长操作数带来的一个不好的结果就是公钥方案的计算会变得极其复杂。前文已经提到，一个公开操作(比如数字签名)比使用 AES 或 3DES 对一个分组进行加密要慢 2～3 阶。此外，这三种算法家族的计算复杂度的增长大致是位长增长的三次方。例如，在一个给定的 RSA 签名生成软件内，将位长从 1 024 位增加到 3 076 位将导致执行时间变慢 $3^3 = 27$ 倍。在现代 PC 中，几十毫秒到几百毫秒的执行时间都非常常见；对大多数应用而言也不会带来什么问题。然而，在类似手机、智能卡或网络服务器等每秒需要执行多次公钥操作的、主要使用小型 CPU 的受限设备中，公钥的性能会是一个非常严重的瓶颈。第 7 章、第 8 章和第 9 章将介绍几种相对有效的公钥算法实现技术。

6.3 公钥算法的基本数论知识

下面将介绍公钥密码学中一些非常重要的技术，这些技术都来自于数论。本章主要介绍了欧几里得算法、欧拉函数及费马小定理和欧拉定理。这些算法和定理对非对称算法的学习，尤其是对 RSA 密码方案的理解，都非常重要。

6.3.1 欧几里得算法

首先介绍计算最大公约数(gcd)的问题。两个正整数 r_0 和 r_1 的 gcd 表示为

$$\gcd(r_0, r_1),$$

它指的是被 r_0 和 r_1 同时整除的最大正整数。例如 gcd(21，9) = 3。对小整数而言，gcd 的计算非常容易，就是将两个整数因式分解，并找出最大的公共因子。

示例 6.1 假设 $r_0 = 84$，$r_1 = 30$，因式分解得到

$$r_0 = 84 = 2 \cdot 2 \cdot 3 \cdot 7$$
$$r_1 = 30 = 2 \cdot 3 \cdot 5$$

gcd 是所有公共质因子的乘积：

$$2 \cdot 3 = 6 = \gcd(30, 84)$$

◇

然而，对公钥方案中使用的大整数而言，因式分解通常是不可能的。所以，人们通常使用一种更有效的算法计算 gcd，那就是欧几里得算法。此算法基于一个简单的观察，即

$$\gcd(r_0, r_1) = \gcd(r_0 - r_1, r_1),$$

其中，通常假设 $r_0 > r_1$，并且两个数均为正整数。此属性的证明非常简单：假设 $\gcd(r_0, r_1) = g$，由于 g 可以同时除 r_0 和 r_1，则可以记作 $r_0 = g \cdot x$ 和 $r_1 = g \cdot y$，其中 $x > y$，且 x 和 y 为互素的整数，即它们没有公共因子。此外，证明 $(x - y)$ 与 y 互素也非常简单。因此可以得到：

$$\gcd(r_0 - r_1, r_1) = \gcd(g \cdot (x - y), g \cdot y) = g$$

下面，使用前面例子中的数字来验证此属性：

示例 6.2 同样，假设 $r_0 = 84$，$r_1 = 30$，则 $(r_0 - r_1)$ 与 r_1 的 gcd 为：

$$r_0 - r_1 = 54 = 2 \cdot 3 \cdot 3 \cdot 3$$
$$r_1 = 30 = 2 \cdot 3 \cdot 5$$

最大公因子仍然是 $2 \cdot 3 = 6 = \gcd(30, 54) = \gcd(30, 84)$。

◇

只要满足 $(r_0 - mr_1) > 0$，迭代地使用这个过程可以得到：

$$\gcd(r_0, r_1) = \gcd(r_0 - r_1, r_1) = \gcd(r_0 - 2r_1, r_1) = \cdots = \gcd(r_0 - mr_1, r_1)$$

