# 基础数论算法

我们在这一章介绍一些基本的数论算法，这些算法相当基础，不光在整数分解算法中频繁使用，在其他应用场景中也能见到它们的身影. 当然，我们在这一部分不可能把所有需要的数论算法都列举出来，只能选取几个代表性的算法来展示.

我们选取了欧几里得算法、Eratoshenes筛法、模平方根算法和快速模幂运算. 几乎所有的整数分解算法中都使用欧几里得算法来得到非平凡因子，而扩展的欧几里得算法可以快速地求模逆；许多整数分解算法需要素数表，而Eratoshenes筛法就可以很快地给出小于某个正整数的全部素数；二次筛法需要求解二次同余方程，因此模平方根算法至关重要；快速模幂运算可以在对数级的时间内得到的值，这是Pollard 方法的基本运算，事实上，基于同样的技术，我们也可以加速椭圆曲线上点的数乘运算.

## 欧几里得算法

### 基本形式

欧几里得算法用来求出两个正整数的最大公因数，任何一本初等数论的教科书都有详细的介绍，这里不再赘述原理和细节，直接给出欧几里得算法[10]的伪代码：

|  |
| --- |
| **算法1.1 欧几里得算法** |
| **输入**：正整数  **输出**：的最大公因数  1. 如果，返回，终止算法.  2. 置，，，跳转到步骤1. |

欧几里得算法还有许多变种，比如更适合计算机硬件的Stein算法等等.

### 扩展的欧几里得算法

欧几里得的算法本身只能给出正整数的最大公因数，但很多时候我们需要使用线性表示，也就是求出整数使得. 这时就要使用扩展的欧几里得算法[10].

|  |
| --- |
| **算法1.2 扩展的欧几里得算法** |
| **输入**：正整数  **输出**：  1. 置. 如果，置，终止算法；否则置.  2. 如果，置，终止算法.  3. 令且. 接着置，跳转到步骤2. |

扩展的欧几里得算法非常重要，它还用来求模逆（前提是互素）只要找到整数使得，就有.

## Eratosthenes筛法

Eratosthenes筛法是一个相当古老的方法，它可以给出不超过正整数的全部素数. 之所以把这个算法称之为“筛法”，是因为它就像筛子一样，将到之间所有的合数都排除出去，最后剩下的就是素数.

具体来说，我们一开始把到之间所有的正整数全部列出来，首先将所有2的倍数（不包括2本身）全部划去，接着寻找下一个没有被划去的数，再将所有的倍数（不包括本身）全部划去，依次类推，直到. 此时，到之间没有被划去的数就是素数. 我们举个简单的例子，筛选2到36之间的素数. 因为，我们只需要列举不大于6的没有被划去的数，就能排除所有合数了.

首先，我们将所有2的倍数（不包括2本身）全部划去（我们把数字所在的格子涂黑，来表示将这个数划去）得到如下的结果：

表1.1 筛去2的倍数

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 |

接下来，数2的下一个没有被划去的数是3，保留3这个数，把3的倍数全部划去，得到下面的结果：

表1.2 筛去3的倍数

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 |

数3的下一个没有被划去的数是5，保留数5，把5的倍数全部划去，得到下面的结果：

表1.3 筛去5的倍数

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 |

下一个没有被划去的数7大于6，终止算法. 于是，剩下的没有被划去的数就是素数了. 我们便得到了2到36之间的全部素数：2,3,5,7,11,13,17,19,23,29,31.

这个算法的时间复杂度为，已经非常接近线性时间复杂度了，是一个相当高效地素数生成算法. 当然，也有更加高效的线性时间复杂度的素数筛法，这里就不介绍.

## 模平方根算法

对于一般的模平方根问题，我们需要知道的完全分解式，然后再利用中国剩余定理才能求解. 可以证明，模平方根问题和整数分解问题是等价的. 因此，在这里我们只讨论平方根模素数和素数幂这两种情况[2].

### 模为素数的情况

设为奇素数，整数不是的倍数，且Legendre符号，二次同余方程

|  |  |
| --- | --- |
|  | （1-1） |

有两个解[2]. 当时，解为；当时，就要分两种情况讨论. 因为，我们有

|  |  |
| --- | --- |
|  | （1-2） |

当右边为“”时，解为，否则，式（1-2）右边为“”，此时的解应为[2].

