在合理的假设下，算法的预计算阶段花费的渐近运行时间为

$$O(e^{(1+o(1))\sqrt{\ln p \ln\ln p}})$$

计算特定的离散对数所需时间为

$$O(e^{(1/2+o(1))\sqrt{\ln p \ln\ln p}})$$

19.1.8 $Cado-nfs$工具

$Cado-nfs$是 C / C ++中数域筛法（NFS）算法的完整实现，用于分解整数并计算有限域

中的离散对数。它包含与算法所有阶段相对应的各种程序，以及可能在计算机网络上并行运行它们的通用脚本。

(1) 基本使用方法：$GF(p)$中的离散对数

$cado-nfs.py$脚本可以用来计算$GF(p)$中的离散对数。例如，要计算$GF(p)$中的离散对数，是由目标值$target$为$92800609832959449330691138186$、$p-1$的$ell$因子为$10153850953424616963261743 9$、$p=191907783019725260605646959711$所确定的离散对数问题参数。命令行执行如下：

```
$ ./cado-nfs.py -dlp -ell 10153850953424616963261743 9
target=92800609832959449330691138186 191907783019725260605646959711
```

原则上，只要输入

```
$ ./cado-nfs.py -dlp -ell <ell> target=<target> <p>
```

就应该计算$GF(\langle p\rangle)$中<target>的离散对数和<ell>的模数。现在，只有 30、60、100 或 155 位左右的素数p才有参数(将在 parameters/dlp 子目录下检查)。如果没有给出目标，那么输出是一个包含所有因子基数元素的虚拟对数文件。

$cado-nfs$求解离散对数问题具有更多的灵活性。parameter/dlp/param.p60 中给出了一个参数文件例子。与整数因式分解的参数文件相比，主要区别在于与字符和 sqrt 有关的行消失了，而且还有一个与单个对数有关的额外参数块。

计算结束后，可以再次运行$cado-nfs.py$脚本，但目标 target 不同：其只运行最后一步。为了确保真正使用预先计算的数据，复制粘贴第一次计算的输出中包含“If you want to compute a new target…”的命令行，并在最后设置新的目标。

注：$cado-nfs$的离散对数是以一个任意的(未知的)基数给出的。如果想相对于一个特定的生成器g来定义它们，那么就必须计算g的对数，然后用这个值除以所有的对数。

以下将介绍一个简单例子，来讲述如何使用$cado-nfs$工具和使用时的注意事项。

例 19-7 $cado-nfs$工具使用

$cado-nfs$工具的常用命令如下：

```
$ ./cado-nfs.py -dlp -ell <ell> target=<target> <p>
```

其一般至少包含三个参数$ell, target, p$，其中p确定乘法群G中的$mod\ p$运算，$target$为待求离散对数的真数，ell为$p-1$的最大素因子，注意，待求离散对数的底数是$cado-nfs$算法参数文件确定的，即当需要求$a=log_g target$时，用户一般不能指定使用题设的g，而是自动使用工具预定义的基。

当需要求$log_g target$时，用户可先利用上述命令求出工具预定义基下的$logg$和$logtarget$，而后通过换底公式计算出$log_g target$。换底公式计算，即为计算$logtarget * logg^{-1}(mod\ ell)$。注意，上述计算的离散对数结果，都是真实离散对数结果在$mod\ ell$上的结果。

当计算出$a=log_g target$后，通过$g^a\ mod\ p=target$来验算可能会验算失败，因为a是$log_g target$在$mod\ ell$下的解，并非$mod\ (p-1)$下的解。通过中国剩余定理可以知道，如果能够确定$log_g target$在$mod\ ((p-1)/ell)$下的解，结合$mod\ ell$下的解，可以利用中国剩余定理确定$mod\ (p-1)$下的实际解。当然，当$(p-1)/ell$为小素数因子的乘积时，可通过修改剩余类进行遍历求得实际解，即循环判断

$$a+i*ell, i=0,1,2,\cdots,(p-1)/ell-1$$

是否满足$g^a\ mod\ p=target$来确定真实的$log_g target(mod\ (p-1))$。

(2) 使用$Joux-Lercier$多项式选择法

默认情况下，使用与因式分解相同的多项式选择算法。在某些(很多)情况下，使用 polyselect/dlpolyselect 中实现的$Joux-Lercier$多项式选择会好得多。为了使用它，有必要添加参数 jlpoly = true 并给出附加参数。

```
tasks.polyselect.degree = 3
```

```
    tasks.polyselect.bound = 5
    tasks.polyselect.modm = 5
```

这里，polynomial.degree 是具有小系数多项式的级数。另一个的级数会少一个。因此，在这个例子中，这是对级数(3,2)的选择。polynomial.bound 和 polynomial.modm 参数直接传递给 dlpolyselect。搜索是通过客户端/服务器机制并行化的，就像经典的多项式选择一样。每个任务在0和$modm-1$之间做一个值“$modr$”(同样，这是 dlpolyselect 的术语)。尝试的多项式的数量大约是$(2*bound+1)^{degree+1}$，因此这里是14641(越大越好，但这样多项式选择会持续更长时间)。

例如，上面的 30 位数的例子可以用JL多项式选择来完成，命令行如下。

```
    $ ./cado-nfs.py -dlp -ell 10153850953424616963261743 9 191907783019725260605646959711
jlpoly=true tasks.polyselect.bound=5 tasks.polyselect.modm=7 tasks.polyselect.degree=3
tasks.constructlog.checkdlp=false
```

在这种情况下，单个对数阶段的实现是基于 GMP-ECM 的，所以只有在安装了这个库并且被配置脚本检测到的情况下才可以使用。

注：tasks.constructlog.checkdlp=false 是为了禁用一些在JL模式下不能进行的一致性检查。

这仍然是实验性的，但是为JL多项式选择优化的参数可以在 parameters/dlp/Joux-Lercier/中找到。将 parameters/dlp/Joux-Lercier/params.p30 复制到 parameters/dlp/params.p30 将自动激活这个大小素数的JL多项式选择(但如果在编译时未能检测到 GMP-ECM，将引发崩溃)。例子如下。

```
    $ ./cado-nfs.py -dlp -ell 10153850953424616963261743
target=928006098329594493306911381 86 191907783019725260605646959711
```

然后应用JL进行运作并计算给定目标的对数。

(3) 使用非线性多项式

就像因式分解一样，可以使用两个非线性多项式进行DLP。除了$Joux-Lercier$多项式的选择之外，用户必须提供多项式文件。另外，当前的下降脚本将无法工作。

参见 README.nonlinear，可了解导入含有 2 个非线性多项式的多项式文件例子。

一个重要的问题是，由于这种情况下的下降法还没有发挥作用，如果没有线性多项式，脚本就没有办法检查结果。一个解决方法是设置 tasks.reconstructlog.partial = false，这样就可以在使用过滤过程中删除所有关系的同时进行许多一致性检查。

(4) 数值较小k的$GF(p^k)$中的离散对数

该算法对$GF(p^k)$中的离散对数计算“必要修改”工作。唯一的区别是，两个多项式必须有一个$GF(p)$上的k度公共不可约因子。这种情况下的多项式选择还没有包括在内，所以必须根据文献中的构造，自行建立，并按照 scripts/cadofactor/README 中的指示导入。另外，在这种情况下，还必须实现个别对数。

对于$GF(p^2)$中的DLP来说，事情要稍微复杂一些：

```
    $ ./cado-nfs.py -dlp -ell <ell> -gfpext 2
```

对于$p=7\ mod\ 8$来说应该是有效的，前提是有一个参数文件来确定p的大小(目前只支持 20 位小数的p)。

(5) 自建参数文件

如果目标大小的参数文件缺失，可以通过在现有的参数文件之间进行内插/外推来创建它们。需要 params.pNNN 和 pNNN.hint，其中 NNN 是目标尺寸。关于提示文件，见 parameters/dlp/README。

19.2 示例分析

19.2.1 *BSGS*算法破解*crackme java*

题目要求：试题提供一个 java 源文件，给出了加密和解密的算法，给出了一组密文，要求明文。题目所涉及 java 源文件内容如下：

```
1. import java.math.BigInteger;
2. import java.util.Random;
3.
4. public class Test1 {
5.     static BigInteger two =new BigInteger("2");
6.     static BigInteger p = new
BigInteger("113607382951770029984953840578931299649801318065095729278866758994222141744083339321508139393572797031615567671936218327956057084566287338770840153674977 11");
7.     static BigInteger h= new
BigInteger("785499889356720883127062723315576365894740561093810699808399138930736308583702836415480957781657751502156098549170760616578827421874269287530821624396691 6");
8.
9.     /*
10.     Alice write the below algorithm for encryption.
11.     The public key {p, h} is broadcasted to everyone.
12.    @param val: The plaintext to encrypt.
13.        We suppose val only contains lowercase letter {a-z} and numeric charactors,
and is at most 256 charactors in length.
14.    */
15.    public static String pkEnc(String val){
16.        BigInteger[] ret = new BigInteger[2];
17.        BigInteger bVal=new BigInteger(val.toLowerCase(),36);
18.        BigInteger r =new BigInteger(new Random().nextInt()+"");
19.        ret[0]=two.modPow(r,p);
20.        ret[1]=h.modPow(r,p).multiply(bVal);
21.        return ret[0].toString(36)+"=="+ret[1].toString(36);
22.    }
23.
24.    /* Alice write the below algorithm for decryption. x is her private key, which
she will never let you know.
25.    public static String skDec(String val,BigInteger x){
26.        if(!val.contains("==")){
27.            return null;
28.        }
29.        else {
30.            BigInteger val0=new BigInteger(val.split("==")[0],36);
31.            BigInteger val1=new BigInteger(val.split("==")[1],36);
32.            BigInteger s=val0.modPow(x,p).modInverse(p);
33.            return val1.multiply(s).mod(p).toString(36);
34.        }
35.    }
36.   */
37.
38.    public static void main(String[] args) throws Exception {
39.        System.out.println("You intercepted the following message, which is sent
from Bob to Alice:");
40.        BigInteger bVal1=new
BigInteger("a9hgrei38ez78hl2kkd6nvookaodyidgti7d9mbvctx3jjniezhlxs1b1xz9m0dzcexwiyhi4nhvazh
hj8dwb91e7lbbxa4ieco",36);
41.    BigInteger bVal2=new
BigInteger("2q17m8ajs7509yl9iy39g4znf08bw3b33vibipaa1xt5b8lcmgmk6i5w4830yd3fdqfbqaf82386z5o
dwssyo3t93y91xqd5jb0zbgvkb00fcmo53sa8eblgw6vahl80ykxeylpr4bpv32p7flvhdtwl4cxqzc",36);
```

```
    BigInteger r =new BigInteger(new Random().nextInt()+"");
    System.out.println(r);
        System.out.println(bVal1);
    System.out.println(bVal2);

System.out.println("a9hgrei38ez78hl2kkd6nvookaodyidgti7d9mbvctx3jjniezhlxs1b1xz9m0dzcexwiyh
i4nhvazhhj8dwb91e7lbbxa4ieco==2q17m8ajs7509yl9iy39g4znf08bw3b33vibipaa1xt5b8lcmgmk6i5w4830y
d3fdqfbqaf82386z5odwssyo3t93y91xqd5jb0zbgvkb00fcmo53sa8eblgw6vahl80ykxeylpr4bpv32p7flvhdtwl
4cxqzc");
        System.out.println("Please figure out the plaintext!");
    }
}
```

题目分析：结合$pkEnc$的函数内容可以知晓，其加密过程的基本功能为计算

$$r_0 = 2^r \ mod \ p$$

$$r_1 = b * h^r \ mod \ p$$

可以发现，r的范围为$[0, 2^{32})$，所以可以使用$BSGS$算法(即$Shanks$算法)求解。

解题代码：

```
from sage.groups.generic import bsgs
import base36
c1 =
int('a9hgrei38ez78hl2kkd6nvookaodyidgti7d9mbvctx3jjniezhlxs1b1xz9m0dzcexwiyhi4nhvazhhj8dwb9
1e7lbbxa4ieco', 36)
c2 =
int('2q17m8ajs7509yl9iy39g4znf08bw3b33vibipaa1xt5b8lcmgmk6i5w4830yd3fdqfbqaf82386z5odwssyo3
t93y91xqd5jb0zbgvkb00fcmo53sa8eblgw6vahl80ykxeylpr4bpv32p7flvhdtwl4cxqzc', 36)
print (c1, c2)
p =
113607382951770029984953840578931299649801318065095729278866758994222141744083339321508139393572797031615567671936218327956057084566287338770840153674977 11
h =
785499889356720883127062723315576365894740561093810699808399138930736308583702836415480957781657751502156098549170760616578827421874269287530821624396691 6
# 生成
const2 = 2
const2 = Mod(const2, p)
c1 = Mod(c1, p)
c2 = Mod(c2, p)
h = Mod(h, p)
r = bsgs(const2, c1, bounds=(1, 2 ^ 32))
print ('2', r)
num = int(c2 / (h**r))
print (base36.dumps(num))
```

解题结果：$r = 152351913$，明文为

ciscncongratulationsthisisdesignedbyalibabasecurity424218533

19.2.2 $Cado-nfs$计算DLP

题目要求：假定在一个DH交换协议的背景下，已知参数$p, g, h = g^x (mod \ p), k = g^y (mod \ p)$，以及$p-1$的因式分解，欲计算通信者之间协商的密钥$g^{xy} (mod \ p)$。并进行进一步的讨论分析。

题目示例：给定参数内容如下：

$p = 22345678901234567830123456789012345678901234567890123456807 1$

$g = 173111254804046301125$

$p - 1 = 2 * 5 *$
$22345678901234567830123456789012345678901234567890123456807$

$h = g^x (mod \ p) =$

$4934187330375128509560317493098121016496489415597804987492$0

$k = g^y (mod\ p) =$

$1147010785503565676377667024223788608331996333817020535033$9

(1) 计算$logh$

注意，此时的基不是题目内所选的g，而是算法运行过程中的另外一个基。

```
$ ./cado-nfs.py -dlp -ell 22345678901234567830123456789012345678901234567890123456807
target=4934187330375128509560317493098121016496489415597804987492$
22345678901234567830123456789012345678901234567890123456807$
```

其中，ell选择的是$p-1$的最大的素因子，运行得到结果：

$$logh = 11068439637671712943054178216756460395598012657532627052040(mod\ ell)$$

(2) 计算$logg$

根据提示，只需要执行：

```
$ ./cado-nfs.py /tmp/cado.503gn9v7/p60.parameters_snapshot.0
target=173111254804046301125
```

得到结果：

$$logg = 353051940241047920010586424126888471542192079897415989093$4(mod\ ell)$$

(3) 计算$x = log_g h$

换底公式计算：$logh * logg^{-1} (mod\ ell)$并验算

```
1. p=22345678901234567830123456789012345678901234567890123456807
2. R=GF(p)
3. g=R(173111254804046301125)
4. gx=R(4934187330375128509560317493098121016496489415597804987492$)
5. gy=R(1147010785503565676377667024223788608331996333817020535033$)
6. ell=22345678901234567830123456789012345678901234567890123456807
7. log_h = 11068439637671712943054178216756460395598012657532627052040
8. log_g = 353051940241047920010586424126888471542192079897415989093$
9. temp=log_h * inverse_mod(log_g, ell) % ell
10. temp;g^temp;gx
```

已经得到$x = 8480023$

(4) 计算$g^{xy} (mod\ p)$

得到333333333333333333333333333333333333333333333，即为口令。

以下作进一步讨论：

(5) 计算$logk$

运行：

```
$ ./cado-nfs.py /tmp/cado.503gn9v7/p60.parameters_snapshot.1
target=1147010785503565676377667024223788608331996333817020535033$
```

得到

$$logk = 210470646955338677907448831456292780092970033865585418919$51(mod\ ell)$$

(6) 计算$log_g k$

换底公式计算：$logk * logg^{-1} (mod\ ell)$

```
1. p=22345678901234567830123456789012345678901234567890123456807
2. R=GF(p)
3. g=R(173111254804046301125)
4. gx=R(4934187330375128509560317493098121016496489415597804987492$)
5. gy=R(1147010785503565676377667024223788608331996333817020535033$)
6. ell=22345678901234567830123456789012345678901234567890123456807
7. log_k = 21047064695533867790744883145629278009297003386558541891951
8. log_g = 353051940241047920010586424126888471542192079897415989093$
```

```
 9. temp=log_k * inverse_mod(log_g, ell) % ell
10. temp;g^temp;gy
```

经验算

$$y = logk * logg^{-1}(mod\ ell) =$$

$$855419465233406649452797354249204212197482762660957 9$$

以上x, y均比ell小，现在已知了

$$s = g^{xy}(mod\ p) = 333333333333333333333333333333333333333333$$

尝试求$333333333333333333333333333333333333333333$的离散对数

(7) 计算$logs$

运行：

```
$ ./cado-nfs.py /tmp/cado.503gn9v7/p60.parameters_snapshot.2
target=333333333333333333333333333333333333333333
```