如果 m 选择了最大值，则此算法使用的步骤也是最少的。这种情况可以表示为：

$$\gcd(r_0, r_1) = \gcd(r_0 \bmod r_1, r_1)$$

由于第一项 $(r_0 \bmod r_1)$ 比第二项 r_1 小，通常可以交换它们：

$$\gcd(r_0, r_1) = \gcd(r_1, r_0 \bmod r_1)$$

这个过程的核心关注点在于，我们可将查找两个给定整数的 gcd 简化为查找两个较小整数的 gcd。迭代地进行这个过程，直到最后得到 $\gcd(r_l, 0) = r_l$。由于每轮迭代都保留了前一轮迭代步骤的 gcd，事实证明：最终的 gcd 就是原始问题的 gcd，即：

$$\gcd(r_0, r_1) = \cdots = \gcd(r_l, 0) = r_l$$

我们首先来看几个使用欧几里得算法计算 gcd 的例子，然后再正式地讨论此算法。

示例 6.3 假设 $r_0 = 27$，$r_1 = 21$，图 6-6 直观地给出了参数长度在每轮迭代中的缩减方式。迭代中的阴影部分为新余数 $r_2 = 6$（第一轮迭代）和 $r_3 = 3$（第二轮迭代），这两个

余数形成了下一轮迭代的输入值。请注意，最后一轮迭代的余数 $r_4 = 0$，这意味着算法的结束。◇

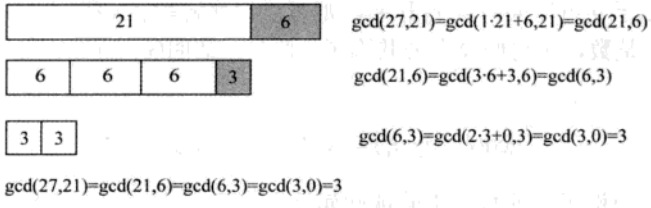

图 6-6　输入值为 $r_0 = 27$，$r_1 = 21$ 的欧几里得算法示例

观察使用较大整数计算欧几里得算法也是非常有用的，如示例 6.4 所示。

示例 6.4　假设 $r_0 = 973$，$r_1 = 301$，gcd 的计算方式为：

973=3 • 301+70	gcd(973, 301) = gcd(301, 70)
301=4 • 70+21	gcd(301, 70) = gcd(70, 21)
70 =3 • 21+7	gcd(70, 21) = gcd(21, 7)
21 =3 • 7+0	gcd(21, 7) = gcd(7, 0)=7

◇

现在，大家对欧几里得算法应该有了一定的了解，下面将给出此算法的正式描述。

欧几里得算法

输入：正整数 r_0 和 r_1，且 $r_0 > r_1$
输出：$\gcd(r_0, r_1)$
初始化：$i = 1$
算法：

1　　DO
1.1　　$i = i + 1$
1.2　　$r_i = r_{i-2} \bmod r_{i-1}$
　　WHILE　$r_i \neq 0$
2　　RETURN
　　$\gcd(r_0, r_1) = r_{i-1}$

请注意：当计算到余数 $r_i = 0$ 时，此算法结束。前一轮迭代计算得到的余数(表示为 r_{i-1})就是原始问题的 gcd。

即使处理非常长的数字(这些数字通常在公钥密码学中使用)，欧几里得算法依然高效。迭代次数与输入操作数的位数有紧密的关系。这意味着如果一个 gcd 涉及的数字都是 1 024 位，则此 gcd 的迭代次数就是 1 024 乘以一个常数。当然，只有几千次迭代的算法在当今 PC 上很容易实现，这也使得该算法在实际中的效率极高。