最复杂的情况就是. 理论上我们也可以给出解的表达式，但这里我们给出一个模平方根算法——Cipolla方法[29]. 随机选取一个正整数，使得Legendre符号，设是的一个根，即，则就是的一个解.

举例来说，我们求解方程. 取得到.

于是，即. 计算

|  |  |
| --- | --- |
|  | （1-3） |
|  |
|  |
|  |
|  |
|  |

注意，上面的计算都是模17意义下的. 于是，我们得到.

### 模为素数幂的情况

在一定的条件下，我们可以通过的解，来寻找

|  |  |
| --- | --- |
|  | （1-4） |

的解. 这里正整数. 这里默认. 当为奇素数时，方程有两个解.

但是，当为奇数时，情况就变得复杂了. 当时，如果，方程只有两个解，但方程就无解. 当时，如果时，方程有四个解，对于其他情况的，方程都没有解.

大多数初等数论的教科书会给出二次同余方程的解法，并且考虑到效率问题，我们在二次筛法中，一般不会实现模为素数幂的模平方根算法，这里就不介绍具体的解法了.

## 快速模幂运算

如果不采用特殊的技巧，计算的时间复杂度为，如果对指数二进制展开，就可以在的时间内得到计算结果，根据计算方式的不同，快速模幂运算又分为从右向左取幂和从左向右取幂两个版本.

在这里，我们只给出从右向左取幂的快速模幂运算[10]：

|  |
| --- |
| **算法1.3 快速模幂运算** |
| **输入**：正整数，非负整数  **输出**：  1. 置，如果，输出，终止算法；否则，置.  2. 如果为奇数，置.  3. 置. 如果，输出并终止算法；否则，置，然后跳转到步骤2. |

上述的快速模幂运算其实不加任何改造，就可以推广到一般的乘法群上. 在计算椭圆曲线上的点的数乘时，也可以用上面的计算技巧.

## 小结

在这一章里，我们粗略地介绍了一些重要和基本的数论算法，但没有详细介绍背后的数学原理，以及具体的计算机代码实现的技巧. 许多关于基础数论算法的知识点都很琐碎，但了解算法的优化细节对工程实践来说非常有帮助. 许多成熟的科学计算库和数学软件对这些基本的算法都进行了高度的优化. 想要更加深入地了解更多基础数论算法，可以参考Henri Cohen的专著[10].

# 特殊整数分解算法

在整数分解的算法中，有一类特殊的因子分解方法，它们的共同特点就是找到待分解整数的小因子. 事实上，设待分解的合数为，我们只要找到所有的小于等于的素数因子，就可以完全分解了[8]. 我们有下面的引理.

**引理** 正整数最多只有一个大于的素数因子.

反证法可以很容易证明这个引理，如果有两个大于等于的素因子，这两个素数相乘显然大于，和假设矛盾.

**推论** 正整数最多只有个大于的素数因子.

上述引理和推论表明正整数的“大”素因子很少，要找到小于等于的全部素数因子，最朴素的方法就是“试除法”，用所有小于等于的素数去除整数，通过余数判断是否为的素因子. 试除法的效率非常低，大约要做次除法，其中数论函数表示不大于的素数的个数.

因此，高效地找到一些小因子对整数的完全分解是有帮助的. 下面介绍的一些特殊整数分解方法，有的可以快速地找到合数的小因子，有的适用于特殊形式的整数. 在实际应用的整数分解中，我们通常先使用这些特殊整数分解算法找到一部分容易分解的因子，然后再使用一般整数分解方法继续完成整数的完全分解.

## Pollard 方法

Pollard 方法是一个非常有趣的概率性整数分解算法，该方法是J.M. Pollard在1975年提出的[13]，能够非常快速地找到合数的较小的因子（不一定是素数）下面我们给出方法的核心思想和实现原理，并对该方法的运行效率进行简单的分析.

### 算法核心思想

Pollard 方法非常巧妙. 设待分解的合数为，并且有一个较小的素因子. 该方法需要一个伪随机数序列，序列满足一个简单且方便计算的递推式（假设函数是多项式函数）模的剩余类是有限集，序列中所有元素落在，显然是个循环序列（从某一项开始）.