得到

$$logs =$$

$$74152432460951710811543949074869474305985657888950114700 17(mod\ ell)$$

(8) 计算$log_g s$

换底公式计算：$logs * logg^{-1}(mod\ ell)$

```
 1. p=22345678901234567830123456789012345678901234567890123456807 1
 2. R=GF(p)
 3. g=R(173111254804046301125)
 4. s=333333333333333333333333333333333333333333
 5. gx=R(49341873303751285095603174930981210164964894155978049874920)
 6. gy=R(11470107855035656763776670242237886083319963338170205350339)
 7. ell=22345678901234567830123456789012345678901234567890123456807
 8. log_s = 74152432460951710811543949074869474305985657888950114700 17
 9. log_g = 35305194024104792001058642412688847154219207987 4159890934
10. temp=log_s * inverse_mod(log_g, ell) % ell
11. temp;g^temp;s
```

此时得到的是

$$log_g s =$$

$$55027306945661840667562194166869574746116399910142715698 96(mod\ ell)$$

而且验算也不能通过，说明$temp$值并不是最终的$log_g s$

那么如何得到$log_g s$呢？根据费马小定理，只需要得到$log_g s(mod\ p-1)$即可

因为$p-1 = 10 * ell$，已知

$$log_g s =$$

$$55027306945661840667562194166869574746116399910142715698 96(mod\ ell)$$

那么$log_g s(mod\ p-1)$必然是下列之一（剩余类改写）

$$55027306945661840667562194166869574746116399910142715698 96 + i * ell, i =$$

$$0,1,2,\cdots,9$$

于是穷举一下：

```
 1. p=22345678901234567830123456789012345678901234567890123456807 1
 2. R=GF(p)
 3. g=R(173111254804046301125)
 4. s=333333333333333333333333333333333333333333
 5. gx=R(49341873303751285095603174930981210164964894155978049874920)
 6. gy=R(11470107855035656763776670242237886083319963338170205350339)
 7. ell=22345678901234567830123456789012345678901234567890123456807
 8. log_s = 74152432460951710811543949074869474305985657888950114700 17
 9. log_g = 35305194024104792001058642412688847154219207987 4159890934
10. temp=log_s * inverse_mod(log_g, ell) % ell
11. for i in range(0,10):
12.     if g^(temp+i*ell)==s:
```

```
13.         print(i,temp+i*ell)
```

结果为(3,725397673982698875571265897837239945113153436946846419403 17)

说明

$$log_g s = 725397673982698875571265897837239945113153436946846419403 17 (mod\ p-1)$$

即为3333333333333333333333333333333333333333以g为底$mod\ p$的离散对数。

19.3 举一反三

随着全球信息化和网络化的发展，公钥密码体制被大量应用。目前投入应用的公钥密码体制的安全性都是基于一些难解的数学问题；其中，Diffie-Hellman 密钥交换体制安全性基础是有限域上的离散对数的计算困难性。1976 年，Diffie 和 Hellman 发表的论文首次提出了“公钥密码学”的概念，它是密码学历史上的一个重大成就，奠定了现代密码学的基础，文章提出了基于离散对数计算困难性这一数学难题的DH密钥交换体制，DH体制的出现，极大地推动了对离散对数问题的研究。

在密码学中，主要应用离散对数问题的如下特点：求解离散对数是困难的，而其逆运算指数运算是能够有效计算的。密码学中，许多公钥密码体制都是基于离散对数问题而构建的，其安全性也依赖于离散对数问题的难解性，ElGamal 是这类密码体制中第一个也是最为著名的密码体制。

针对离散对数问题的求解，除去穷举搜索以外，相关研究工作者也构建出了许多有效的求解方法，具体包括$Shanks$算法、$Pollard's\ \rho$离散对数算法、$Pollard's\ kangaroo$算法、$Pohlig-Hellman$算法、指数演算法，以及$CADO-NFS$工具计算离散对数，等等。

在一些密码学相关竞赛中，也常出现以离散对数问题求解为核心的试题。除去暴力的方法，如$BSGS$等离散对数求解方法能够起到较好的计算效果。值得注意的是，当离散对数问题的数据情况特殊时，部分算法的优势会明显体现，在合适的情况选择合适的算法将是有效的。

开源数学软件系统 SageMath 中，集成了许多现有的离散对数求解方法，包括$BSGS$算法、$Pollard's\ \rho$算法、$Pollard's\ kangaroo$算法，等等，使用起来十分便利。$Cado-nfs$是 C/C++中数域筛法（NFS）算法的完整实现，用于分解整数并计算有限域中的离散对数。$Cado-nfs$可用于求解离散对数问题，但值得注意的是，其求解得到的离散对数问题结果一般是$mod\ ell$的结果，和实际离散对数问题结果在$mod\ p-1$计算下是可能有出入的，需要确定$mod\ ((p-1)/ell)$的结果，而后通过中国剩余定理确定真实离散对数问题结果，也可改写剩余类通过遍历确定结果。

第20章 椭圆曲线上的离散对数

20.1 基本原理

椭圆曲线上的离散对数问题，简称$ECDLP$。1987 年，$Koblitz$利用椭圆曲线上点形成的$Abelian$加法群构造了$ECDLP$。实验证明，在椭圆曲线加密算法中采用 160bits 的密钥可与1024bits 密钥的RSA算法的安全性相当，且随着模数的增大，它们之间安全性的差距猛烈增大。因此，它可以提供一个更快、具有更小的密钥长度的公开密钥密码系统，备受人们的广泛关注，为人们提供了诸如实现数据加密、密钥交换、数字签名等密码方案的有力工具。

20.1.1 椭圆曲线上的离散对数

椭圆曲线密码体制使用的是有限域上的椭圆曲线，即变量和系数均为有限域中的元素。一个有限域的具体例子就是模p的整数域，其中p是一个素数。有限域$GF(p)$上的椭圆曲线是指满足方程

$$y^2 \equiv x^3 + ax + b(mod\ p)$$

的所有点(x,y)再加上一个无穷远点O构成的集合，其中，a,b,x和y均在有限域$GF(p)$上取值，p是素数。这里把该椭圆曲线记为$E_p(a,b)$。该椭圆曲线只有有限个点，其个数N由$Hasse$定理确定。

定理 20-1 ($Hasse$定理)设E是有限域$GF(p)$上的椭圆曲线，N是E上点的个数，则

$$p + 1 - 2\sqrt{p} \le N \le p + 1 + 2\sqrt{p}$$

当$4a^3 + 27b^2(mod\ p) \neq 0$时，基于集合$E_p(a,b)$可以定义一个$Abel$群，其加法规则与实数域上描述的代数方法一致。设$P,Q \in E_p(a,b)$，则

(1) $P + O = P$。

(2) 如果$P = (x,y)$，那么$(x,y) + (x,-y) = O$，即点$(x,-y)$是P的加法逆元，表示为$-P$。

(3) 设$P = (x_1,y_1)$和$Q = (x_2,y_2)$，$P \neq -Q$，则$S = P + Q = (x_3,y_3)$由以下规则确定：

$$x_3 \equiv \lambda^2 - x_1 - x_2(mod\ p)$$
$$y_3 \equiv \lambda(x_1 - x_3) - y_1(mod\ p)$$

式中

$$\lambda \equiv \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1}(mod\ p), & P \neq Q \\ \dfrac{3{x_1}^2 + a}{2y_1}(mod\ p), & P = Q \end{cases}$$

例 20-1 设$p = 11$，$a = 1$，$b = 6$，即椭圆曲线方程为

$$y^2 \equiv x^3 + x + 6(mod\ 11)$$

要确定椭圆曲线上的点，对于每个$x \in GF(11)$，首先计算$z \equiv x^3 + x + 6(mod\ 11)$，然后再判定$z$是否是模11的平方剩余(方程$y^2 \equiv z(mod\ 11)$是否有解)，若不是，则椭圆曲线上没有与这一$x$相对应的点；若是，则求出$z$的两个平方根。该椭圆曲线上的点如下表所示。

x	0	1	2	3	4	5	6	7	8	9	10
z	6	8	5	3	8	4	8	4	9	7	4
平方剩余	否	否	是	是	否	是	否	是	是	否	是

y			4	5		2		2	3		2
			7	6		9		9	8		9

只有$x = 2,3,5,7,8,10$时才有点在椭圆曲线上，$E_{11}(1,6)$是由上表中的点再加上一个无穷远点O构成，即

$$E_{11}(1,6) = \{O,(2,4),(2,7),(3,5),(3,6),(5,2),(5,9),(7,2),(7,9),(8,3),(8,8),(10,2),(10,9)\}$$

设$P = (2,7)$，计算$2P = P + P$。首先计算

$$\lambda \equiv \frac{3 \times 2^2 + 1}{2 \times 7}(mod\ 11) = \frac{2}{3}(mod\ 11) \equiv 8$$

于是

$$x_3 \equiv 8^2 - 2 - 2(mod\ 11) \equiv 5$$
$$y_3 \equiv 8 \times (2 - 5) - 7(mod\ 11) \equiv 2$$

所以$2P = (5,2)$。同样可以算出

$$3P = (8,3), 4P = (10,2), 5P = (3,6), 6P = (7,9),$$
$$7P = (7,2), 8P = (3,5), 9P = (10,9), 10P = (8,8),$$
$$11P = (5,9), 12P = (2,4), 13P = O$$

由此可以看出，$E_{11}(1,6)$是一个循环群，其生成元是$P = (2,7)$。

椭圆曲线上的离散对数问题$(ECDLP)$的定义如下，已知椭圆曲线E及其上的两个点P和Q，k为一个整数，且

$$Q = kP,$$

期望计算出符合条件的k。

将椭圆曲线中的加法运算与离散对数中的模乘运算相对应，将椭圆曲线中的乘法运算与离散对数中的模幂运算相对应，可以建立基于椭圆曲线的密码体制。在椭圆曲线加密中，点P称为基点，k为私有密钥，Q为公开密钥，则给定k和P，根据加法法则，计算Q很容易，但给定P和Q，求k非常困难。

20.1.2 椭圆曲线的群结构

一般的，用F_q表示包含q个元素的有限域，并用$\mathbb{Z}_n$表示n阶循环群。设E为定义在F_q上的椭圆曲线，设$q = p^m$，其中p为F_q的特征。如果p大于3，则$E(F_q)$是以下仿射方程在$F_q \times F_q$中所有解的集合

$$y^2 = x^3 + ax + b, \qquad (22.1)$$

其中$a, b \in F_q$，$4a^3 + 27b^2 \neq 0$，以及一个额外的点O(称作无穷远点)。如果$p = 2$，则$E(F_q)$的仿射方程为

$$y^2 + a_3 y = x^3 + a_4 x + a_6, \qquad (22.2)$$

其中$a_3, a_4, a_6 \in F_q$，$a_3 \neq 0$，且曲线的$j - invariant$等于0，以及

$$y^2 + xy = x^3 + a_2 x^2 + a_6, \qquad (22.3)$$

其中$a_2, a_6 \in F_q$，$a_6 \neq 0$，且曲线的$j - invariant$不等于0。在$E(F_q)$的点上定义由“切线及弦线方法”给定的自然加法，其涉及到F_q上的一些算术运算。在这种加法运算条件下，$E(F_q)$的点组成了秩为1或2的$Abelian$群。根据$Hasse$定理，群的阶为$q + 1 - t$，其中$|t| \leq 2\sqrt{q}$。群的结构为(n_1, n_2)，例如$E(F_q) \cong \mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2}$，其中$n_2 | n_1$，更进一步的有$n_2 | q - 1$。为简单起见，以下称$E(F_q)$为$F_q$上的椭圆曲线。以下引理确定了某一阶的椭圆曲线是否存在。

引理 20-1 在F_q上存在一个阶为$q + 1 - t$的椭圆曲线，当且仅当下列条件之一成立。

1) $t \not\equiv 0(mod\ p)$且$t^2 \leq 4q$。
2) m是奇数且下列条件之一成立：
   a) $t = 0$；
   b) $t^2 = 2q$且$p = 2$；

c) $t^2 = 3q$且$p = 3$。

3) m是偶数且下列条件之一成立：

a) $t^2 = 4q$；

b) $t^2 = q$且$p \not\equiv 1(mod\ 3)$；

c) $t = 0$且$p \not\equiv 1(mod\ 4)$。

设$\#E(F_q) = q + 1 - t$表示曲线的阶数。如果p能整除t，则称E为超奇异的。由前面的结果，当且仅当$t^2 = 0, q, 2q, 3q$或$4q$时，可以推导出E是超奇异的。下面的引理给出了超奇异曲线的群结构。

引理 20-2 设$\#E(F_q) = q + 1 - t$。

a) 如果$t^2 = q, 2q$或$3q$，那么$E(F_q)$是循环的。

b) 如果$t^2 = 4q$，那么$E(F_q) \cong \mathbb{Z}_{\sqrt{q}-1} \oplus \mathbb{Z}_{\sqrt{q}-1}$或$E(F_q) \cong \mathbb{Z}_{\sqrt{q}+1} \oplus \mathbb{Z}_{\sqrt{q}+1}$成立，具体取决于$t = 2\sqrt{q}$还是$t = -2\sqrt{q}$。

c) 如果$t = 0$且$q \not\equiv 3(mod\ 4)$，那么$E(F_q)$是循环的。如果$t = 0$且$q \equiv 3(mod\ 4)$，那么要么$E(F_q)$是循环的，要么$E(F_q) \cong \mathbb{Z}_{(q+1)/2} \oplus \mathbb{Z}_2$。

曲线E也可以看作是F_q的任意扩域K上的椭圆曲线；$E(F_q)$是$E(K)$的子群。$Weil$定理使得能够按照以下方式从$\#E(F_q)$计算$\#E(F_{q^k})$。令$t = q + 1 - \#E(F_q)$。那么$\#E(F_{q^k}) = q^k + 1 - \alpha^k - \beta^k$，其中$\alpha, \beta$是由$1 - tT + qT^2 = (1 - \alpha T)(1 - \beta T)$因式分解所确定的复数。

n阶挠点P是指满足$nP = O$的点。设$E(F_q)[n]$表示$E(F_q)$中n阶挠点的子群，其中$n \neq 0$。一般将$E(\overline{F_q})[n]$写成$E[n]$，其中$\overline{F_q}$表示F_q的代数闭包。如果n和q是互素的，则$E[n] \cong \mathbb{Z}_n \oplus \mathbb{Z}_n$。如果$n = p^e$，则要么$E[p^e] \cong \{O\}$(如果$E$是超奇异的)，要么$E[p^e] \cong \mathbb{Z}_{p^e}$($E$不是超奇异的)。

下面的引理提供了$E(F_q)$包含 $E(\overline{F_q})$中所有n阶挠点的充分必要条件。

引理 20-3 如果$\gcd(n, q) = 1$，那么$E[n] \subset E(F_q)$，当且仅当下列条件之一成立：

a) $n^2 | \#E(F_q)$；

b) $n | q - 1$；

c) 要么$\phi \in \mathbb{Z}$要么$\vartheta(t^2 - 4q/n^2) \subset End_{F_q}(E)$。

设n是一个与q互素的正整数。$Weil\ pairing$是以下函数

$$e_n : E[n] \times E[n] \to \overline{F_q}.$$

以下列出了该函数的一些有用特性。

1) 确定性：对于所有$P \in E[n]$，$e_n(P, P) = 1$。

2) 交替性：对于所有$P_1, P_2 \in E[n]$，$e_n(P_1, P_2) = e_n(P_2, P_1)^{-1}$。

3) 双线性：对于所有$P_1, P_2, P_3 \in E[n]$，$e_n(P_1 + P_2, P_3) = e_n(P_1, P_3)e_n(P_2, P_3)$，以及$e_n(P_1, P_2 + P_3) = e_n(P_1, P_2)e_n(P_1, P_3)$。

4) 非退化性：如果$P_1 \in E[n]$，那么$e_n(P_1, O) = 1$。如果对于所有$P_2 \in E[n]$都有$e_n(P_1, P_2) = 1$，那么$P_1 = O$。

5) 如果$E[n] \subseteq E(F_{q^k})$，那么对于所有$P_1, P_2 \in E[n]$都有$e_n(P_1, P_2) \in F_{q^k}$。

$Miller$开发了一种有效的概率多项式时间算法来计算$Weil\ pairing$。概率多项式算法，指的是一种随机算法，其预期运行时间受输入大小的多项式的限制。基于输入x的概率亚指数级算法，是指该随机算法的期望运行时间以$L(\alpha, x)$为界，其中$0 < \alpha < 1$(α为常数)，且有

$$L(\alpha, x) = \exp\Big((c + o(1))(\ln x)^{\alpha}(\ln \ln x)^{1-\alpha}\Big).$$

下面的引理提供了一种将椭圆曲线$E(F_q)$的元素划分为$\langle P \rangle$的陪集的方法，其中$\langle P \rangle$是由拥有最大阶的点P所生成的$E(F_q)$的子群。

引理 20-4 设$E(F_q)$为具有群结构(n_1, n_2)的椭圆曲线，设P为拥有最大阶n_1的一个元素。那么对于所有$P_1, P_2 \in E(F_q)$，P_1和P_2都在$\langle P \rangle$的同一陪集，当且仅当$e_{n_1}(P, P_1) = e_{n_1}(P, P_2)$。