6.3.2　扩展的欧几里得算法

到目前为止，我们发现两个整数 r_0 和 r_1 的 gcd 的计算可以通过不断迭代地减小操作数来实现。然而事实证明，欧几里得算法的主要应用并不是计算 gcd。扩展的欧几里得算法可以用来计算模逆元，而模逆元在公钥密码学中占有举足轻重的地位。扩展的欧几里得算法(EEA)除了可以计算 gcd 外，还能计算以下形式的线性组合：

$$\gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$$

其中 s 和 t 均表示整型系数。这个等式通常也称为丢番图方程(Diophantine equation)。

现在的问题是：我们应该如何计算 s 和 t 这两个系数？此算法背后的思路为，执行标准欧几里得算法，但将每轮迭代中的余数 r_i 表示为以下形式的线性组合：

$$r_i = s_i r_0 + t_i r_1 \tag{6.1}$$

如果这个过程成功了，则最后一轮迭代对应的等式为：

$$r_l = \gcd(r_0, r_1) = s_l r_0 + t_l r_1 = s r_0 + t r_1 。$$

这也意味着最后一个系数 s_l 也是等式(6.1)所寻找的系数 s，同时 $t_l = t$。下面来看一个例子。

示例 6.5　假设某个扩展的欧几里得算法的输入值与前一个示例相同，即 $r_0 = 973$，$r_1 = 301$。左手边计算标准欧几里得算法，即计算余数 r_2，r_3…；同时还计算每轮迭代中的整数商 q_{i-1}。右手边计算满足等式 $r_i = s_i r_0 + t_i r_1$ 的系数 s_i 和 t_i；这些系数也显示在括号里。

i	$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i$	$r_i = [s_i] r_0 + [t_i] r_1$
2	973=3 • 301+70	$70 = [1] r_0 + [-3] r_1$
3	301=4 • 70+21	21=301 − 4 • 70 $= r_1 - 4(1 r_0 - 3 r_1)$ $= [-4] r_0 + [13] r_1$

(续表)

4	70=3 • 21+7	7=70 − 3 • 21 =(1r_0 − 3r_1) − 3(− 4r_0+13r_1) =[13]r_0+[− 42]r_1
	21=3 • 7+0	

此算法计算了三个参数，即 gcd(973, 301) = 7，s = 13 及 t = − 42。这三个参数的正确性可以通过以下表达式进行验证：

$$\text{gcd}(\,973, 301\,) = 7 = [13]973 + [-42]301 = 12649 - 12642$$

◇

请仔细观察上述示例最右边列中执行的代数步骤，尤其需要注意的是，右手边的线性组合都是使用前一个线性组合的结果构建的。下面将得到每轮迭代中计算 s_i 和 t_i 的递归公式。假设当前迭代对应的索引为 i，则前两轮迭代中计算的值为：

$$r_{i-2} = [s_{i-2}]r_0 + [t_{i-2}]r_1 \tag{6.2}$$

$$r_{i-1} = [s_{i-1}]r_0 + [t_{i-1}]r_1 \tag{6.3}$$

在当前第 i 轮迭代中，首先需要从 r_{i-1} 与 r_{i-2} 中计算商 q_{i-1} 和新余数 r_i：

$$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i \,.$$

这个等式也可写成：

$$r_i = r_{i-2} - q_{i-1} \cdot r_{i-1} \,. \tag{6.4}$$

回顾一下我们的目标：将新余数 r_i 表示为等式(6.1)所示的 r_0 和 r_1 的线性组合；而实现此目标的核心步骤就是：将等式(6.4)中的 r_{i-2} 用等式(6.2)替换，同时将 r_{i-1} 用等式(6.3)替换，得到：

$$r_i = (s_{i-2}r_0 + t_{i-2}r_1) - q_{i-1}(s_{i-1}r_0 + t_{i-1}r_1)$$

将这些项重新排序，就可得到想要的结果：

$$r_i = [s_{i-2} - q_{i-1}s_{i-1}]r_0 + [t_{i-2} - q_{i-1}t_{i-1}]r_1 \tag{6.5}$$

$$r_i = [s_i]r_0 + [t_i]r_1$$

等式(6.5)直观地给出了计算 s_i 和 t_i 的递归公式，即 $s_i = s_{i-2} - q_{i-1}s_{i-1}$ 和 $t_i = t_{i-2} - q_{i-1}t_{i-1}$。