事实上，我们关注的并不是序列的周期，而是序列在模意义下的新序列，满足递推式，也是个有限集，也会是一个循环序列. 也就是说，存在. 由于的周期不大于的周期，即的同时，大概率的有，而，这时计算就可以得到合数的素因子.

在实际应用中，合数的素因子是未知的，我们只能计算序列，根本不知道的值. 如何确定呢？我们可以计算序列中，如果，说明，即；如果，说明.

举一个简单的例子，选取，序列的递推关系如下：. 序列满足



图2.1 循环序列和

循环序列和的周期明显不同，当时，，但有且，于是计算得到了非平凡因子.

通过和不同的周期关系，如果，据此找到合数的非平凡因子，就是Pollard 方法的核心原理. 如图2.1所示，循环序列和的各项用图表示出来很像希腊字母“”，这就是该方法得名的原因.

### Pollard 方法的原始版本

为了平衡序列的伪随机性与计算复杂性，使得算法的效率更高，我们选取二次式作为序列的递推关系式. 在后面的讨论中，都是指上述形式的二次式.

是很容易理解的. 事实上，当时， 会有形如的平凡解[9]. 取其他的值可以保证序列足够随机.

在实际的整数分解中，的素因子是未知的. 考虑模的意义下的新序列，. 存在，使得当时，有，但，即序列从第M项开始进入循环节，循环周期为T. 我们并不需要精确地找到周期T，只要找到，相差的整数倍即可. 初看上去，我们似乎需要对两个变量进行“穷举”，Pollard 方法用的周期检测方法只需要对一个变量进行“穷举”.

Pollard 方法的实现方式，就是找到这样的，，此时是周期T的整数倍[13]. 这样，只需要“穷举”一个变量依次计算，当时停止，就找到的一个非平凡因子. 下面给出算法描述[29]：

|  |
| --- |
| **算法2.1 Pollard 方法** |
| **输入**：一个合数，且不是素数的方幂，  **输出**：的一个非平凡因子  1. 随机选取，多项式  2. 对于    如果，选取下一个  如果，输出并终止算法  如果，则返回步骤1选取下一个和整数 |

由于计算最大公因数的欧几里得算法和计算相比速度要慢，而大多数情况下，gcd的结果都是1，因此可以通过将个连乘后再计算gcd[13]. 对于某个固定的正整数，令

|  |  |
| --- | --- |
|  | （2-1） |

然后依次计算直到找到.

下一节将要介绍一个更加有效的Pollard 方法改进版本.

### Brent的改进方法

Brent于1980年在文献[14]中给出了方法的一个改进版本，可以更高效地找到伪随机递推周期序列中两个重复的元素. 不同于寻找，Brent的改进方法寻找的等式是，其中. 在Henri Cohen的专著[10]中详细描述了Brent改进方法的数学原理，这里只给出Brent改进方法的描述[8].

|  |
| --- |
| **算法2.2 Brent改进方法** |
| **输入**：一个合数，且不是素数的方幂.  **输出**：的一个非平凡因子  1. [初始化] 令.  （变量和分别存储和，变量存储，变量）  2. [欧几里得算法] 置.  如果，跳转到步骤3；  如果，输出，算法结束；  如果，算法失败，终止算法.  3. [继续] . 如果，置.  置，返回步骤2. |

### 算法效率和适用范围

Pollard 方法的时间复杂度依赖于待分解合数的较小的素因子，一般可以在次迭代后找到非平凡因子（可能不是素因子）对于该方法的渐进时间复杂度的理论分析，可以参考文献[10].