下面的引理与上一引理相似，证明也相似。为了完整起见，将它也一并阐述。

引理 20-5 设$E(F_q)$是一条椭圆曲线，且使得$E[n] \subseteq E(F_q)$，其中n是与q的互素正整数。设$P \in E[n]$是一个n阶点。那么，对于所有$P_1, P_2 \in E[n]$，P_1和P_2都在$E[n]$内$\langle P \rangle$的同一陪集，当且仅当$e_n(P, P_1) = e_n(P, P_2)$。

证明：如果$P_1 = P_2 + kP$，那么显然有

$$e_n(P, P_1) = e_n(P, P_2) e_n(P, P)^k = e_n(P, P_2).$$

反过来，假设P_1和P_2在$E[n]$内$\langle P \rangle$的不同陪集。那么则有$P_1 - P_2 = a_1 P + a_2 Q$，其中$(P, Q)$是$E[n] \cong \mathbb{Z}_n \oplus \mathbb{Z}_n$的生成对，其中$a_2 Q \neq O$。如果$b_1 P + b_2 Q$是$E[n]$中的任意点，那么

$$e_n(a_2 Q, b_1 P + b_2 Q) = e_n(a_2 Q, P)^{b_1} e_n(Q, Q)^{a_2 b_2} = e_n(P, a_2 Q)^{-b_1}.$$

如果$e_n(P, a_2 Q) = 1$，那么根据e_n的非退化性，可以得到$a_2 Q = O$，这是一个矛盾。因此$e_n(P, a_2 Q) \neq 1$。最终，

$$e_n(P, P_1) = e_n(P, P_2) e_n(P, P)^{a_1} e_n(P, a_2 Q) \neq e_n(P, P_2).$$

在接下来的算法中，关键是能够在概率多项式时间内在椭圆曲线$E(F_q)$上均匀随机地选择点P。这可以通过以下方式完成：首先随机选择一个元素$x_1 \in F_q$。如果x_1是$E(F_q)$中某一点的x坐标，那么可以通过求解F_q中的寻根问题，找到y_1使得$(x_1, y_1) \in E(F_q)$。在概率多项式时间内求F_q上多项式的根，目前已有很多方法。然后，如果曲线有式(22.1)，则设$P = (x_1, y_1)$或$(x_1, -y_1)$，(相应的，如果曲线有式(22.2)或式(22.3)，分别设$P = (x_1, y_1)$或$(x_1, y_1 + a_3)$和$P = (x_1, y_1)$或$(x_1, y_1 + x_1)$)。根据$Hasse$定理，x_1是$E(F_q)$中某点的x坐标的概率至少为$1/2 - 1/\sqrt{q}$。注意，用上述方法选出一个阶为2的点的概率是选出任何其他点的概率的两倍；这没有问题，因为阶为2的点至多有三个。

最后，给出如下可能提供参考的一些引理。

引理 20-6 设G是一个群，$\alpha \in G$。设$n = \sum_{i=1}^{k} p_i^{\beta_i}$为$n$的素因数分解，那么$\alpha$是的阶为$n$当且仅当

a) $\alpha^n = 1$，且

b) 对于每个i，$1 \leq i \leq k$，有$\alpha^{n/p_i} \neq 1$。

引理 20-7 设G是结构为(cn, cn)的$Abelian$群。如果从G中均匀随机地选择元素$\{\alpha_i\}$，则元素$\{c\alpha_i\}$在结构为(n, n)的G的子群元素上是均匀分布的。

20.1.3 $SSSA$攻击

$SSSA$攻击，一般攻击对象为有理点群的阶等于有限域大小的特殊椭圆曲线，即$|E| = p$，这类特殊曲线一般称为“$anomalous\ curve$”。$SSSA$攻击将$ECDLP$问题约化到有限域加法群的DLP，使得该类$ECDLP$问题存在多项式时间算法。通过构造映射将F_p上的椭圆曲线提升到p的扩域上，再通过模p的约化获得$E(F_p)$的离散对数解。

针对椭圆曲线上的点的计算，点的x, y坐标在计算时总是模p，此处尝试考虑模p^k的情况，其中k为大于等于2的整数。针对每一个原椭圆曲线对应有限域上的整数，均可按照p进制的方式进行重写，如$a_0 + a_1 p + a_2 p^2 + \cdots$，其中$a_i$为$[0, p-1]$的整数，那么，在整数进行模$p^k$计算时仅需考虑前$k$项。虽然此级数不会收敛，但是在进行模$p^k$的情况后，除去前$k$项的项均会变为0，因此可不用先考虑其收敛问题。至于一个有理数试图表示成类似p进制的方式，则可写成$a_{-k} p^{-k} + \cdots + a_{-2} p^{-2} + a_{-1} p^{-1} + a_0 + a_1 p + a_2 p^2 + \cdots$，此处仅考虑了有限小数。对应的，若是进行模$p^{-k}$的计算，仅需将$p$的幂次大于等于$-k$的项删去再计算剩余项即可。由此，从原本的椭圆曲线$y^2 = x^3 + ax + b (mod\ p)$，可以扩展至$y^2 = x^3 + ax + b (mod\ p^2)$，乃至扩展为$y^2 = x^3 + ax + b (mod\ p^k)$，称原本模$p$的曲线为$E(F_p)$，扩展后的曲线为$E(Q_p)$。

不难发现，对于一个点(x_0, y_0)，假设它满足 $y^2 = x^3 + ax + b (mod\ p)$，但它不一定满足 $y^2 = x^3 + ax + b (mod\ p^2)$。由$Hansel's\ Lemma$可知，即使满足$y^2 = x^3 + ax + b (mod\ p)$的点$(x_0, y_0)$不再满足 $y^2 = x^3 + ax + b (mod\ p^2)$，但其依旧存在。按照具体例子来看，(1,1)是满足

$y^2 = x^3 + 14x + 1\ (mod\ p)(p = 3)$的，但不满足$y^2 = x^3 + 14x + 1(mod\ p^2)$，假设$(1 + kp, 1)$满足后式，代入可得$17kp \equiv -15(mod\ p^2)$，可以发现，在$[0, p-1]$可找到唯一的$k$，使得上式满足。也就是说$(1 + kp, 1)$满足$y^2 = x^3 + 14x + 1(mod\ p^2)$。依次类推，已知$y^2 = x^3 + ax + b(mod\ p^2)$上的一点，可以求得对应的在$y^2 = x^3 + ax + b(mod\ p^3)$上的一个点，然后一直往上推。也就是说$E(F_p)$上所有的点都可以扩展成$E(Q_p)$上的点。从$E(Q_p)$向$E(F_p)$转变，只需进行模$p$计算即可，即幂次大于等于1的均舍去。对于$p$的负数次幂的数，将其视为$E(F_p)$下的无穷远点$O$。这种$E(F_p)$与$E(Q_p)$之间点的转换，不会影响正常椭圆曲线上的加法，考虑原椭圆曲线上的离散对数问题，在已知$E(F_p)$上点P和Q的情况，且存在n使得 Q= nP，欲求解n。现可将P和Q点扩展为$E(Q_p)$上的P'和Q'点，然后求解满足$Q' - nP' = O'$的n值，其中O'是$E(Q_p)$下的某个非整数点。

考虑$y^2 = x^3 + ax + b$，设$w = -\frac{1}{y}$，$z = -\frac{x}{y}$，则原$y^2 = x^3 + ax + b$式可变换为$w = z^3 + azw^2 + bw^3$ 。将上式代入上式右侧替换掉w，可不断代入，此过程中将产生新的不包含w仅包含z的项，那么在不断代入的过程中，最后原式右侧是一个不包含w仅包含z的式子，可记为$f(z)$。且右侧关于z的最小幂次为z^3，由此可知必有$z^3 | w$。由此推导出了原椭圆曲线的一个参数式$w = f(z)$。

在进行以上计算得到$w = f(z)$后，可用$P(Z)$表示原椭圆曲线上的点，以参数z即可确定该点。现在需要找到一个映射函数ϕ。使得$P(z_1) + P(z_2) = P(z_3) \Leftrightarrow \phi(z_1) + \phi(z_2) = \phi(z_3)$。那么就有$Q = nP \Leftrightarrow \phi(Q) = n\phi(P)$。

定义$F(z_1, z_2)P^{-1}\big(P(z_1) + P(z_2)\big)$，参数为$z_1$的点加上参数为$z_2$的点，可得到参数为$F(z_1, z_2)$的点，即$F(x, y) = z \Leftrightarrow \phi(x) + \phi(y) = \phi(z)$。按照$SSSA$攻击研究者的推导，其构建了适用于椭圆曲线情况下的函数。

$$\phi_F(x) = \int \frac{1}{1 + x(\dots)} dx = x + x^2(\dots)$$

针对这个ϕ函数，其不一定收敛。在Q_p域下，如果$p | z$，那么对应ϕ的级数一定会收敛。由定义可知，$z = -\frac{x}{y}$，在$E(Q_p)$中，假设x和y在p进制的表示中幂次最小的分别为p^{-a}和p^{-b}，其中a和b为正整数。根据椭圆曲线式为$y^2 = x^3 + ax + b$，那么式左和式右的最小幂次分别为p^{-2b}和p^{-3a}，则$3a = 2b$，故$a < b$，则$z = -\frac{x}{y}$一定为p的倍数。对于非整数点$P(z)$，则有$\phi(z) = z + z^2(\dots) \equiv z(mod\ p^2)$。

ϕ函数的确定，将方便进行离散对数的计算。已知$Q' - nP' = O'$，此处无法直接计算，在式子两侧乘上p后，则有$pQ' - npP' = pO'$，则$\phi(pQ)' - (pP') = p\phi(O')$，而$\phi\big((P(z)\big) \equiv z(mod\ p^2)$，故$n \equiv \frac{\phi(pQ')}{\phi(pP')} \equiv \frac{P^{-1}(pQ')}{P^{-1}(pP')} (mod\ p)$。

$SSSA$攻击算法具体代码如下。

```
def SmartAttack(P,Q,p):
    E = P.curve()
    Eqp = EllipticCurve(Qp(p,2),[ZZ(t)+randint(0,p)*p for t in E.a_invariants() ])

    P_Qps = Eqp.lift_x(ZZ(P.xy()[0]), all=True)
    for P_Qp in P_Qps:
        if GF(p)(P_Qp.xy()[1]) == P.xy()[1]:
            break

    Q_Qps = Eqp.lift_x(ZZ(Q.xy()[0]), all=True)
    for Q_Qp in Q_Qps:
        if GF(p)(Q_Qp.xy()[1]) == Q.xy()[1]:
            break

    p_times_P = p*P_Qp
    p_times_Q = p*Q_Qp
```

```
    x_P,y_P = p_times_P.xy()
    x_Q,y_Q = p_times_Q.xy()

    phi_P = -(x_P/y_P)
    phi_Q = -(x_Q/y_Q)
    k = phi_Q/phi_P
    return ZZ(k)
```

20.1.4 $Pohlig-Hellman$算法

$Pohlig-Hellman$攻击，一般攻击对象需满足基点P的阶可以较为容易的因数分解，且不包含较大的素数。$Pohlig-Hellman$攻击的主要思想是对基点P的阶数进行分解，把对应的离散对数问题转移到了各个因子条件下的离散对数问题，最终利用中国剩余定理进行求解。

对于问题$Q_A=[n_A]P$，其中P为基点，n_A是要求解的私钥。

首先求得P的阶o，并将o进行分解，设$o=p_1^{e_1}*p_2^{e_2}*\dots\dots*p_r^{e_r}$，对于$i$属于$[1,r]$，计算$n_i=n_A\ mod\ p_i^{e_i}$，得到下面的这一系列等式：

$$n_A\equiv n_1\ mod\ p_1^{e_1}$$
$$n_A\equiv n_2\ mod\ p_2^{e_2}$$
$$\dots$$
$$n_A\equiv n_r\ mod\ p_r^{e_r}$$

根据中国剩余定理，可唯一确定n_A，故求n_A转换成了求解$(n_1,n_2,\dots,n_{e_r})$。

接下来，可以将n_i设为p_i表示的多项式，如下：

$$n_i=z_0+z_1p_i+z_2p_i^2+\dots+z_{e_r-1}p_i^{e_r-1}$$

由此，确定$(z_0,z_1,z_2,\dots,z_{e_r-1})$，即可确定$n_i$。

设$\bar{P}=\frac{o}{p_i}P$，$\bar{Q}=\frac{o}{p_i}Q$，易知$\bar{P}$的阶为p_i。由$n_i\equiv n_A\ mod\ p_i^{e_i}$，设$n_A=n_i+mp_i^{e_i}$，$m$为正整数，则$\bar{Q}=n_i\bar{P}=(z_0+z_1p_i+z_2p_i^2+\dots+z_{e_r-1}p_i^{e_r-1})\bar{P}=z_0\bar{P}$，通过求$\bar{P}$与$\bar{Q}$的离散对数即为$z_0$，同理可以依次算出$z_1,z_2,\dots,z_{e_r-1}$，最后求出$n_i$，再根据中国剩余定理即可求出$n_A$。

$Pohlig-Hellman$攻击算法具体代码如下。

```
def pohlig_hellman(P,Q,p):
    E = P.curve()
    n_P = P.order()
    tmp = factor(n_P)
    print(tmp)
    primes = []
    i = 0
    while(i<len(tmp)):
        primes += [tmp[i][0]^tmp[i][1]]
        i += 1
    print(primes)
    dlogs = []
    for fac in primes:
        t = int(n_P) // int(fac)
        dlog = discrete_log(t*Q,t*P,operation="+")
        dlogs += [dlog]
        print("factor: ",str(fac), "Discrete Log: ", str(dlog))
    res = crt(dlogs,primes)
return res
```

20.1.5 $Pollard-rho$算法

$Pollard-rho$攻击的基本原理为构建一个可迭代的点R，令$R_{i+1}=f(R_i)=a_iP+b_iQ$，通过不断迭代寻找点对$R_i=R_{2i}$，即可得到私钥$n_A=\frac{a_i-a_{2i}}{b_{2i}-b_i}$。此处，为了提高碰撞率，会将椭圆曲

线分为三个点集来构造迭代函数f。

令$f: S \to S$是一个S到它自身的映射。S的大小是n。对于一个随机的值$x_0 \in S$，对于每个$i \geq 0$，计算$x_{i+1} = f(x_i)$。对于每一步来说$x_{i+1} = f(x_i)$都是一个确定的函数，就得到了一个确定的随机序列$x_0, x_1, x_2, \cdots$。

因为S是有限的，最终会得到$x_i = x_j$因此$x_{i+1} = f(x_i) = f(x_j) = x_{j+1}$。因此，序列$x_0, x_1, x_2, \cdots$将变成一个循环。目标是在上述的序列中找到一个碰撞，就是找到i, j使得$i \neq j$并且$x_i = x_j$。

为了寻找一个碰撞，使用$Floyd's$发现算法，给定(x_1, x_2)，计算(x_2, x_4)，然后是(x_3, x_6)等等。例如给定(x_i, x_{2i})，就可以计算$(x_{i+1}, x_{2i+2}) = \big(f(x_i), f\big(f(x_{2i})\big)\big)$。当发生碰撞的时候有$x_m = x_{2m}$。此时$m = O(\sqrt{n})$。(此处计算的是当$f$为完全随机函数时的时间复杂度)

对于椭圆曲线离散对数问题，将群S人为划分成三个组S_1, S_2, S_3。假设$1 \in S_2$，然后定义下面的随机序列

$$R_{i+1} = f(R_i) = \begin{cases} Q + R_i, & R_i \in S_1 \\ 2R_i, & R_i \in S_2 \\ P + R_i, & R_i \in S_3 \end{cases}$$

$$a_{i+1} = \begin{cases} a_i, & R_i \in S_1 \\ 2a_i \bmod N, & R_i \in S_2 \\ a_i + 1, & R_i \in S_3 \end{cases}$$

$$b_{i+1} = \begin{cases} b_i + 1, & R_i \in S_1 \\ 2b_i \bmod N, & R_i \in S_2 \\ b_i, & R_i \in S_3 \end{cases}$$

然后初始化参数$(x_0, a_0, b_0) = (1,0,0)$，可以知道对所有的$i$有$log_g(x_i) = a_i + b_i$，$log_g(h) = a_i + b_i x$。使用$Floyd's$算法，找到$x_m = x_{2m}$。这样就计算出了$x = \frac{a_{2m} - a_m}{b_m - b_{2m} \bmod n}$。