这个递归表达式只对 $i \geq 2$ 的索引值有效。与其他递归一样，此递归也需要 s_0、s_1、t_0、t_1 的初始值。这些初始值(从问题 6.13 中可以得到)应该为 $s_0 = 1$，$s_1 = 0$，$t_0 = 0$，$t_1 = 1$。

扩展的欧几里得算法(EEA)

输入： 正整数 r_0 和 r_1，且 $r_0 > r_1$
输出： $\gcd(r_0, r_1)$，以及满足 $\gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$ 的 s 和 t。
初始化：
$s_0 = 1$　　$t_0 = 0$
$s_1 = 0$　　$t_1 = 1$
$i = 1$
算法：
1　　DO
1.1　　$i = i + 1$
1.2　　$r_0 = r_{i-2} \bmod r_{i-1}$
1.3　　$q_{i-1} = (r_{i-2} - r_i)/r_{i-1}$
1.4　　$s_i = s_{i-2} - q_{i-1} \cdot s_{i-1}$
1.5　　$t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$
　　WHILE　$r_i \neq 0$
2　　RETURN
　　$\gcd(r_0, r_1) = r_{i-1}$
　　$s = s_{i-1}$
　　$t = t_{i-1}$

正如前文所述，EEA 在非对称密码学中的主要应用就是计算整数的模逆。第 1.4.4 节中已经提到过这样的问题。对仿射密码而言，它需要找出密钥值 a 模 26 的逆元，而使用欧几里得算法，这个过程非常简单明了。假设我们想计算 $r_1 \bmod r_0$ 的逆元，其中 $r_1 < r_0$。从第 1.4.2 节可知，只有在 $\gcd(r_0, r_1) = 1$ 的情况下逆元才存在。因此，如果使用 EEA，则可得到 $s \cdot r_0 + t \cdot r_1 = 1 = \gcd(r_0, r_1)$，将此等式执行模 r_0 计算可得：

$$\begin{aligned} s \cdot r_0 + t \cdot r_1 &= 1 \\ s \cdot 0 + t \cdot r_1 &\equiv 1 \bmod r_0 \\ r_1 \cdot t &\equiv 1 \bmod r_0 \end{aligned} \tag{6.6}$$

等式(6.6)恰巧就是 r_1 逆元的定义。这意味着 t 本身就是 r_1 的逆元：

$$t = r_1^{-1} \bmod r_0 。$$

因此，如果需要计算逆元 $a^{-1} \bmod m$，直接使用输入参数为 m 和 a 的 EEA 即可，计算得到的输出值 t 即为其逆元。下面来看一个示例。

示例 6.6 本例的任务就是计算 $12^{-1} \bmod 67$。值 12 和 67 是互素的，即 gcd(67, 12)= 1。如果使用EEA，可以得到gcd(67, 12)= 1 = $s \cdot 67 + t \cdot 12$ 中的系数 s 和 t。如果初始值 $r_0 = 67$，$r_1 = 12$，则此算法的计算过程为：

i	q_{i-1}	r_i	s_i	t_i
2	5	7	1	−5
3	1	5	−1	6
4	1	2	2	−11
5	2	1	−5	**28**

于是可以得到如下线性组合

$$-5 \cdot 67 + 28 \cdot 12 = 1$$

如上所示，12 的逆元为

$$12^{-1} \equiv 28 \bmod 67$$

验证这个结果的等式为：

$$28 \cdot 12 = 336 \equiv 1 \bmod 67$$

◇

请注意：通常不需要系数 s，而且实际中也一般不计算该值。还有一点需要注意的是，该算法结果中的 t 可以是一个负数，但这个结果仍然正确。这种情况下，我们必须计算 $t = t + r_0$，这是一个有效的操作，因为 $t \equiv t + r_0 \bmod r_0$。