下面是我在个人计算机上分解形如形式的数，其中和都是素数，并且在十进制下位数相同. 结果如下表:

表2.1 Pollard 方法测试数据

|  |  |  |
| --- | --- | --- |
| 合数n | 素因子p（13位） | 耗时（单位：秒） |
| 53710838125727488749468899 | 8780437221649 | 13.465 |
| 22336790869621579139586221 | 8105423406547 | 8.217 |
| 7368395104324019116258727 | 1823064324533 | 7.561 |
| 14431869638102713479337961 | 2043084530567 | 5.702 |
| 42258421578247817705821417 | 5136949767227 | 6.577 |
| 59625482488861614659663279 | 6858136652231 | 7.607 |
| 7946088226254842457755657 | 1725991332347 | 10.685 |
| 62320538233306266158870861 | 8126291374427 | 15.574 |
| 22567115633893514054584373 | 3332585320667 | 10.904 |
| 11957064344478288825838259 | 1557972880691 | 10.732 |

对于个人计算机来说，Pollard 方法可以在10s左右的时间找到十进制13位的非平凡因子. Pollard 方法最成功的案例是，Richard Brent使用方法的一个变体找到了费马数的一个素因子1238926361552897.

Pollard 方法的渐进时间复杂度约为，素因子每增加2位，平均用时就会增加10倍以上. 我们可以估计Pollard 方法在不同规模素因子下的耗时.

表2.2 Pollard 方法耗时估计

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 素因子p的位数（十进制） | 13 | 15 | 17 | 19 |
| 估计用时 | 10s | 3min | 35min | 5h 50min |

上面的估计只是理想情况，在真实环境中有可能需要更长时间. 从上述实验来看，我们建议只在下面两种情形下使用Pollard 方法：

1. 直接分解十进制不超过30位的整数.
2. 寻找规模小于15位的“小”因子.

在我的计算机上进行的实际测试表明，Brent改进方法找到十进制10位、11位的因子所需的迭代次数少于，而大于11位时，算法的表现就不稳定了. 因此在情况（2）下需要改造Pollard 方法，增加迭代次数限制. 对于其他情况，应该使用更有效的整数分解算法，比如后面介绍的ECM方法和二次筛法.

## Pollard 方法

Pollard 方法是John Pollard在1974年提出的一个整数分解算法[15]，这是专门针对一类特殊合数设计的因子分解算法. 如果待分解的合数有某一个素因子，且只有较小的素因子时，该算法就能以高概率找到这个素因子.

### 核心原理——费马小定理

从初等数论的角度来看，Pollard 方法所依赖的数学原理就是费马小定理. 从抽象代数的角度来看，Pollard 方法就是乘法群上使用拉格朗日定理的自然结果. 我们设待分解的合数为，素数是的一个素因子，并且. 根据费马小定理

|  |  |
| --- | --- |
|  | （2-2） |

那么我们就有，其中为任意正整数[2].

如果我们能找到一个尽可能小的数，它是的倍数，也就是存在某个正整数使得，那么有. 我们知道，现在又找到了使得. 我们记正整数，那么就有是的一个真因子，这样就分解成功分解了合数[2].

### 算法实现原理

Pollard 方法的关键就是找到这样的，满足，从而使得上述过程中的表达式成立. 这里的整数可以随机取值，不过一般为了方便起见，我们可以直接取作为底数. 需要注意的是，合数的素因子我们是不知道的，因此我们需要用类似于穷举搜索的方法，去尽可能“覆盖”的每一个因子.

一个比较朴素的办法就是给出一个界限，我们寻找合数满足的素因子. 假设这样的素因子是存在的，那么一定存在正整数,有，于是取，就是满足条件的数.

因此，原始的Pollard 方法如下[2]：

设待分解的合数为，以及给定一个整数

随机选取底数

整数，依次计算

如果，那么就取下一个值.

如果，就是合数的一个非平凡因子，算法结束.

如果对于任意的，得到的最大公因数都为1，那么算法失败，说明界限的值太小了，这时可以适当扩大然后在重复上述步骤，继续尝试分解.

我们用合数来举例，取底数，运算过程如表2.3：

**表2.3 Pollard 方法运算过程示例**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 2 | 3 | 1 |
| 3 | 63 | 1 |
| 4 | 81423 | 1 |
| 5 | 45689 | 1 |
| 6 | 63166 | 1 |
| 7 | 85839 | 71 |

事实上，由于显然后者的分解式中的素因子更小，所以会最先被分解出来.

Pollard 方法的版本不止一个，可以用最小公倍数来代替，也可以通过连乘的方法来提高速度[10]：

令，则，

设

然后依次计算找出的非平凡因子.

接下来我们给出Pollard 方法的其中一个实现版本.