$Pollard - rho$攻击算法具体代码如下。

```
def Block(num):
    global csz
    return int(num[0]) % csz

def F():
    global Fm, A1, B1, r
    i = Block(Fm)
    if i < sz :
        A1 += da[i]
        B1 += db[i]
        Fm += stp[i]
    else:
        A1 <<= 1
        B1 <<= 1
        Fm += Fm
    if A1 >= r :
        A1 -= r
    if B1 >= r :
        B1 -= r

def check():
    global Fm, check_mod
    return int(Fm[0]) % check_mod == 0

def insert():
    global A1, B1, Fm, list_A1, list_B1, list_Fm
    list_A1 += [A1]
    list_B1 += [B1]
    list_Fm += [Fm]
```

```
30.
31. def count_and_query():
32.     global A1, B1, A2, B2, Fm, list_A1, list_B1, list_Fm
33.     length =len(list_Fm)
34.     i = 0
35.     while(i < length):
36.         if Fm == list_Fm[i] and B1 != list_B1[i]:
37.             A2 = list_A1[i]
38.             B2 = list_B1[i]
39.             return 1
40.         i += 1
41.     A2 = A1
42.     B2 = B1
43.     return 0
44.
45. def pollard_rho():
46.     global Fm, A1, B1, A2, B2
47.     cnt = 0
48.     msz = 0
49.     while(1):
50.         if (check()):
51.             if (count_and_query()):
52.                 if (B1 != B2):
53.                    break
54.                 A1 = randint(1,r-1)
55.                 B1 = randint(1,r-1)
56.                 Fm = A1 * P + B1 * R
57.             else:
58.                 insert()
59.         F()
60.     return (A2 - A1 + r) * inverse_mod((B1 - B2 + r), r) % r
```

20.1.6 袋鼠算法

袋鼠算法，也称$Pollard\ Lambda$算法，其定义了两个点在解空间里面各自跳跃，其中一点的参数是确定的，而另一点的参数则是由题目要求的。第一个点每次跳跃之后都会做一个标记，如果后者某次跳跃碰到了这个陷阱，则表明它们的参数是一致的。这样就可以使用第一点的参数来推导出第二点的参数。即通过两条不同的路径经过变化得到一个交点的过程。

已知$0 < a \le x \le b < n = g.order$，首先定义两点$tame$和$wild$，其中$tame$的初始位置是$T = \left[\frac{a+b}{2}\right]P$，$wild$初始位置为$W = [n_A]P$，之后定义点跳跃的集合

$$S = \left\{PS_0, PS_1, \cdots, PS_r; S_i = O\left((b-a)^{\frac{1}{2}}\right)\right\}, r = O(\log(b-a))$$

以及定义映射$v: G \to \{0,1,\ldots,\mathrm{r}\}$

对于$k = 0,1,\ldots$:

$$tame: t_{k+1} = t_k * s_i, \text{其中} i = v(t_k)$$

$$wild: w_{k+1} = w_k * s_i, \text{其中} i = v(w_k)$$

为了计算需要，在跳跃的过程中记录下每次跳跃的距离：

$$D_0(T) = D_0(W) = 0$$

$$D_{k+1}(T) = D_k(T) + s_i$$

$$D_{k+1}(W) = D_k(W) + s_i$$

在某几次跳跃之后，$wild$落入了$tame$的标记，二者相遇，则可以得知：

$$t_k = w_l$$

$$\left[\frac{a+b}{2}\right]P * D_k(T) = [n_A]P * D_l(W)$$

进而可求出n_A的值。

袋鼠算法具体代码如下：

```
hashValue = 0
def Hash(P):
    if P == 0:
        return 1
    return int(P.xy()[0]) % hashValue +int(P.xy()[1]) % hashValue+ 1
def pollardKangaroo(P, Q, a, b):
    global hashValue
    hashValue = math.ceil(sqrt((b-a))/2)
    # Tame 的袋鼠迭代:
    xTame, yTame = 0, b * P
    for i in range(0,math.ceil(0.7*sqrt(b-a))):
        xTame += Hash(yTame)
        yTame += Hash(yTame) * P
    # yTame == (b + xTame) * P 等式必须成立
    # Wild 的袋鼠迭代:
    xWild, yWild = 0, Q
    for i in range(0, math.ceil(2.7*sqrt(b-a) ) ):
        xWild += Hash(yWild)
        yWild += Hash(yWild) * P
        if yWild == yTame:
            return b + xTame - xWild
    # 二者未相遇，结果未找到:
    return 0
```

20.1.7 *MOV*攻击

*MOV*攻击，将有限域F_q上曲线E上的椭圆曲线对数问题简化为F_q的适当扩展域F_{q^k}上的离散对数问题。这是通过在P生成的E的子群$\langle P\rangle$和F_{q^k}中第n个单位根的子群(n表示P的阶)之间建立同构来实现的，同构由*Weil*双线性对给出。

双线性配对是指在三个素数p阶乘法循环群G_1、G_2和G_T下，定义映射$e: G_1 \times G_2 \to G_T$，满足如下三个性质：

(1) 双线性：对于$\forall g_1 \in G_1, g_2 \in G_2$，$\forall a, b \in Z_p$，有$e(ag_1, bg_2) = e(g_1, g_2)^{ab}$；

(2) 非退化性：$\exists g_1 \in G_1, g_2 \in G_2$，使得$e(g_1, g_2) \neq 1_{G_T}$；

(3) 可计算性：存在有效的算法，对于$\forall g_1 \in G_1, g_2 \in G_2$均可计算$e(g_1, g_2)$。

*Weil*对除满足上述性质外，还对于超奇异椭圆曲线有一些额外的性质，可以把超奇异椭圆曲线$E(F_p)$作为G_1、G_2，把F_{p^k}作为G_T构建对称双线性映射，用于该类椭圆曲线的攻击。

设$E(F_q)$为有限域F_q上具有群结构$\mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2}$的椭圆曲线，其中$n_2 | n_1$。已知$E(F_q)$的定义方程，可以使用*Schoof*算法在多项式时间内计算$\#E(F_q)$。在给定$\gcd(\#E(F_q), q-1)$的整数因式分解的情况下，利用*Miller*给出的算法可以在概率多项式时间内确定n_1和n_2。进一步假设$\gcd(\#E(F_q), q-1) = 1$，使得$E[n_1]$同构于$\mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2}$。

设$P \in E(F_q)$为n阶点，其中n整除n_1，设$R \in E(F_q)$。假设n是已知的。椭圆曲线对数问题如下：给定P和R，确定唯一的整数l，$0 \leq l \leq n-1$，使得$R = lP$，前提是存在这样的整数。

由于$e_n(P, P) = 1$，从有限域椭圆曲线理论引理 20-4 推导出$R \in \langle P\rangle$，当且仅当$nR = O$，$e_n(P, R) = 1$，这些条件可以在概率多项式时间内检验。因此，将假定$R \in \langle P\rangle$。

首先描述在P具有最大阶的情况下，通过求解F_q域本身的一个离散对数问题来获得关于l的部分信息的一种算法。

算法 20-1：

输入：元素$P \in E(F_q)$，最大阶为n_1，以及$R = lP$。

输出：整数$l' \equiv l (mod\ n')$，其中n'是n_2的除数。

1) 随机选一个点$T \in E(F_q)$。

2) 计算$\alpha = e_{n_1}(P,T)$以及$\beta = e_{n_1}(R,T)$。

3) 计算l'，F_q中β以α为底的离散对数。

算法 20-1 正确地计算出$l' \equiv l(mod\ n')$，其中n'是n_2的某个除数。

由于在$E(F_q)$中有n_2个$\langle P\rangle$的陪集，从有限域椭圆曲线理论引理 20-4 可推导出$n' = n_2$的概率是$\frac{\phi(n_2)}{n_2}$。然而，如果n_2比n_1小(正如预期的那样，如果曲线是随机选择的，$n_2|\gcd(n_1, q-1)$，那么这种方法不能提供关于l的任何重要信息。而后将描述一种计算l模n的技术。

设k为$E[n] \in (F_{q^k})$的最小正整数；很明显，存在这样一个整数k。那么，存在$Q \in E[n]$，使得$e_n(P,Q)$是整体的原始n次根。设Q是$E[n]$中的一个点，使$e_n(P,Q)$是一个数的原始n次方根。那么，若由$f: R \mapsto e_n(P,Q)$定义$f: \langle P\rangle \to \mu_n$，则$f$是群同构的。

现在可以描述在有限域内将椭圆曲线对数问题简化为离散对数问题的方法。

算法 20-2：

输入：元素$P \in E(F_q)$，阶为n，以及$R \in \langle P\rangle$。

输出：整数l，使得$R = lP$。

1) 确定最小整数k，使得$E[n] \subseteq E(F_{q^k})$。

2) 找到$Q \in E[n]$，使得$\alpha = e_{\mathrm{n}}(P,Q)$的阶为$n$。

3) 计算$\beta = e_n(R,Q)$。

4) 计算l，F_{q^k}中β以α为底的离散对数。

注意，该算法的输出正确是由于

$$\beta = e_{\mathrm{n}}(lP,Q) = e_{\mathrm{n}}(P,Q)^l = \alpha^l.$$

由于k一般是指数级大小的，上述的约简一般需要指数级时间($\ln q$)。算法 20-2 也是不完整的，因为没有提供确定k和找到点Q的方法。对于超奇异椭圆曲线，将有特殊的确定方法。

设$E(F)$是F_q上$q+1-t$为阶的超奇异椭圆曲线，设$q = p^m$。由有限域椭圆曲线理论引理 20-1 和 20-2 可知，E属于以下一类曲线：

(1) $t = 0$且$E(F_q) \cong \mathbb{Z}_{q+1}$。

(2) $t = 0$且$E(F_q) \cong \mathbb{Z}_{(q+1)/2} \oplus \mathbb{Z}_2$(且$q \equiv 3(mod\ 4)$)。

(3) $t^2 = q$(且m是偶数)。

(4) $t^2 = 2q$(且$p = 2$，m是奇数)。

(5) $t^2 = 3q$(且$p = 3$，m是奇数)。

(6) $t^2 = 4q$(且m是偶数)。

设P是$E(F_q)$中的n阶点。因为$n_1|(q+1-t)$且$p|t$，则有$\gcd(n_1, q) = 1$。通过应用$Weil$定理和有限域椭圆曲线理论引理 20-2，可以很容易地确定$E[n_1] \subseteq E(F_{q^k})$的最小正整数$k$，因此$E[n] \subseteq E(F_{q^k})$。现在对超奇异曲线的约简进行详细的描述。

算法 20-3：

输入：超奇异曲线$E(F_q)$中的n阶元素P，以及$R \in \langle P\rangle$。

输出：整数l，使得$R = lP$。

1) 确定k和c。

2) 选择随机点$Q' \in E(F_{q^k})$，设$Q = (cn_1/n)Q'$。

3) 计算$\alpha = e_{\mathrm{n}}(P,Q)$和$\beta = e_n(R,Q)$。

4) 计算F_{q^k}中β以α为底的离散对数l'。

5) 检查是否有$l'P = R$。如果是那么$l = l'$即为所求。否则，α的阶必须比n小，所以转至 2)。

注意，根据有限域椭圆曲线理论引理 20-7，Q是$E[n]$中的一个随机点。还要注意，域元素a的阶为n的概率是$\phi(n)/n$。这源于有限域椭圆曲线理论引理 20-5 和F_{q^k}中有$\phi(n)$个n阶元素，$E[n]$中有n个$\langle P\rangle$陪集的事实。

如果$E(F_q)$是一个超奇异曲线，那么将在$E(F_q)$椭圆曲线的对数问题化简为在F_{q^k}的离散对数问题是一个概率多项式时间$(\ln q)$的化简。

注意到，当q和k都趋于无穷时，对于F_{q^k}域的离散对数问题是否存在亚指数级算法是未知的。亚指数级算法对于$q=2$，$q=p$且$k=1$，$q=p$且$k=2$，以及$q=p$且$\log p < n^{0.98}$的情况，通过严格证明可证明其$L[1/2,q]$的期望运行时间。对于$q=2$或q为固定素数的情况，启发式期望运行时间为$L[1/3,q]$的亚指数级算法是有效的，对于$q=p$且$k=1$，以及$q=p$且k固定的情况，启发式期望运行时间为$L[1/2,q]$的亚指数级算法是有效的(后者在$k=2$的情况下描述，但适用于k固定的情况)。对于$q=p$且$k=1$，以及$q=p$且k固定的情况，启发式期望运行时间为$L[1/3,q]$的算法，但这些目前还不实用。

注意，算法 20-3 的步骤 4)中解决的F_{q^k}中的离散对数问题有一个阶为n的基元素α，其中$n<q^k-1$。上述计算有限域离散对数的概率亚指数级算法要求基是本源的。利用这些算法，得到如下结果。

设P为超奇异椭圆曲线$E(F_q)$上的n阶元素，设$R=lP$为$E(F_q)$上的一点。如果q是素数，或者q是素数幂次$q=p^m$，其中p是固定的，那么新算法可以在概率亚指数级时间内确定l。

在实际求解椭圆曲线对数问题时，首先要分解n，利用这种分解，可以很容易地检查α的阶数。因此，为了求Q，反复在$E[n]$中随机选取点，直到α有阶数n为止。这就避免了需要解多个离散对数的可能性。但是请注意，这种修改后的约简不同于算法 20-3 中描述的约简，特别是不再是有限域离散对数问题的概率多项式时间约简。

之前改进的算法的主要步骤是对q^k-1进行因式分解，并在最后阶段计算F_{q^k}中的离散对数。用于分解整数n的数域筛法的预期运行时间为$L[1/3,n]$。因此，该算法的预期运行时间是$L[1/2,q^k]$或$L[1/3,q^k]$，这取决于F_{q^k}中离散对数问题的最佳算法运行时间。

得出对于超奇异曲线，椭圆曲线离散对数问题比以往认为的更容易处理的结论。

MOV攻击算法具体代码如下：

```
def getLog(beta,alpha):
    return discrete_log_rho(beta,alpha,alpha.multiplicative_order(),operation="*")
def MOV(P,Q,p,k):
    # 生成原 EC
    F1 = GF(p)
    r = F1.order() # 阶
    E1 = EllipticCurve(F1, [0, 0, 0, a, b])
    # EC 的阶
    n = E1.order()
    print("E1 : ",E1)
    # 将 EC(Fp) 映射到 EC(Fp^k)
    F2 = GF(p^k)
    phi = Hom(F1, F2)(F1.gen().minpoly().roots(F2)[0][0])
    E2 = EllipticCurve(F2 ,[0, 0, 0, a, b])
    print("n = ", n)
    # 两个点映射到 EC(Fp^k)上去
    P1 = E1(P)
    Q1 = E1(Q)
    P2 = E2(phi(P1.xy()[0]), phi(P1.xy()[1]))
    Q2 = E2(phi(Q1.xy()[0]), phi(Q1.xy()[1]))
    # cn
    cn1 = E2.gens()[0].order()
    coeff = ZZ(cn1 // n)
    print("cn1 = ",cn1)
    print("coeff = ", coeff)
    # 倍点
    while (1):
        Q = coeff * E2.random_point()
        print(Q.order() == n)
```

```
        if (Q.order() == n):
            print("Q = ",Q)
            break
    alpha = P2.weil_pairing(Q, n)
    beta = Q2.weil_pairing(Q, n)
    print("alpha = ", alpha)
    print("beta = ", beta)
    print(alpha.multiplicative_order())
    print(beta.multiplicative_order())
    d = getLog(beta,alpha)
    return d
```

20.2 示例分析

以第七届全国高校密码数学挑战赛-赛题三（椭圆曲线加密体制破译）为例进行椭圆曲线离散对数分析。该赛题中所使用的椭圆曲线加密体制(ECC)描述如下：

公共参数设定：

(1) 选择一个大素数p(在具体赛题中，选择p为 160 比特左右)；

(2) 选择一条定义在F_p上的椭圆曲线$E: y^2 = x^3 + ax + b$，以及在椭圆曲线E上的有理点P作为基点。

公钥生成：

(1) $Alice$选择私钥n_A，之后计算$Q_A = [n_A]P$；

(2) $Alice$公布自己的公钥Q_A。

加密过程：

(1) Bob将明文信息m通过某种方式(嵌入方式下面会详细说明)嵌入到椭圆曲线上的一个点$M \in E(F_p)$；

(2) 每次加密时，加密者Bob固定选择一个 160 比特的随机数k(部分随机数k由某一特定的随机数发生器生成)，计算

$$C_1 = [k]P \in E(F_p),$$
$$C_2 = M + [k]Q_A \in E(F_p);$$

(3) Bob发送C_1和C_2给$Alice$。

解密过程：

(1) $Alice$收到C_1和C_2后，使用自己的私钥计算$M = C_2 - [n_A]C_1 \in E(F_p)$；

(2) $Alice$根据M恢复明文消息m。

明文嵌入：

赛题中，加密者Bob每一次需要加密的明文消息包含 16 个明文字符以及分配给该消息的对应编号。解密者$Alice$通过自己的私钥解密得到所有明文信息和相应编号后，按照正确编号排序所有明文消息，可以恢复出Bob传输的完整有意义的消息。Bob每次加密时的消息嵌入规则如下：

(1) 将该次明文消息的 16 个明文字符转为 ASCII 码M_1，共计 128 比特；

(2) 将编号信息转为 ASCII 码(需要使用 8 比特)并添加在M_1尾部，得到M_2，此时M_2为 136 比特；

(3) 在M_2后再填充0变为M_3，使得M_3的比特长度达到 160 比特；

(4) 把M_3看作F_p中的元素，考虑$M_3, M_3 + 1, M_3 + 2, \cdots$，直到某一个最小非负整数$i$使得$x_M = M_3 + i$满足：${x_M}^3 + ax_M + b$在有限域$F_p$中等于某个元素的平方；