下面将讨论如何利用 EEA 计算伽罗瓦域内的乘法逆元。在现代密码学中，这部分内容与 AES 中 S-盒的起源以及椭圆曲线公钥算法有着重要关联。EEA 完全可以当做多项式(而不是整数)来使用。如果想计算有限域 $GF(2^m)$ 中的一个逆元，算法的输入就是域元素 $A(x)$ 和不可约多项式 $P(x)$。EEA 计算辅助多项式 $s(x)$和 $t(x)$，以及最大公约数 gcd($P(x)$，$A(x)$)，且这些结果满足：

$$s(x)P(x) + t(x)A(x) = \gcd(P(x), A(x)) = 1$$

注意，由于 $P(x)$是不可约多项式，所以 gcd 总是等于 1。如果对上述等式左右两边同时约简，执行模 $P(x)$计算，就会发现辅助多项式 $t(x)$等于 $A(x)$的逆：

$$s(x)0 + t(x)A(x) \equiv 1 \bmod P(x)$$

$$t(x) \equiv A^{-1}(x) \bmod P(x)$$

下面将给出在小型域 $GF(2^3)$ 内使用扩展的欧几里得算法的示例。

示例 6.7　计算基于 $P(x)=x^3+x+1$ 的有限域 $GF(2^3)$ 中 $A(x)=x^2$ 的逆元。$t(x)$多项式的初始值为：$t_0(x)=0$，$t_1(x)=1$。

迭代轮数	$r_{i-2}(x)=[q_{i-1}(x)]r_{i-1}(x)+[r_i(x)]$	$t_i(x)$
2	$x^3+x+1=[x]x^2+[x+1]$	$t_2=t_0-q_1t_1=0-x1\equiv x$
3	$x^2=[x](x+1)+[x]$	$t_3=t_1-q_2t_2=1-x(x)\equiv 1+x^2$
4	$x+1=[1]x+[1]$	$t_4=t_2-q_3t_3=x-1(1+x^2)$
		$t_4\equiv 1+x+x^2$
5	$x=[x]\,1+[0]$	Termination since $r_5=0$

注意，多项式系数的计算在 $GF(2)$ 内，而由于加法操作与乘法操作相同，因此负系数(比如 $-x$)都可以替换为一个正数。每轮迭代中得到的新余数和商就是上面括号里的数值。多项式 $t_i(x)$ 是根据本节前面介绍的计算整数 t_i 的递归公式得到的。如果余数为 0，则 EEA 终止。在本例中，当索引为 5 时 EEA 结束。计算得到的最后的 $t_i(x)$的值，即 $t_4(x)$，就是所要求的逆元：

$$A^{-1}(x)=t(x)=t_4(x)=x^2+x+1.$$

下面是检查 $t(x)$的确是 x^2 的逆元的过程。这里使用的特性为：$x^3\equiv x+1 \bmod P(x)$ 和 $x^4\equiv x^2+x \bmod P(x)$：

$$\begin{aligned} t_4(x)\cdot x^2 &= x^4+x^3+x^2 \\ &\equiv (x^2+x)+(x+1)+x^2 \bmod P(x) \\ &\equiv 1 \bmod P(x) \end{aligned}$$

◇

注意：在 EEA 的每一轮迭代中，人们通常使用非常长的除数(与上面显示的短除数不同)来确定新的商 $q_{i-1}(x)$和新的余数 $r_i(x)$。

第 4 章的表 4-2 中的逆元都是通过扩展欧几里得算法得到的。

6.3.3　欧拉函数

下面来学习公钥密码体制(尤其是 RSA)中非常有用的一个工具。在环 $\mathbb{Z}_m=\{0,1,\ldots,m-1\}$中，我们感兴趣的问题(这个问题在此处显得有点奇怪)就是这个集合中有多少个数字与 m 互素。这个数目可以由欧拉(*Euler's Phi*)函数给出，其定义如下：