(5) 令$M = (x_M, y_M) \in E(F_p)$，其中$y_M$满足

$${y_M}^2 = {x_M}^3 + ax_M + b \bmod p,$$

且$y_M < \frac{p}{2}$。

赛题的问题描述如下：

已知条件：

给定椭圆曲线加密体制中每次加密所使用的椭圆曲线$E(F_p)$的基本参数。有理点$P \in E(F_p)$作为基点和公钥$Q_A = [n_A]P$的坐标也被给定，每次加密后的密文C_1和C_2也被给定。

求解目标：

1. 已知点$P \in E(F_p)$和$Q_A = [n_A]P$的坐标信息，求解出私钥n_A。
2. 恢复出与密文C_1和C_2所对应的明m。

注意：赛题中的点均使用点压缩技术表示。

20.2.1 *SSSA*攻击破解$T1$

题目要求：赛题第一题给出如下已知信息：

$p = 0xb0000000000000006c5b40000000000010ad7f77$；

$A = 0x7cbc14e5e0a72f05864385397829cbc15a2fe1d6$；

$B = 0xa9cbc14e5e0a72f0bc20f05397829cbc24fd94a8$；

$P = [0x725c5b2943cc60511e0ff0dc2caa3d5b718c5453,0]$；

$Q_A = [0x2a7e3b1a3960ee48d5e74dbb59859e433ead2dc0,1]$；

$C_1 = [0x5962af303c964f4b538ea857524bd13b8c84c2fb,0]$；

$C_2 = [0x9ff7a0335bb9ad42c5d95d4e956f6410483001a2,0]$。

题目分析：针对第一题，可以发现椭圆曲线的阶等于有限域参数p，即$|E| = p$，可见，本题的椭圆曲线是一种特殊曲线，称为“*anomalous curve*”，*SSSA*攻击方法可以很轻松的求解这种椭圆曲线上的离散对数问题。

解题代码：

```
# 第一题
import base64
def SmartAttack(P,Q,p):
    E = P.curve()
    Eqp = EllipticCurve(Qp(p,2),[ZZ(t)+randint(0,p)*p for t in E.a_invariants() ])

    P_Qps = Eqp.lift_x(ZZ(P.xy()[0]), all=True)
    for P_Qp in P_Qps:
        if GF(p)(P_Qp.xy()[1]) == P.xy()[1]:
            break

    Q_Qps = Eqp.lift_x(ZZ(Q.xy()[0]), all=True)
    for Q_Qp in Q_Qps:
        if GF(p)(Q_Qp.xy()[1]) == Q.xy()[1]:
            break

    p_times_P = p*P_Qp
    p_times_Q = p*Q_Qp

    x_P,y_P = p_times_P.xy()
    x_Q,y_Q = p_times_Q.xy()

    phi_P = -(x_P/y_P)
    phi_Q = -(x_Q/y_Q)
    k = phi_Q/phi_P
    return ZZ(k)

p = 0xb0000000000000006c5b40000000000010ad7f77
a = 0x7cbc14e5e0a72f05864385397829cbc15a2fe1d6
b = 0xa9cbc14e5e0a72f0bc20f05397829cbc24fd94a8
E = EllipticCurve(GF(p), [a, b])
```

```
32.
P=E(0x725c5b2943cc60511e0ff0dc2caa3d5b718c5453,884597905704094447395288637637997597425208728878)
33.
Q=E(0x2a7e3b1a3960ee48d5e74dbb59859e433ead2dc0,5110095460659830189491231230822116095672081457765)
34. while(1):
35.     k = SmartAttack(P, Q, p)
36.     if(P*k == Q):
37.         print('%#x'%k)
38.         break
39. print(P*k == Q)
40.
41.
C1=E(0x5962af303c964f4b538ea857524bd13b8c84c2fb,66181166109242058805343157518131582581853167500 8)
42.
C2=E(0x9ff7a0335bb9ad42c5d95d4e956f6410483001a2,46025205270261831185645018230341348738017785334)
43. M=C2-C1*k
44. print('%#x'%M[0])
45. #M[0]=0x6c6520737465702e204974206d65616e34000002
46. m='6c6520737465702e204974206d65616e'
47. print(base64.b16decode(m.upper()))
```

以下代码可用于从点的点压缩技术表示还原为点的仿射坐标表示，仅在$T1$中进行描述：

```
1. #结果为 10 进制，根据压缩坐标确定奇偶性来选择解
2. p=0xb0000000000000006c5b40000000000010ad7f77
3. R=GF(p)
4. x=R(0x725c5b2943cc60511e0ff0dc2caa3d5b718c5453)
5. a=R(0x7cbc14e5e0a72f05864385397829cbc15a2fe1d6)
6. b=R(0xa9cbc14e5e0a72f0bc20f05397829cbc24fd94a8)
7. ans1=(sqrt(x^3+a*x+b))
8. ans2=p-ans1
9. print(ans1,ans2)
```

解题结果：私钥$n_A = 0x46a79bf05f70d85552d0c2e587354e6bd8ad972f$，明文$m$为$'le\ step.It\ mean'$。

20.2.2 $Pohlig-Hellman$攻击破解$T2$

题目要求：赛题第二题给出如下已知信息：

$p = 0x8000000000000000000000000000000000000012b$；

$A = -0x3$；

$B = 0x74f$；

$P = [0x25e3ea3957e945a871b9ceb6ff1659e15e325167,1]$；

$Q_A = [0x43835d772f7dd4f90399fb35645538bb487f22cd,0]$；

$C_1 = [0x3a7b6c5df7a1d54b871e410d8d7b4d37c60f98ad,0]$；

$C_2 = [0x500504db962dcf4bb68a458414ee8a44db0b2c1b,0]$。

题目分析：椭圆曲线有理点即基点P的阶可以较为容易地进行因数分解，且不包含较大的素数，满足$Pohlig-Hellman$攻击的条件，可以较为容易地使用$Pohlig-Hellman$攻击进行求解。

解题代码：

```
1. # 第二题
2. import base64
3. def pohlig_hellman(P,Q,p):
4.     E = P.curve()
```

```
    n_P = P.order()
    tmp = factor(n_P)
    print(tmp)
    primes = []
    i = 0
    while(i<len(tmp)):
        primes += [tmp[i][0]^tmp[i][1]]
        i += 1
    print(primes)
    dlogs = []
    for fac in primes:
        t = int(n_P) // int(fac)
        dlog = discrete_log(t*Q,t*P,operation="+")
        dlogs += [dlog]
        print("factor: ",str(fac), "Discrete Log: ", str(dlog))
    res = crt(dlogs,primes)
    return res

p = 0x800000000000000000000000000000000000012b
a = -0x3
b = 0x74f
E = EllipticCurve(GF(p), [a, b])

P=E(0x25e3ea3957e945a871b9ceb6ff1659e15e325167,3167397273789126583482132939565653471687127317)

Q=E(0x43835d772f7dd4f90399fb35645538bb487f22cd,6074595967690297922525715970779283529044494455990)
k=pohlig_hellman(P,Q,p)
print('%#x'%k)
print(P*k == Q)


C1=E(0x3a7b6c5df7a1d54b871e410d8d7b4d37c60f98ad,4959439110383160675511223703755441361368569681236)

C2=E(0x500504db962dcf4bb68a458414ee8a44db0b2c1b,7005225034828452195676622155927332515720325556314)
M=C2-C1*k
print('%#x'%M[0])
#M[0]=0x726561636820796f757220676f616c2e38000003
m='726561636820796f757220676f616c2e'
print(base64.b16decode(m.upper()))
```

解题结果：私钥$n_A = 0x890f30353cda7d2a0b3129b8049fe578924a585$，明文$m$为$'reach\ your\ goal.'$。

20.2.3 $Pollard-rho$攻击破解$T4$

题目要求：赛题第四题给出如下已知信息：

$p = 0x8049a325d5a0ed72448756f61ddf54149b7ed883$；

$A = 0x0$；

$B = 0x8$；

$P = [0x1e3b0742ebf7d73ff1a781164c46739a153663f3,1]$；

$Q_A = [0x475f5475f46d4a694186e77390aa785fbc947dd3,0]$；

$C_1 = [0x59cc1083b33c0860d2e784dece17ee0ac0d4ae8f,1]$；

$C_2 = [0x211764e736b1dbea35b0aade9f42a78007fe05bf,1]$。

题目分析：在第四题中，可计算得基点P的阶，与第二题类似，同样可以较为容易地因式分解为 2 * 13 * 3282463 * 40301143 * 106469111404179991516915685034613，因此可以考虑通

过$Pohlig-Hellman$攻击进行求解，但与第二题不同的是，其出现了一个较大的素因子，在这一素因子下，难以通过一般方法求解离散对数，可考虑使用$Pollard-rho$攻击。

解题代码：

```
#第四题 rho
import random, base64
sz = 2
csz = 3
check_mod = 100003

p = 0x8049a325d5a0ed72448756f61ddf54149b7ed883
a = 0x0
b = 0x8
r = 732392985241959794169950395659112048534622554084
E = EllipticCurve(GF(p), [a, b])
P =
E(128020842781858036439905859096809334532090926327,216638414648378924980783135174517618296047306608)
R =
E(227159112311867888196468423788662266128033801383,469044615126628421543254796879693227012957965202)

A1 = B1 = A2 = B2 = x = y = 0
Fm = E(0,1,0)
stp = [E(0,1,0) for i in range(sz)]
da = [0 for i in range(sz)]
db = [0 for i in range(sz)]

seeda = 1655165156177
seedb = 384864864613

list_A1 = []
list_B1 = []
list_Fm = []

def Block(num):
    global csz
    return int(num[0]) % csz

def F():
    global Fm, A1, B1, r
    i = Block(Fm)
    if i < sz :
        A1 += da[i]
        B1 += db[i]
        Fm += stp[i]
    else:
        A1 <<= 1
        B1 <<= 1
        Fm += Fm
    if A1 >= r :
        A1 -= r
    if B1 >= r :
        B1 -= r

def check():
    global Fm, check_mod
    return int(Fm[0]) % check_mod == 0

def insert():
    global A1, B1, Fm, list_A1, list_B1, list_Fm
    list_A1 += [A1]
```

```
    list_B1 += [B1]
    list_Fm += [Fm]

def count_and_query():
    global A1, B1, A2, B2, Fm, list_A1, list_B1, list_Fm
    length =len(list_Fm)
    i = 0
    while(i < length):
        if Fm == list_Fm[i] and B1 != list_B1[i]:
            A2 = list_A1[i]
            B2 = list_B1[i]
            return 1
        i += 1
    A2 = A1
    B2 = B1
    return 0

def pollard_rho():
    global Fm, A1, B1, A2, B2
    cnt = 0
    msz = 0
    while(1):
        if (check()):
            if (count_and_query()):
                if (B1 != B2):
                   break
                A1 = randint(1,r-1)
                B1 = randint(1,r-1)
                Fm = A1 * P + B1 * R
            else:
                insert()
        F()
    return (A2 - A1 + r) * inverse_mod((B1 - B2 + r), r) % r

i = 0
while(i < sz):
    da[i] = seeda if i == 0 else (((da[i - 1])^2)%r)
    db[i] = seedb if i == 0 else (((db[i - 1])^2)%r)
    stp[i] = da[i] * P + db[i] * R
    print(stp[i])
    i += 1
A1 = randint(1,r-1)
B1 = randint(1,r-1)
Fm = A1 * P + B1 * R
print('Your initial point is ',Fm)
result = pollard_rho()
if (result > -1):
    print('result : ',result)
k = result
print('%#x'%k)
print(P*k == Q)


C1=E(0x59cc1083b33c0860d2e784dece17ee0ac0d4ae8f,187649590762676843210091051264549844222092800055)

C2=E(0x211764e736b1dbea35b0aade9f42a78007fe05bf,178342522639367375368502039660772148399065726933)
M=C2-C1*k
print('%#x'%M[0])
#M[0]=
m=''
```

```
print(base64.b16decode(m.upper()))
```

20.2.4 袋鼠算法破解$T6$

题目要求：赛题第六题给出如下已知信息：

$p = 0xb77902abd8db9627f5d7ceca5c17ef6c5e3b0969$；

$A = 0x9021748e5db7962e1b208e3949d42ad0388a18c$；

$B = 0x744f47974caabdd8b8192e99da51c87f91cc453e$；

$P = [0x609e413d6e302e1c79664f785bf869d467dd6858,1]$；

$Q_A = [0x352ad1b63b37373cb04bf5c7309c1b6c401f7bdb, 1]$；

$C_1 = [0xa7007abfbcfe16406c4ba61bcad325e5f25bd01b, 1]$；

$C_2 = [0x805ba9899330d7e9f79f86857927d9ee551f47f5,1]$。

注：在本题中，$Alice$因操作不当使得n_A在生成过程中泄露了信息。已知n_A的取值范围为$[2^{100}, 2^{100} + 2^{80}]$。

题目分析：题目给出了n_A的范围，可以考虑使用袋鼠算法进行攻击，该算法可以利用到$Pollard - rho$方法无法利用的范围信息。

解题代码：

```
# 第六题
import base64
hashValue = 0
def Hash(P):
    if P == 0:
        return 1
    return int(P.xy()[0]) % hashValue +int(P.xy()[1]) % hashValue+ 1
def pollardKangaroo(P, Q, a, b):
    global hashValue
    hashValue = math.ceil(sqrt((b-a))/2)
    # Tame 的袋鼠迭代:
    xTame, yTame = 0, b * P
    for i in range(0,math.ceil(0.7*sqrt(b-a))):
        xTame += Hash(yTame)
        yTame += Hash(yTame) * P
    # yTame == (b + xTame) * P 等式必须成立
    # Wild 的袋鼠迭代:
    xWild, yWild = 0, Q
    for i in range(0, math.ceil(2.7*sqrt(b-a) ) ):
        xWild += Hash(yWild)
        yWild += Hash(yWild) * P
        if yWild == yTame:
            return b + xTame - xWild
    # 二者未相遇，结果未找到:
    return 0

p = 0xb77902abd8db9627f5d7ceca5c17ef6c5e3b0969
a = 0x9021748e5db7962e1b208e3949d42ad0388a18c
b = 0x744f47974caabdd8b8192e99da51c87f91cc453e
E = EllipticCurve(GF(p), [a, b])

P=E(0x609e413d6e302e1c79664f785bf869d467dd6858,28628277113903018053174842372461579987252160
2949)

Q=E(0x352ad1b63b37373cb04bf5c7309c1b6c401f7bdb,63810226592385431083638590339373783530273661
8629)
k = pollardKangaroo(P,Q,2^100-1,2^100+2^80+1)
print('%#x'%k)
print(P*k == Q)
```

```
36.
37.
C1=E(0xa7007abfbcfe16406c4ba61bcad325e5f25bd01b,6186521796570204170115529607505073670596809
34053)
38.
C2=E(0x805ba9899330d7e9f79f86857927d9ee551f47f5,4151631651564675939461664784478577191004979
21159)
39. M=C2-C1*k
40. print('%#x'%M[0])
41. #M[0]=
42. m=''
43. print(base64.b16decode(m.upper()))
```

20.2.5 MOV破解$T3$

题目要求：赛题第三题给出如下已知信息：

$p = 0x10000000000000000000000000000000000018f3$；

$A = 0x1$；

$B = 0x0$；

$P = [0x77d0847d0a4b9448433de6eef45cbdf32dc82fdf, 0]$；

$Q_A = [0xb69c1c1d4180cbe558799cc71bc5cd72df01d877,1]$；

$C_1 = [0x6feb50134b7538cf7287dab9b68c8af9c8bb8dac, 0]$；

$C_2 = [0x56a716fb6f13480f509915339fd9c4e5f496124,1]$。

题目分析：MOV攻击是一种针对超奇异椭圆曲线的攻击方式，而第三题的椭圆曲线属于超奇异椭圆曲线，可以使用该方法尝试进行攻击。

在第三题中，椭圆曲线的阶恰好等于$p+1$，且可以整除p^2-1，属于k为 2 的超奇异椭圆曲线，可以构造F_{p^2}域上的离散对数问题，即可以使用MOV攻击求解。

解题代码：

```
1. # 第三题
2. import base64
3. @parallel(ncpus = 40)
4. def getLog(beta,alpha):
5.     return discrete_log_rho(beta,alpha,alpha.multiplicative_order(),operation="*")
6. def MOV(P,Q,p,k):
7.     # 生成原 EC
8.     F1 = GF(p)
9.     r = F1.order() # 阶
10.    E1 = EllipticCurve(F1, [0, 0, 0, a, b])
11.    # EC 的阶
12.    n = E1.order()
13.    print("E1 : ",E1)
14.    # 将 EC(Fp) 映射到 EC(Fp^k)
15.    F2 = GF(p^k)
16.    phi = Hom(F1, F2)(F1.gen().minpoly().roots(F2)[0][0])
17.    E2 = EllipticCurve(F2 ,[0, 0, 0, a, b])
18.    print("n = ", n)
19.    # 两个点映射到 EC(Fp^k)上去
20.    P1 = E1(P)
21.    Q1 = E1(Q)
22.    P2 = E2(phi(P1.xy()[0]), phi(P1.xy()[1]))
23.    Q2 = E2(phi(Q1.xy()[0]), phi(Q1.xy()[1]))
24.    # cn
25.    cn1 = E2.gens()[0].order()
26.    coeff = ZZ(cn1 // n)
27.    print("cn1 = ",cn1)
28.    print("coeff = ", coeff)
```

```
    # 倍点
    while (1):
        Q = coeff * E2.random_point()
        print(Q.order() == n)
        if (Q.order() == n):
            print("Q = ",Q)
            break
    alpha = P2.weil_pairing(Q, n)
    beta = Q2.weil_pairing(Q, n)
    print("alpha = ", alpha)
    print("beta = ", beta)
    print(alpha.multiplicative_order())
    print(beta.multiplicative_order())
    d = getLog(beta,alpha)
    return d

p = 0x10000000000000000000000000000000000018f3
a = 0x1
b = 0x0
E = EllipticCurve(GF(p), [a, b])
P =
(0x77d0847d0a4b9448433de6eef45cbdf32dc82fdf,563054081111822273597187358803861228363729137
38)
Q =
(0xb69c1c1d4180cbe558799cc71bc5cd72df01d877,600014912768132632409964071651227335951517428
425)
# 嵌入度
k = 2  # 满足 n | (p^k - 1) 的最小 k
na = MOV(P, Q, p, k)
print('%#x'%na)
print(E(P)*na == E(Q))


C1=E(0x6feb50134b7538cf7287dab9b68c8af9c8bb8dac,703496648379954379181247738651965779411518
03834)
C2=E(0x56a716fb6f13480f509915339fd9c4e5f496124,113519567759002067019512140046205841677242
4070257)
M=C2-C1*na
print('%#x'%M[0])
#M[0]=
m=''
print(base64.b16decode(m.upper()))
```

20.3 举一反三

$ECDLP$，即椭圆曲线上的离散对数问题。1987 年，$Koblitz$利用椭圆曲线上点形成的$Abelian$加法群构造了$ECDLP$。它可以提供一个更快、具有更小的密钥长度的公开密钥密码系统，备受人们的广泛关注，为人们提供了诸如实现数据加密、密钥交换、数字签名等密码方案的有力工具。

椭圆曲线上离散对数问题$ECDLP$定义如下：给定素数p和椭圆曲线E，对于$Q = kP$，在已知P, Q的情况下求出小于p的正整数k。可以证明由k和P计算Q比较容易，而由Q和P计算k则比较困难。将椭圆曲线中的加法运算与离散对数中的模乘运算相对应，将椭圆曲线中的乘法运算与离散对数中的模幂运算相对应，就可以建立基于椭圆曲线的对应的密码体制。

针对椭圆曲线上离散对数问题的求解，除去遍历搜索以外，相关研究工作者也构建出了许多有效的求解方法，具体包括$SSSA$算法、$Pohlig-Hellman$算法、$Pollard-rho$算法、袋鼠算法、MOV算法等等。各种算法的适用场景不同，算法复杂程度也不同，需要结合实际椭圆曲线

上离散对数问题进行分析和使用。在理解椭圆曲线的群结构的知识后，理解困难求解算法如MOV算法会更为容易。

在一些密码学相关竞赛中，也常出现以椭圆曲线上离散对数问题求解为核心的试题。一般情况下暴力破解的能力有限，诸多算法下的椭圆曲线离散对数求解方法能够起到较好的计算效果。值得注意的是，即使使用了多种算法进行$ECDLP$问题的求解，其仍旧可能需要大量计算时间才能得到结果。

第21章 分组密码

21.1 基本原理

分组加密即每次加密一组固定大小的明文，现代分组密码体制的基本原理起源于香农提出的乘积密码，通常采用迭代结构，这类密码体制会明确定义一个密钥编排方案和一个轮函数。迭代结构会依据定义好的密钥编排方案基于初始密钥K产生N_r个轮密钥，来对明文分组进行多轮迭代加密，单轮迭代内会依据定义好的轮函数使用轮密钥对上轮结果进行多项加密处理。轮函数的定义会充分运用香农提出的混淆与扩散两大策略：混淆是指将密文与密钥之间的统计关系变得尽可能复杂，使得攻击者即使获取了密文的一些统计特性，也无法推测密钥；而扩散则负责使明文中的一位变化趋向于影响密文中更多位的变化，使密文位与明文位的映射关系趋于模糊。

当下流行的迭代结构基础设计方案包括 Feistel 网络结构，例如上一代对称加密算法 DES，以及 SPN 结构，例如目前最为常使用的 AES 加密算法。

针对分组密码体制，衍生了线性分析、差分分析等知名的迭代密码攻击方法，它们试图利用明文子集与迭代最后一轮输入状态子集间的概率关系，在候选密钥中寻找满足相应关系的最后一轮轮密钥，以获知初始密钥的一部分。

有关 DES、AES 算法定义，以及线性分析、差分分析等传统攻击方法原理的内容，已在密码学等相关课程中给予详细讨论，在本篇章中便不再加以赘述。本章将着眼于当前主流且最为有效的两类分组密码攻击方法：代数攻击与积分攻击，展开讨论。

21.1.1 代数攻击

香农在《$Communication\ Theory\ of\ Secrecy\ Systems$》一书中提出，破译一个密码系统的问题可以转化为求解一个大型未知复杂方程组系统，将密码系统重写成求解多元方程组的形式，求出的解即为密钥。其中心思想应用到现代的密码系统中，就是将密码系统看成一个整体结构，将输入的明文和初始密钥看作是这个结构的输入部分，加密产生的密文看作是输出部分，由此建立相应的多元低次方程组。

而一个密码系统中可能会同时存在线性变换和非线性变换，为使问题进一步简化，建立方程时，可以先将明文、密文、密钥等信息通过已知的线性变换转化为加密过程中起主要作用的非线性变换的输入输出流，再建立起方程组，进而通过某种方法由这一信息流求解出密钥。

上述过程中，密码系统被转化为有限域 GF(2)上的低次多元方程组时，通常情况下这些方程组是超定的非线性方程组，求解这类方程组通常无法在多项式时间内解决。早期的密码系统设计一般都没有把香农的警告明确地考虑进密码系统的设计工作中去，但是，随着密码学界的进一步深入研究，如果将现代普通密码系统转化为方程组，目前求解这些方程组的过程并不会像求解具有随机系数的多元方程组那样复杂。例如，已有学者在 2000 年时提出了一种名为XL的方法，它是线性化方法的一种变型，原理是在原有的方程组中引入新的线性独立的方程来进行消元，进而解决原方程组，这些拓展后方程的系数可以由一个矩阵来表示，矩阵的行表示一个方程，列表示一个单项式，通过不断降低矩阵的行秩最终求解出方程组的解，证明了代数攻击可以对 128 比特密钥的 AES 加密算法和其他结构类似的分组密码系统构成威胁。

上述攻击思想的提出极大地促进了代数攻击的研究，总体而言，当下代数攻击的主要思想是通过求解包含消息、密文和密钥信息的非线性方程组来反向推导密钥。

21.1.2 Grobner基理论

Grobner基理论是一种被普遍认同的用于求解多变元高次方程系统的有效算法。其本质是从多项式环中任意理想的生成元出发，刻画和计算出一组具有优秀性质的生成元，进而研究理想的结构并进行相关的理想运算，进而得到方程系统的解。

利用Grobner基求解多元高次方程组的主要原理描述为，先利用Grobner基理论依次求出由方程组中各方程所生成理想I的消元理想$I^{(1)}, I^{(2)}, \dots, I^{(n-1)}$的Grobner基，其中$I^{(1)} = I \cap K[x_2, x_3, \dots, x_n]$，进而求解单变元理想的零点，再利用扩张定理将前述消元理想的零点扩张成理想I的公共零点，从而将含有n个变元的代数方程组转化为单变元方程组进行求解。

为实现上述过程，需给出如下相关定义及结论。

定义 21-1 设K是域，$K[x_1, x_2, \cdots, x_n]$是定义在域K上文字（变元）为$x_1, x_2, \cdots, x_n$的n元多项式环，$K[x_1, x_2, \cdots, x_n]$中的所有**单项**的集合记为$T = \{{x_1}^{\alpha_1}{x_2}^{\alpha_2}\cdots{x_n}^{\alpha_n} | \alpha_i \in \mathbb{Z}, \alpha_i \geq 0\}$，如果令$x = (x_1, x_2, \cdots, x_n)$，$\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_n)$，那么${x_1}^{\alpha_1}{x_2}^{\alpha_2}\cdots{x_n}^{\alpha_n}$也可以记作$x^\alpha$，它的**全幂次**为$|\alpha| = \alpha_1 + \alpha_2 + \cdots + \alpha_n$，$\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_n)$为**幂次向量**。

对于一元多项式，所有的项一般按照次数的高低即可以简单排序，对于n元多项式，需要在项集合上定义“项序”。下面给出常见的三种“项序”的定义。

定义 21-2 纯字典序。设$\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_n)$，$\beta = (\beta_1, \beta_2, \cdots, \beta_n)$为幂次向量，若$\alpha - \beta$的第一个非零项大于 0，则称关于变元序$x_1 > x_2 > \cdots > x_n$，$\alpha >_{lex} \beta$或者$x^\alpha >_{lex} x^\beta$。

定义 21-3 次数字典序。设$\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_n)$，$\beta = (\beta_1, \beta_2, \cdots, \beta_n)$为幂次向量，如果$|\alpha| > |\beta|$或者$|\alpha| = |\beta|$且$\alpha >_{lex} \beta$，则称$\alpha >_{grlex} \beta$或者$x^\alpha >_{grlex} x^\beta$。

定义 21-4 次数反字典序。设$\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_n)$,$\beta = (\beta_1, \beta_2, \cdots, \beta_n)$为幂次向量，如果$|\alpha| > |\beta|$或者$|\alpha| = |\beta|$且$\alpha - \beta$最后一个非零项小于 0，则称$\alpha >_{grevlex} \beta$或者$x^\alpha >_{grevlex} x^\beta$。

例如，如果规定变元序$x > y > z$，那么，

$$xy^2z^3 >_{lex} y^2z^3,\ xy^3z^3 >_{lex} xy^2z^6$$

$$xy^2z^3 >_{grlex} y^2z^3,\ xy^2z^6 >_{grlex} xy^3z^3,\ xy^3z^3 >_{grlex} xy^2z^4$$

$$xy^2z^3 >_{grevlex} y^2z^3,\ xy^2z^6 >_{grevlex} xy^3z^3,\ y^4z^3 >_{grevlex} xy^2z^4$$

定义了项序后，对于一个n元多项式$f \in K[x_1, x_2, \cdots, x_n]$，$f \neq 0$，可以将$f$的所有非零项按照项序降序进行排列，假设排列后的第一项为$c{x_1}^{\alpha_1}{x_2}^{\alpha_2}\cdots{x_n}^{\alpha_n}$，一般称 $\mathrm{lt}(f) = c{x_1}^{\alpha_1}{x_2}^{\alpha_2}\cdots{x_n}^{\alpha_n}$为$f$的**首项**，$\mathrm{lc}(f) = c$为$f$的**首项系数**，$\mathrm{lm}(f) = {x_1}^{\alpha_1}{x_2}^{\alpha_2}\cdots{x_n}^{\alpha_n}$为$f$的**首项单项式**。显然，

$$\mathrm{lt}(f) = \mathrm{lc}(f)\ \mathrm{lm}(f)$$

一元多项式之间可以进行带余除法，$K[x_1, x_2, \cdots, x_n]$上的n元多项式类似地有约化的概念。

定义 21-5 设多项式$f, g, h \in K[x_1, x_2, \cdots, x_n]$,$g \neq 0$,称$f$模$g$可**约化**为$h$，记作$f \xrightarrow{g} h$，是指 lt($g$)是$f$某一非零项 X 的因子，且$h = f - \frac{X}{\mathrm{lt}(g)}g$。如果$f$模$g$不能约化，那么称$f$模$g$是**既约**的。将$f$多次模$g$约化，最终将得到模$g$既约的多项式，称为$f$模$g$的**余式**。

为了指明非零单项式 X，$f \xrightarrow{g} h$有时也记作$f \xrightarrow[\mathrm{X}]{g} h$。

例 21-1 设多项式$f = x^3 + xy^3z^3 + xy^2z^6$，$g = x^2 + x \in \mathbb{Z}_2[x, y, z]$，规定字典序$x > y > z$，项序为纯字典序，试求$f$模$g$的余式。

解： lm(g)= $x^2|x^3$，所以$h = f - \frac{x^3}{x^2}g = x^2 + xy^3z^3 + xy^2z^6$，即$f \xrightarrow[x^3]{g} x^2 + xy^3z^3 + xy^2z^6$。

$x^2+xy^3z^3+xy^2z^6$模g不是既约的，继续约化有$h \xrightarrow[x^2]{g} xy^3z^3+xy^2z^6+x$。

当模多项式是多个多项式时，有和定义 21-5 类似的定义。

定义 21-6 设多项式$f \in K[x_1, x_2, \cdots, x_n]$，$G=\{g_i | g_i \in K[x_1, x_2, \cdots, x_n], g_i \neq 0, 1 \leq i \leq r\}$，可以用$G$中的任意多项式多次对$f$进行约化，若结果为$h$，记作$f \xrightarrow{G} h$。如果$f$模所有$g_i(1 \leq i \leq r)$均是既约的，则称$f$模$G$是**既约**的。将$f$多次模$G$约化，最终将得到模$G$既约的多项式，称为$f$模$G$的**余式或者范式**，记作 nform($f$，$G$)。

例 21-2 设多项式 $f=x^3+xy^3z^3+xy^2z^6$，$g_1=x^2+x$，$g_2=y^2+y$，$g_3=z^2+z$，$\in \mathbb{Z}_2[x, y, z]$，规定变元序$x>y>z$，项序为纯字典序，试求$f$模$G_1=\{g_1, g_2, g_3\}$的余式。

解： $f \xrightarrow{g_1} xy^3z^3+xy^2z^6+x \xrightarrow{g_2} xyz^3+xyz^6+x \xrightarrow{g_3} xyz+xyz+x=x$。

例 21-3 设多项式 $f=x^3+xy^3z^3+xy^2z^6$，$g_1=x^2+x$，$g_2=xy+y$，$g_3=yz+z$，$\in \mathbb{Z}_2[x, y, z]$，规定变元序$x>y>z$，项序为纯字典序，试求$f$模$G_2=\{g_1, g_2, g_3\}$的余式。

解： $f \xrightarrow{g_1} x+xy^3z^3+xy^2z^6 \xrightarrow{g_2} x+y^3z^3+xy^2z^6 \xrightarrow{g_2} x+y^3z^3+y^2z^6 \xrightarrow{g_3} x+z^3+z^6$，

如果改变约化的顺序，还可以得到：

$$f \xrightarrow{g_1} x+xy^3z^3+xy^2z^6 \xrightarrow{g_3} x+xz^3+xy^2z^6 \xrightarrow{g_3} x+xz^3+xz^6$$

上例说明，对于同一个多项式集合G，不同的约化顺序可能得到不同的余式。

定义 21-7 给定多项式有限集$G \subset K[x_1, x_2, \ldots, x_n]$及项序，称$G$是给**定项序下的Grobner基**，当且仅当对于任意多项式$g \in K[x_1, x_2, \ldots, x_n]$，nform($f$，$G$)都是唯一的。称$G$为多项式有限集$P \subset K[x_1, x_2, \ldots, x_n]$的**Grobner基**，是指$G$为Grobner基且理想$\langle G \rangle = \langle P \rangle$。

根据定义，上例中G_2不是Grobner基。

接下来，讨论Grobner基的计算及性质。

定义 21-8 设$f, g \in K[x_1, x_2, \ldots, x_n]$为域$K$上的$n$元多项式，$g \neq 0, L=\mathrm{lcm}(\mathrm{lm}(f), \mathrm{lm}(g))$，则称

$$\mathrm{spol}(f, g)=\mathrm{lc}(g)\frac{L}{\mathrm{lm}(f)}f-\mathrm{lc}(f)\frac{L}{\mathrm{lm}(g)}g$$

为f和g的**s**-多项式。

例 21-4 $g_1=x^2+x$，$g_2=xy+y$，$g_3=yz+z \in \mathbb{Z}_2[x, y, z]$，规定变元序$x>y>z$，项序为纯字典序，试计算spol($g_1, g_2$)，spol($g_1, g_3$)和spol($g_2, g_3$)。

解： spol$(g_1, g_2)=\frac{x^2y}{x^2}(x^2+x)-\frac{x^2y}{xy}(xy+y)=x^2y+xy+x^2y+xy=0$，spol$(g_1, g_3)=\frac{x^2yz}{x^2}(x^2+x)-\frac{x^2yz}{yz}(yz+z)=x^2yz+xyz+x^2yz+x^2z=x^2z+xyz$

spol$(g_2, g_3)=\frac{xyz}{xy}(xy+y)-\frac{xyz}{yz}(yz+z)=xyz+yz+xyz+xz=xz+yz$

不加证明给出如下基本定理。

定理 21-1 非空多项式有限集$G \subset K[x_1, x_2, \ldots, x_n]$是Grobner基，当且仅当对于任意$f$，$g \in G$，nform(spol$(f, g)$，$G$)=0。