定义 6.3.1　欧拉函数

$\mathbb{Z}_m$ 内与 m 互素的整数的个数可以表示为 $\Phi(m)$。

首先来看几个示例，并通过手动统计 $\mathbb{Z}_m$ 内所有互素的整数来计算欧拉函数。

示例 6.8　假设 $m=6$，对应的集合为 $\mathbb{Z}_6=\{0, 1, 2, 3, 4, 5\}$。

gcd(0, 6) = 6
gcd(1, 6) = 1 ★
gcd(2, 6) = 2
gcd(3, 6) = 3
gcd(4, 6) = 2
gcd(5, 6) = 1 ★
由于该集合中有两个与 6 互素的数字，即 1 和 5，所以欧拉函数的值为 2，即 $\Phi(6)=2$。

◇

再看一个例子。

示例 6.9　假设 $m=5$，对应的集合为 $\mathbb{Z}_5=\{0, 1, 2, 3, 4\}$。

gcd(0, 5) = 5
gcd(1, 5) = 1 ★
gcd(2, 5) = 1 ★
gcd(3, 5) = 1 ★
gcd(4, 5) = 1 ★
由于该集合中有 4 个与 5 互素的数字，所以 $\Phi(5)=4$。

◇

从上面的例子可以推测出，如果数值非常大的话，将集合内的元素从头至尾都处理一遍并计算 gcd 的欧拉函数的计算方法会非常慢。实际上，使用这种最直接的方法计算公钥密码学中使用的非常大的整数对应的欧拉函数是非常困难的。幸运的是，如果 m 的因式分解是已知的，则存在一个更简单的计算方法，如以下定理所示。

定理 6.3.1　假设 m 可以因式分解为以下数的连乘

$$m = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_n^{e_n},$$

其中，p_i 表示不同素数的个数，e_i 表示正整数，则有

$$\Phi(m) = \prod_{i=1}^{n}(p_i^{e_i} - p_i^{e_i-1})。$$

即使对很大的整数 m 而言，n 的值(即不同素因子的个数)也总是很小，所以评估乘积符号 $\prod$ 在计算上也是非常简单的。下面列举一个使用这个关系计算欧拉函数的例子。

示例 6.10　假设 m=240，240 因式分解对应的连乘形式为：

$$m = 240 = 16 \cdot 15 = 2^4 \cdot 3 \cdot 5 = p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3}$$

其中有三个不同的质因子，即 $n = 3$，则欧拉函数的值为

$$\Phi(m) = (2^4 - 2^3)(3^1 - 3^0)(5^1 - 5^0) = 8 \cdot 2 \cdot 4 = 64$$

这意味着在范围{0，1，…，239}内存在 64 个整数与 m = 240 互素。而替代方法需要计算 240 次 gcd，即使对较小的整数而言，这个计算过程也会非常慢。

◇

需要强调的一点是，在用这种方法快速计算欧拉函数时，我们必须知道 m 的因式分解。在第 7 章我们将了解到，这个特性也是 RSA 公钥方案的核心；相反地，如果已知某个整数的因式分解，就可以计算出欧拉函数并解密密文。如果因式分解未知，也就不能计算欧拉函数，也无法解密。

6.3.4　费马小定理与欧拉定理

下面将介绍公钥密码学中非常有用的两个定理，首先介绍费马小定理(Fermat's Little Theorem[1])。此定理在素性测试和公钥密码学的其他很多方面都非常有用。在做指数模整数操作时，此定理得到的结果将非常令人惊讶。

1. 请不要把费马小定理和费马最后定理混为一谈，费马最后定理是一个非常著名的数论问题，并于 20 世纪 90 年代(即自它诞生 350 年之后)才被论证。

定理 6.3.2　费马小定理

假设 a 为一个整数，p 为一个素数，则

$$a^p \equiv a(\mathrm{mod}\, p)$$

请注意，有限域 $GF(p)$ 上的算术运算都是通过 mod p 实现的，因此，对有限域 $GF(p)$ 内的所有整数元素 a 而言，此定理始终成立。此定理也可表示为以下形式：