在例 21-4 中，spol$(g_2, g_3)=xz+yz$，$xz+yz \xrightarrow[yz]{g_3} xz+z$，而$xz+z$模$G_2=\{g_1, g_2, g_3\}$是既约的，因此 nform(spol$(g_2, g_3)$，$G$)$=xz+z \neq 0$。所以，$G_2$不是Grobner基。

一般地，根据定理 21-1，可以在多项式有限集G中选择多项式对，计算它们的s-多项式，以此判断G是否为Grobner基。同时，根据定理 21-1，还可以得到如下Grobner基的计算方法。

Buchberger Grobner基算法

输入：非空的多项式有限集$P \subset K[x_1, x_2, \dots, x_n]$及项序

输出：P的Grobner基G

步骤 1 令$G = P$，$S = \{\{f，g\}|f \neq g，f，g \in P\}$

步骤 2 重复下列步骤，直至S为空：

2.1 选取$\{f，g\} \in S$，令$S = S\backslash\{\{f，g\}\}$

2.2 计算 r =nform(spol$(f，g)$，G)

2.3 若 $r \neq 0$，则令$S = S \cup \{\{r，w\}|w \in G\}$，$G = G \cup \{r\}$

Buchberger Grobner基算法的输出较大，有较多的冗余多项式，得到的结果也不唯一，为了保证唯一性，引进约化Grobner基的概念。

定义 21-9 Grobner基$G \subset K[x_1, x_2, \dots, x_n]$是**约化Grobner基**，是指如果$G$中每个多项式$g$都有$\mathrm{lc}(g) = 1$，且对$G\backslash\{g\}$是既约的。

在 Buchberger Grobner基算法的基础上，可以得到如下约化Grobner基算法。

Buchberger 约化Grobner基算法

输入：非空的多项式有限集$G \subset K[x_1, x_2, \dots, x_n]$及项序，$G$为Grobner基

输出：G的约化$Grobner$基G^*

步骤 1 令$P = G$，G^*为空

步骤 2 重复下列步骤，直至$\boldsymbol{P}$为空：

2.1 选取$f \in P$，令$P = P\backslash\{f\}$

2.2 若对所有$g \in P \cup G^*$都有$\mathrm{lt}(g) \nmid \mathrm{lt}(f)$，则令$G^* = G^* \cup \{f\}$

步骤 3 重复下列步骤，直至G^*为约化：

3.1 选取$g \in G^*$，使得g模$G^*\backslash\{g\}$可约化，令$G^* = G^*\backslash\{g\}$

3.2 计算 r =nform(g，G^*)，若 $r \neq 0$，则令$G^* = G^* \cup \{r\}$

步骤 4 令$G^* = \left\{\frac{g}{\mathrm{lc}(g)}|g \in G^*\right\}$

Faugere 分别于 1999 年和 2002 年提出了基于线性代数的 F4 算法和基于标签的 F5 算法，F4 算法和 F5 算法是目前公认的计算Grobner基的较高效的算法。

例 21-5 $g_1 = x^2 + x$，$g_2 = xy + y$，$g_3 = yz + z \in \mathbb{Z}_2[x，y，z]$，规定变元序$x > y > z$，项序为纯字典序，利用 SageMath 计算理想$\langle g_1，g_2，g_3\rangle$的Grobner基。

解：

```
1. K.<x,y,z>=GF(2)[]
2. I=ideal(x^2+x,x*y+y,y*z+z)
3. I.groebner_basis()
```

输出结果为：$[x^2 + x, x * y + y, x * z + z, y * z + z]$

Grobner基求解多变量方程组的一个重要理论基础是Grobner基的消元性质。

定理 21-2 设$G \subset K[x_1, x_2, \cdots, x_n]$是基于变元序$x_1 > x_2 > \cdots > x_n$的纯字典序 Grobner 基，那么，对于任意的$1 \leq i \leq n$，都有$\langle G \rangle \cap K[x_1, x_2, \cdots, x_i] = \langle G \cap K[x_1, x_2, \cdots, x_i] \rangle$，其中$\langle G \cap K[x_1, x_2, \cdots, x_i] \rangle$是指在$K[x_1, x_2, \cdots, x_i]$中的理想。

定理 21-2 说明，如果$\langle G \rangle \cap K[x_1, x_2, \cdots, x_i]$为非零理想，那么Grobner基$G$中一定包含变元仅包含$x_1, x_2, \cdots, x_i$的多项式。通俗地理解，可以认为Grobner基中的多项式都是经过消元后的“较短”的多项式。

例 21-6 求解如下多变量方程组，假设$x1, x2, \cdots, x6$均是$\mathbb{Z}_2$上的变量。

$$\begin{cases} f1 = x1^3 * x2 + x3 + x4 * x5 = 0 \\ f2 = x1 * x2 * x3 * x6 + x4 + x5 = 0 \\ f3 = x2^2 * x3 * x4 + x5 + x6 = 0 \\ f4 = x3^5 + x2 * x4 = 0 \\ f5 = x3 * x4 + x5 = 0 \\ f6 = x5 + x6 = 0 \end{cases}$$

解：设理想$I=\langle f1, f2, \cdots, f6 \rangle$，任意多项式$f \in I$，如果存在$x1, x2, \cdots, x6$满足原方程组，它们也一定可满足$f = 0$。

通过 SageMath 求得$\{f1, f2, \cdots, f6\}$的 Grobner 基为$\{x3^5, x1^3 * x2 + x3, x4, x5, x6\}$，由此立刻得到$x4 = x5 = x6 = 0$。

因为在$\mathbb{Z}_2$上，对于$1 \leq i \leq 6$，均有$xi^2 - xi = 0$，所以$x3^5 = x3 = 0$，$x1^3 * x2 + x3 = x1 * x2 = 0$，得到$x1 = 0$或者$x2 = 0$。方程组共有 3 组解。

针对分组密码系统，以 AES 加密算法即$Rijndael$算法为例，基于$Grobner$基的代数攻击方法的主要思路为：

（1）将$Rijndael$算法规约为对一个超定多变元二次方程组的求解，方程组的变元为输入、输出、密钥或者一些临时变量；

（2）通过$Grobner$基求解该方程组；

（3）将求出的结果反馈到 AES 加密算法上验证，最终达到对 AES 密码系统进行攻击、破译的目的。

其详细过程可参考相关文献及本章后续示例。

21.1.3 SAT 问题

代数攻击中，除了可以使用Grobner基理论来进行高次方程系统的有效求解外，还有一类流行且更为有效的求解算法，即转换为 SAT 问题进行求解。

布尔可满足性问题（也称为命题可满足性问题）即 SAT 问题，是一类用于确定是否存在满足给定布尔公式的解释的问题。换句话说，它询问给定布尔公式内的所有变量是否可以一致地用值$TRUE$或$FALSE$进行替换，使得公式计算结果为$TRUE$，如果满足这一要求，则称该公式为可满足的，相反的，对于所有可能的赋值情况，公式的计算结果始终为$FALSE$，则称该公式为不可满足的。

例如，公式“a AND NOT b”便可以满足的，因为在$a = TRUE$、 $b = FALSE$时，可以使得a AND NOT $b = TRUE$。相应地，“a AND NOT a”则不可满足的，因为无论a取值为何，该表达式的值始终为$FALSE$。

有关于 SAT 问题，首先给出如下定义。

定义 21-10 对于任意布尔公式，其内所有命题变量$x_1, x_2, \dots, x_n$的集合记作$X = \{x_1, x_2, \dots, x_n\}$，称为公式的**变量集**，所含变量的个数$n = |X|$称为**问题规模**。

定义 21-11 对于任意变量x_i，形如x_i或$\overline{x_1}$的表达式称为**文字**，文字是一个逻辑公式的最小组成单元。其中，称x_i为**正文字**，称$\overline{x_1}$为**负文字**，一般用l_i表示一个文字。

定义 21-12 X上有限个文字的析取$l_1 \vee l_2 \vee \dots \vee l_n$称为**子句**，记作$C$。子句中包含的文字数量称为子句长度，长度为$k$的子句记作一个$k$-子句，特别地，当$k = 1$时，称为**单元子句**。

定义 21-13 X上有限个子句的合取$C_1 \wedge C_2 \wedge \dots \wedge C_m$称为**合取范式($CNF$)**，记作$F(X)$。

定义 21-14 **真值指派**σ是指一个定义在X上的n元布尔函数，$\sigma(X): X \longrightarrow \{0,1\}$。对于正文字$l_i = x_i$，若$\sigma(x_i) = 1$，则称$l_i$在真值指派$\sigma$下为真；对负文字$l_i = \overline{x_1}$，若$\sigma(x_i) = 0$，也称$l_i$在真值指派$\sigma$下为真；对于子句$C = l_1 \vee l_2 \vee \dots \vee l_n$，$C$在真值指派$\sigma$下为真当且仅当存在至少一个$l_i$，有$l_i$在真值指派$\sigma$下为真；对合取范式$F(X) = C_1 \wedge C_2 \wedge \dots \wedge C_m$，$F(X)$在真值指派$\sigma$下为真当且仅当任意子句$C$在真值指派$\sigma$下均为真。

定义 21-15 定义在X上的布尔公式ϕ，称ϕ是可满足的当且仅当存在真值指派σ使ϕ在σ下为真，记作$\sigma \vDash \phi$。对于任意合取范式$F(X)$，若存在真值指派σ，使得$\sigma \vDash F(X)$，则称$F(X)$是可满足的。

基于上述定义，可以便利地引出有关 SAT 问题的数学描述。给定变量集X和X上的一个合取范式$F(X)$，SAT 问题即判断$F(X)$的可满足性。设k为一个 SAT 问题中所有子句的长度上限，称以k为子句长度上限所构成的 SAT 问题为k-SAT 问题。

对一个 SAT 问题，$F(X)$的所有不同真值指派称为该问题的解空间，其中真值指派的个数称为解空间的大小。若问题规模为n，容易得知，此时的解空间大小为2^n。而若存在真值指派σ_0，使得$\sigma_0 \vDash F(X)$，则称σ_0为该 SAT 问题的一个解。

SAT 求解算法指可以在有限时间内判断一个 SAT 问题可满足性的算法，对一个可满足的 SAT 问题，SAT 求解算法往往会给出一个具体解。典型的 SAT 求解算法可分为完备性算法，非完备性算法和组合算法等。

经典的 SAT 求解算法有DPLL算法、CDCL冲突驱动子句学习算法、LA算法等。而为了寻求一个特定解时，则还会涉及到解空间的搜索，除常规的回溯、剪枝等策略外，SAT 求解算法中通常还会采用许多其它的关键技术，如预处理、变量决策、BCP、冲突分析、重启、非时序回溯等。

对于前述一系列 SAT 求解算法，其相关原理较为复杂且已有较为成熟的执行方案，因此将不会作为本章的重点进行具体介绍。接下来，将讨论如何将由分组密码体制得到的多变量方程组的求解问题转换为 SAT 问题。

将$\mathbb{Z}_2$中的 0 和 1 与布尔值 FALSE 和 TRUE 等同起来。那么每一个$\mathbb{Z}_2$上的n元多项式的求根问题都可以转化为一个 SAT 问题。假设变量a、b是$\mathbb{Z}_2$上的变量，同时也将它们当作布尔变量，那么：

$$ab = a \wedge b$$

$$a + b = \overline{(a \vee \bar{b}) \wedge (\bar{a} \vee b)} = (a \wedge \bar{b}) \vee (\bar{a} \wedge b)$$

$\mathbb{Z}_2$上变量的乘法即为布尔变量的 AND，$\mathbb{Z}_2$上变量的加法即为布尔变量的 XOR。

因为 SAT 问题可满足时要求表达式计算结果为 TRUE，这样一来，线性方程$a + b + c + d = 0$恰好可被转换为如下的 SAT 问题：

$$(\bar{a} \vee b \vee c \vee d) \wedge (a \vee \bar{b} \vee c \vee d) \wedge (a \vee b \vee \bar{c} \vee d) \wedge (a \vee b \vee c \vee \bar{d}) \wedge$$
$$(\bar{a} \vee \bar{b} \vee \bar{c} \vee d) \wedge (\bar{a} \vee \bar{b} \vee c \vee \bar{d}) \wedge (\bar{a} \vee b \vee \bar{c} \vee \bar{d}) \wedge (a \vee \bar{b} \vee \bar{c} \vee \bar{d})$$

简单来说，即为在公式的4个变元中取1或3个否定变元的所有组合，共有 8 个子句。

一般地，$\mathbb{Z}_2$上 l 个变元相加，共需要$\binom{l}{1}+\binom{l}{3}+\binom{l}{5}+\cdots+\binom{l}{j}=2^{l-1}$个子句来进行变换，其中$j$=2⌊$l/2$⌋，即子句个数关于$l$是指数级的，这会导致求解难度骤升。可以把问题规模较大的公式简化为多个问题规模较小的公式，来降低求解难度。例如，方程$x_1+x_2+\cdots+x_l=0$等价于

$$\begin{cases} x_1+x_2+x_3+y_1=0 \\ y_1+x_6+x_7+y_2=0 \\ \cdots \\ y_i+x_{2i+4}+x_{2i+5}+y_{i+1}=0 \\ \cdots \\ y_h+x_{l-2}+x_{l-1}+x_l=0 \end{cases}$$

其中，$h=\lceil l/2\rceil-2$。此时共生成了$h+1$个子方程，每一个子方程等价化为8个长为4的子句。

以m个方程组成的n元二次方程组(通常称为 MQ 问题)为例。

$$\begin{cases} f_1(x_1,\ldots,x_n)=0 \\ \cdots \\ f_m(x_1,\ldots,x_n)=0 \end{cases}$$

将其中可能出现的单项式（MQ问题只考虑小于等于 2 次的单项式，包括常数项）的个数记为M，则$M=\binom{n}{2}+\binom{n}{1}+1$。每个单项式当作一个新的变量，形成新的$M$元线性方程组。定义一个稀疏常数$0<\beta\leq 1$，$M\beta$表示线性方程组中每个方程的期望长度。根据上面的讨论，方程的总数约为$m(\frac{M\beta}{2}-2+1)$= $m(\frac{M\beta}{2}-1)$，变元的总数约为M+ $m(\frac{M\beta}{2}-2)$，可转化为含有$8m(\frac{M\beta}{2}-1)$ 个子句的 4-SAT 问题。

一般来说，将多变量非线性方程组求解问题转化为 SAT 问题主要有三个步骤：

(1) 预处理。基于高斯消元的思想，尝试减小n的值，进而降低 SAT 问题的规模。

(2) 将多元方程组转化为线性方程组，使得每一个单项式变为线性方程组中的一个变元。

(3) 将线性方程组转化为相应的子句集合。

21.1.4 积分攻击

积分攻击是继差分分析和线性分析后，密码学界公认的最有效的密码分析方法之一。积分攻击考虑将一系列状态求和，由于在有限域$\mathbb{Z}_2$上，差分的定义就是两个元素的求和，而高阶差分是在一个线性子空间上求和，因此积分攻击可以看作差分攻击的一种推广，而高阶差分分析又可以看作积分攻击的一个特例。

积分攻击的最主要环节是寻找积分区分器。在寻找一个分组加密算法的积分区分器时，通常只需知道该算法的变换是满射即可，这导致传统积分攻击的方法对基于比特运算设计的加密算法是无效的，鉴于此，Z'aba 等学者在 FSE 2008 上首次提出了基于比特的积分攻击方法，其实质是一种计数的方法，通过分析特定比特位上元素出现次数的奇偶性来确定该比特位上所有值的异或值，并由此判断该位置上比特的平衡性。

相比于差分分析方法，在实施攻击阶段，利用差分分析方法恢复密钥时，通常需要选择密文，随后利用统计方法对每个可能的密钥进行计数，当某个密钥对应的计数器计数明显高于其它密钥的计数器时，就认为该密钥为正确密钥；而在实施积分攻击时，一般没有选择密文的步骤，也不需要给密钥计数，而是利用淘汰法将不能通过检测的密钥全部淘汰，从而筛选出正确密钥。

首先，需要引入积分攻击的相关基本概念。积分攻击通过对满足特定形式的明文加密，然后对密文求和（也称为积分），通过积分值的不随机性将一个密码算法与随机置换区分开。

定义 21-16 设$f(x)$是从集合A到集合B的映射，$V\subseteq A$，则$f(x)$在集合V上的积分定义为

$$\int_V f = \sum_{x \in V} f(x)$$

通常在找到r轮积分区分器后，为方便攻击，需要将区分器的轮数进行扩展，这就是高阶积分的概念：

定义 21-17 设f是从集合$A_1 \times A_2 \times \ldots \times A_k$到集合$B$的映射，$V_1 \times V_2 \times \ldots \times V_k \subseteq A_1 \times A_2 \times \ldots \times A_k$，则$f$在$V_1 \times V_2 \times \ldots \times V_k$上的$k$阶积分定义为

$$\int_{V_1 \times \ldots \times V_k} f = \sum_{x_1 \in V_1} \ldots \sum_{x_k \in V_k} f(x_1, \cdots, x_k)$$