$$a^{p-1} \equiv 1(\mathrm{mod}\, p)$$

这种形式在密码学中非常有用，其中一个应用就是计算有限域内某个元素的逆元。此等式也可以写成 $a.a^{p-2} \equiv 1(\mathrm{mod}\, p)$，这个表示形式正是乘法逆元的定义。因此，我们立刻可以得到反转整数 a 模一个素数的方法：

$$a^{-1} \equiv a^{p-2}(\mathrm{mod}\, p) \tag{6.7}$$

请注意，只有在 p 为素数时这种反转方法才成立。下面来看一个示例：

示例 6.11　假设 $p=7$，$a=2$，计算 a 的逆元可以表示为：

$$a^{p-2} = 2^5 = 32 \equiv 4 \bmod 7$$

结果的验证也很容易：$2 \cdot 4 \equiv 1 \bmod 7$。

◇

计算等式(6.7)中的指数运算通常比使用扩展的欧几里得算法要慢很多。然而，使用费马小定理在有些情况下也具有一定的优势，比如在智能卡或其他拥有快速指数硬件加速器的设备中。但这些情况都不常见，因为很多公钥算法都需要指数计算，这在后续章节中会看到。

将费马小定理的模数推广到任何整数模，即不一定为素数的模，就可得到欧拉定理。

定理 6.3.3　欧拉定理

假设 a 和 m 都是整数，且 gcd(a，m)=1，则有：

$$a^{\Phi(m)} \equiv 1(\mathrm{mod}\, m)$$

由于这个定理对模数 m 适用，所以它也适用于整数环 $\mathbb{Z}_m$ 内的所有整数。下面来看一个使用欧拉定理计算较小数字的例子。

示例 6.12　假设 $m=12$，$a=5$。首先计算 m 的欧拉函数：

$$\Phi(12)=\Phi(2^2\cdot 3)=(2^2-2^1)(3^1-3^0)=(4-2)(3-1)=4$$

现在验证欧拉定理：

$$5^{\Phi(12)}=5^4=25^2=625\equiv 1 \bmod 12$$

◇

很容易看出来，费马小定理是欧拉定理的一个特例。如果 p 为一个素数，则 $\Phi(p)=(p^1-p^0)=p-1$ 成立。如果将这个值用于欧拉定理，则可得到：$a^{\Phi(p)}=a^{p-1}\equiv 1(\bmod p)$，而这正是费马小定理。

6.4　讨论及扩展阅读

通用的公钥密码学　Whitfield Diffie 和 Martin Hellman[58]的文章对非对称密码学具有里程碑的意义。Ralph Merkle 独立发明了非对称密码学的概念，并提出了完全不同的公钥算法[121]。下面是公钥密码学历史中一些良好的说明。强烈推荐 Diffie 在[57]中提出的方法。另一个关于公钥密码学的精辟概述就是[127]。[100]中描述了椭圆曲线密码学的详细历史，包括 20 世纪 90 年代中 RSA 和 ECC 之间的激烈竞争，所讲的内容极具教育意义。关于非对称密码学最近的发展状况都记录在公钥密码(PKC)系列研讨会中。

模算术运算　关于本章介绍的数学知识，第 1.5 节中推荐的关于数论的入门性书籍都是很好的扩展阅读材料。在实际中，扩展的欧几里得算法也极其重要，因为几乎所有公钥方案的实现都包含这部分内容，尤其是模逆元。此方案一个重要的加速技术就是二进制 EEA，而它优于标准 EEA 之处在于用位移位代替了除法。这对公钥方案使用的非常长的整数而言，是非常具有吸引力的。

替代公钥算法　除了已确立的三种非对称方案外，还存在一些其他的非对称方案。首先，有些算法已经被破译或被认为是不安全的，比如背包方案。其次，有些算法是已成熟算法的推广，比如超椭圆曲线，代数变体或非-RSA 基于因式的方案。所有这些方案都使用相同的单向函数，即整数因式分解或某些群内的离散对数。第三，有些非对称算法基于不同的单向函数。人们感兴趣的单向函数有四种：基于哈希的，基于代码的，基于点阵的