例如，若对任意常数c_1, c_2, c_3，f的一阶积分为$\sum_x f(x, c_1, c_2, c_3) = 0$，则$f$的二阶积分为

$$\sum_x \sum_y f(x, y, c_2, c_3) = \sum_y \left(\sum_x f(x, y, c_2, c_3) \right) = 0$$

积分攻击的主要目的是找到特定的集合V，对于相应的密文$c(x)(x \in V)$，计算相应的积分值$\int_V c$。对于随机的x，有

定理 21-3 若$X_i (0 \le i \le t)$均为$\mathbb{F}_{2^n}$上均匀分布的随机变量，则$\sum_{i=0}^{t} X_i = a$（其中a为一个常数）的概率为$\frac{1}{2^n}$。

上述定理说明，如果某些特殊形式明文对应的密文C_i，能够确定其$\sum C_i$的值，那么就可以将该加密算法与随机置换区分开来。这一能将加密算法与随机置换区分开来的区分器，就被称为积分区分器。

为了计算前述积分值，即寻找区分器，首先需要引入若干相关定义。注意，下列描述中，术语“集合”中的元素是可以重复的，这也是知名的“Multiset 攻击”名称的由来。

定义 21-18 若定义在$\mathbb{F}_{2^n}$上的集合$A = \{a_i \mid 0 \le i \le 2^n - 1\}$对于任意的$i \ne j$，均有$a_i \ne a_j$，则称$A$为$\mathbb{F}_{2^n}$上的活跃集。

定义 21-19 若定义在$\mathbb{F}_{2^n}$上的集合$B = \{a_i \mid 0 \le i \le 2^n - 1\}$满足$\sum_{i=0}^{2^n-1} a_i = a$，则称$B$为$\mathbb{F}_{2^n}$上的平衡集。

定义 21-20 若定义在$\mathbb{F}_{2^n}$上的集合$C = \{a_i \mid 0 \le i \le 2^n - 1\}$对于任意的$i$，均有$a_i = a_0$，则称$C$为$\mathbb{F}_{2^n}$上的稳定集。

下面给出上述集合满足的一些常用性质，这也是寻找一个加密算法积分区分器时所遵循的一些基本原则。

定理 21-4 不同性质字集间的运算满足如下性质：

1. 活跃/稳定字集经过双射（如可逆 S 盒、密钥加）后，仍然是活跃/稳定的；
2. 平衡字集经过非线性双射，通常无法直接确定其性质；
3. 活跃字集与活跃字集的和不一定为活跃字集，但一定是平衡字集；活跃字集与稳定字集的和仍然为活跃字集；两个平衡字集的和仍为平衡字集。

上述性质中，其中第 2 条性质是寻找积分区分器的“瓶颈”所在，如果能确定平衡集通过 S 盒后的性质，那么便有可能可以寻找到更多轮数的积分区分器，可以使用基于多项式理论的代数方法或基于比特积分采用的计数方法来对此性质进行判断。为便于介绍，在后续内容中，用字母A，B和C分别代表活跃字集、平衡字集和稳定字集。

(1) 代数方法

代数方法是指从集合元素的角度出发来研究其积分的性质。为此，需要明确有限域上多项式和多项式函数的定义：

有限域$\mathbb{F}_q$上的多项式是指$f(x) = \sum_{i=0}^{N} a_i x^i$，其中，$a_i \in \mathbb{F}_q$；而有限域$\mathbb{F}_q$上的多项式函数是指次数$\le q - 1$的多项式。因此，$\mathbb{F}_q[x]$中的任意一个多项式$f(x)$都有一个唯一的多项式函数$g(x)$与之对应，使得$g(x) = f(x) \bmod x^q - x$。在后续分析中，若无特殊说明，多项式均指多项式函数。

定理 21-5 多项式$f(x)=\sum_{i=0}^{q-1}a_i\,x^i\in\mathbb{F}_q[x]$，若$q$为某个素数的方幂，则有

$$\sum_{x\in\mathbb{F}_q}f(x)=-a_{q-1}$$

定理 21-6 若多项式$f(x)=\sum_{i=0}^{q-1}a_i\,x^i\in\mathbb{F}_q[x]$为置换多项式，则有$a_{q-1}=0$。

定理 21-5 说明，想要确定加密若干轮后某个字节是否平衡，可以转换为研究该字节和相应明文之间的多项式函数的最高项系数。

可见，活跃集对应着一个置换多项式，平衡集则表示一个多项式最高项系数为 0。

代数方法要求熟悉有限域上的多项式理论，例如置换多项式的复合还是置换多项式，置换多项式与常数的和为置换多项式等常用性质，这些性质在寻找特定算法积分区分器时都将发挥特殊的作用。

(2) 计数方法

基于计数方法求积分值最早由 Z'aba 等学者提出。由于在基于比特运算的密码算法中，上述代数方法很难实施，因此 Z'aba 等在 FSE 2008 上提出了基于比特的积分攻击，该方法实际上就是一种特殊的计数方法。

在有限域$\mathbb{F}_{2^n}$上，若$\int_V f=a$，则$\int_V f^{(i)}=a^{(i)}$，其中，$f^{(i)}(x)$和$a^{(i)}$分别表示$f(x)$和a的第i分量，显然，$f^{(i)}(x),\ a^{(i)}\in\{0,1\}$，这说明要想确定$a^{(i)}$，首先需要知道在序列$\left(f^{(i)}(x)\right)$中不同元素出现次数$N$的奇偶性：若$N$为偶数，则$a^{(i)}=0$，但若$N$为奇数，通常则无法确定相应位置是否平衡。

为了计算序列中元素重复次数的奇偶性，给出序列中不同模式的定义。

定义 21-21（常量模式） 序列$(q_0q_1\dots q_{2^n-1})$（其中$q_i\in\mathbb{F}_2$）为常量模式是指，对于任意$1\le i\le 2^n-1$，均有$q_i=q_0$，记为c。

定义 21-22（第一类活跃模式） 序列$(q_0q_1\dots q_{2^n-1})$（其中$q_i\in\mathbb{F}_2$）为第一类活跃模式是指，存在$0\le t\le n-1$，使得在序列中，2^t个0和2^t个1交替出现，记为a_t。

定义 21-23（第二类活跃模式） 序列$(q_0q_1\dots q_{2^n-1})$（其中$q_i\in\mathbb{F}_2$）为第二类活跃模式是指，存在$0\le t\le n-1$，使得在序列中，相同比特总是连续出现2^t次，记为b_t。

定义 21-24（平衡模式） 序列$(q_0q_1\dots q_{2^n-1})$（其中$q_i\in\mathbb{F}_2$）满足$\sum_{i=0}^{2^n-1}q_i=0$，则称该序列平衡。

例如，(00000000)和(11111111)均是常量模式序列(c)；(00110011)是第一类活跃模式序列(a_1)，同时也是第二类活跃模式序列(b_1)；(00110000)是第二类活跃模式序列(b_1)；而上述四个序列均为平衡模式序列。

根据定义可知，若一个序列为第一类活跃模式，则该序列一定也为第二类活跃模式；常量模式和第一类活跃模式均为平衡模式；除(b_0)外，其余第二类活跃模式均为平衡模式。

下面给出有关上述模式的若干性质。

定理 21-7 不同模式序列之间的运算遵从以下规律：

1. $\alpha\oplus c=\alpha$，其中$\alpha\in\{c,a_i,b_i\}$；
2. $a_i\oplus a_i=c$；
3. $\alpha_i\oplus\beta_j=b_{\min\{i,j\}}$，其中$\alpha,\beta\in\{a,b\}$。

容易验证，如序列$T=(0101000101010001)$是平衡序列，但不属于上述a_t、b_t和c中的任何模式，因此还需给出如下定义。

定义 21-25 设序列$M=(m_0m_1\dots m_{N-1})$，则$(M)_k$表示将序列M重复k次后得到的长度为$k\times N$的序列，即$(M)_k=\underbrace{M\dots M}_{k}$。

依据定义 21-23，序列$T=(0101000101010001)=(01010001)_2=((01)_2(0)_31)_2$。

在具体分析一个加密算法时，一般无法确定序列内各个位置的具体值，因此，通常将序列

$M=(m_0m_1\dots m_{N-1})$简记为$M=\underbrace{(v\dots v)}_{N}$，代表有$N$个未知值$v$。但要注意区分$\underbrace{(v\dots v)}_{k}$和$(v)_k$，前者表示任意$k$个值串联，后者则表示同一个值重复了$k$次。联合使用这两个符号，上述序列$T$还可以被记为$((vv)_2(v)_3v)_2$。

由于上述符号本质上是一种对于序列内周期的刻画，定理 21-7 可以表示为

$$\left(v_{2^{k_1}}v_{2^{k_1}}\right)\oplus\left(v_{2^{k_2}}v_{2^{k_2}}\right)=\left(\left(v_{2^{k_2}}v_{2^{k_2}}\right)_{2^{k_1-k_2-1}}\left(v_{2^{k_1}}v_{2^{k_1}}\right)_{2^{k_2-k_1-1}}\right)$$

其中，$k_2<k_1$。另外，容易验证的是，若将一个序列内的每个元素都乘以或加上相同的值后，不会改变原序列的模式。

综上所述，对具有r轮的迭代分组密码，实施积分攻击的一般流程为：

第一步，计算某个特殊的$r-1$轮积分值，即寻找区分器；

第二步，根据区分器，选择相应合适的明文集合，对其进行加密；

第三步，猜测第r轮密钥。使用可能的密钥进行部分解密，验证所得中间值的和是否为$r-1$轮积分值，若不是，则淘汰该密钥；

重复上述第二步和第三步，直到密钥唯一确定。

根据上述步骤可知，第一步确定$r-1$轮加密算法的某个积分值是积分攻击能否成功的关键，而选择明文量和部分解密所需要猜测的密钥量是影响攻击复杂度的主要因素。

21.2 示例分析

例 21-7 （2022 密码数学挑战赛 赛题一）

本赛题的目标是：在单密钥模型下（即整个攻击过程，所有的密文均在同一密钥下加密得到），攻击者任意选择所需要的明文并得到相应的密文，通过这些明文和密文，恢复出J算法的密钥。

J算法流程如图 21-1 所示。算法分组长度n为 32 比特，密钥K的长度t为 64 比特。令明文$m=(L_0,R_0)$，其中L_0和R_0分别为m的左 16 比特和右 16 比特，则算法计算流程如下：

$$T_{i,1}=(RK_{i-1}\&L_{i-1})\oplus R_{i-1}\oplus(i-1)_2$$
$$T_{i,2}=S(T_{i,1})=\left((T_{i,1}\lll 2)\&(T_{i,1}\lll 1)\right)\oplus T_{i,1}$$
$$T_{i,3}=P(T_{i,2})=(T_{i,2}\lll 3)\oplus(T_{i,2}\lll 9)\oplus(T_{i,2}\lll 14)$$
$$L_i=R_{i-1}\oplus(RK_{i-1}\&T_{i,3})$$
$$R_i=L_{i-1}\oplus T_{i,3}$$

其中，RK_{i-1}是轮密钥，$i=1,2,\dots,r$，r称为J算法的迭代轮数。$(i-1)_2$是整数$i-1$的二进制表示，比如$(9)_2=0000000000001001$。轮密钥由初始密钥$K=k_{63}k_{62}\dots k_0$按如下方式计算得到：

令$\begin{cases}K_3=(k_{63}k_{62}\dots k_{48})\\K_2=(k_{47}k_{46}\dots k_{32})\\K_1=(k_{31}k_{30}\dots k_{16})\\K_0=(k_{15}k_{14}\dots k_0)\end{cases}$，$K_{j+4}=\overline{K_j}\oplus(j)_2$，其中$j=0,1,2\dots$，则$RK_{2j}=K_j$，$RK_{2j+1}=\overline{K_j}$。

$c=(R_r,L_r)$定义为明文$m=(L_0,R_0)$在密钥K下的密文。

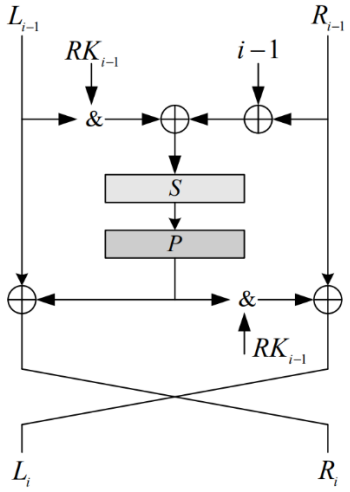


图 21-1 J算法

题目内还给出了$r=1$轮密钥的恢复方法示例。得分标准为，要求在不借助高性能计算平台的前提下，能够正确恢复出J算法的密钥，恢复的轮数越高，得分越高。

接下来采用代数攻击 SAT 方法来进行分析。由于分析重点在于向 SAT 问题的转换，而非 SAT 问题的求解本身，并且当下代数攻击也基本是使用成熟高效的 SAT 求解器来对由特定密码体制转换来的 SAT 问题进行求解，以提高分析效率，因此，可以直接选取一款成熟高效的 SAT 求解器来执行转换后的 SAT 问题的求解，例如 CryptoMiniSat 工具，CryptoMiniSat 是一款开源的先进增量 SAT 求解器。

CryptoMiniSat 提供三种执行方式：命令行、Python 脚本、库调用，由于涉及的CNF较为复杂，因此选用命令行方式读取CNF文件来进行求解。

CryptoMiniSat 采用 DIMACS 标准格式来在文件内记录CNF，形如：

```
1. p cnf 3 3
2. 1 0
3. -2 0
4. -1 2 3 0
```

其中，首行的两个数字分别代表变量数、子句数，随后的每行为一个子句，以 0 结尾，行内的数字和符号分别表示该子句内变量的编号及要求，例如，上述文件内容代表：变量 1 必须为真；变量 2 必须为假；并且，变量 1 为假、变量 2 为真或变量 3 为真。

准备好CNF文件，即可通过命令行工具进行 SAT 求解：

```
cryptominisat5 -t 4 --verb 0 baseline_cryptominisat.cnf
```

输出结果的格式形如：

```
1. s SATISFIABLE
2. v 1 -2 3 0
```

上述结果代表该CNF可解，即当设置变量 1 为真、变量 2 为假和变量 3 为真时，满足CNF中的约束。

推广至分组密码的密钥分析问题，可以使用如下 C 语言代码来对输出结果进行解析，将上述输出结果转换为分组密码的初始密钥。

```
1. char *parseResult(const char *result) {
2.     // 可解密钥长度
3.     int length = (Round + 1) / 2 * 16;
4.     if (length > 64) { length = 64; }
5.     char *key = (char *)malloc((length + 1) * sizeof(char));
6.     memset(key, 0, (length + 1) * sizeof(char));
7.
```

```
    if (result[2] == 'S') { // 求解成功
        const char *p = result;
        char str[10];
        // 解析
        for (int i = 0; i < length; i++) {
            sprintf(str, "%d", i + 1);
            p = strstr(result, str);
            if (p != NULL) {
                if (p[-1] == '-') {
                    key[length - 1 - i] = '0';
                } else {
                    key[length - 1 - i] = '1';
                }
            } else {
                printf("The result is incorrectly formatted!\n");
                break;
            }
            p = p + strlen(str);
        }
    } else { // 求解失败
        printf("UNSATISFIABLE! Please check if the cnf file is correct!\n");
    }
    return key;
}
```

接下来，聚焦于该题目，该题目是明显的选择明文攻击，因此，可以自行随机生成一个初始密钥，对全 0 明文以及两个随机明文进行r轮加密，随后对这三对明密文进行代数攻击。

读取明密文，依照代数攻击 SAT 问题转换方法的原理生成满足 DIMACS 标准格式要求的CNF文件。

调用 CryptoMiniSat 工具来对前述生成的CNF文件进行 SAT 问题求解：

```
cryptominisat5 -t 4 --verb 0 baseline_cryptominisat.cnf
```

解析结果，得出的密钥，即为一开始随机生成的初始密钥。

21.3 举一反三

分组密码体制作为现代对称密码中的重要组成部分，长期受到密码学界的关注，衍生了例如线性分析、差分分析、高阶差分分析、截断差分分析、不可能差分分析、Square 攻击、插值攻击、相关密钥攻击等一系列分析攻击方法，进一步地，诞生了当下作为常用的两类攻击方法，代数攻击以及积分攻击。

值得一提的是，求解 SAT 问题，即求解布尔方程是理论计算机中的基本问题之一，事实上，求解$\mathbb{F}_q$上n个变量的m个非线性多项式方程组本身便是一个基础的数学问题，受到了包括密码学界在内的各个理论计算机研究方向的广泛关注。除了$Grobner$基的求解办法外，由于求解$\mathbb{F}_2$上二次多元方程（MQ问题）是NP^-困难的，另一个NP^-困难的问题便是布尔表达式可满足性问题（SAT 问题）。而又因为所有的NP^-完全问题都是多项式可归约的，所以用 SAT 问题的有效求解工具（如 SAT 求解器）等都可以MQ问题的求解。