### 方法的一个实现版本

Pollard 方法在具体实现上有多个版本，文献[15]为了方法的普适性将算法分成了两个阶段，其中算法第一阶段的工作和上一节描述的原理大致相同，如果算法的第一阶段没有找的合数的非平凡因子，算法就会进入第二阶段搜索中可能的大素因子. 但是通常来说算法进入第二阶段后成功率往往不高，如果可以成功，也可以通过修改算法的初始参数（也就是光滑界）来使得算法在第一阶段就得到非平凡因子. 因此在实际中我们所实现的Pollard 方法只是文献[15]描述的第一阶段算法. 下面我们给出Pollard 方法的其中一个实现版本.

通过之前的讨论，我们可以看出Pollard 方法的核心是猜测待分解合数的某个素因子的分解式，根据算术基本定理，可以表示成如下的形式

|  |  |
| --- | --- |
|  | （2-3） |

其中为正整数，为某个非负整数序列，表示第个素数. 这个意味着我们可以“一个素数次幂一个素数次幂”的猜测的分解式，从而找到. 对于任意的整数，对于上述，显然有，那么

接着，计算就可以得到的非平凡因子.

在实际应用该改进算法时，可以通过迭代计算得到，具体的流程如下[3]：

|  |
| --- |
| **算法2.3 Pollard 方法** |
| **输入：**待分解的合数，光滑界  **输出：**的一个非平凡因子  1. 构建素数表  2. 随机选取，  3. 对于  ，表示第个素数        如果，则进入下一个  如果，则输出并终止算法  如果，则返回步骤1并选取另一个 |

在这个版本的算法中，如果待分解的合数的某个素因子的分解表达式中所有的素因子都在素数表，合数就能以高概率成功分解.

### 适用范围和效率分析

设待分解的合数的最小素因子为，Pollard 方法的主要耗时在于穷举的全部素因子和求最大公因数. 设数论函数表示不大于的素数的个数，是的最大素因子，那么算法只需迭代次就可以得到结果.

在个人计算机上，我们使用方法时要给出一个光滑界，然后生成一个素数表，表中最大的数小于等于. 如果把求最大公因数的次数作为衡量算法时间复杂度的标准，那么运行时间规模是，通常的取值通常在到之间，一般不超过.

该算法还会在某些情况下失败. 最常见的情形就是有一个素因子远远大于，这时Pollard 方法就不太适用了. 还有一种情况非常罕见，就是合数的所有素因子减1后的有相同的光滑界，算法很可能会得到平凡因子本身，此时可以修改初始的底数. 最坏的情况是，其中都是素数且的规模大体相同[10]. 此时算法的时间复杂度为. 在实际中甚至可能比试除法的效率还要低（因为试除法只需要做除法，而方法需要多次求最大公因数和模幂运算）

### 应用举例

我们找的第一个例子是随机产生的一个看上去比较小的合数

33432649280047353540365856155422567745554042733243971991

= 31618624099079 1057372046781162536274034354686893329625329

这是一个十进制56位的合数. 我们设 31618624099079，则

使用方法，底数，光滑界，当迭代至876768轮时，

在我自己的个人计算机上分解这个数花了6.64秒.

下面是一个在实际应用中得到的RSA公钥

-----BEGIN PUBLIC KEY-----

MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCAhifO04qYDXZUVKxd/vwQGV9v

75s1tSt0LbziQZw0CAo+8+lnP+pN1in/OCFVAx6m3LqDctQsGGLzKyvuR+FX+nFQ

xURjUDXzZvfWgjT1b6JBgOtqAKD4XGWq60VbjtKPIoU3bNp4b4xljP6zdS81BKcl

bqPb0i7vICZ9FW+rUQIDAQAB

-----END PUBLIC KEY-----

其中十进制表示的1024比特（十进制308位）的模数：

90252653600964453524559669296618135272911289775949194922543520872164147768650421038176330053599968601135821750672685664360786595430028684419411893316074286312793730822963564220564616708573764764386830123818197183233443472506106828919670406785228124876225200632055727680225997407097843708009916059133498338129

加密指数： 65537

这个数看起来很大，但是使用Pollard 方法可以非常容易的分解出来. 我们设底数，当光滑界时，就分解出了的一个素因子. 它的十进制表示如下：

1719620105458406433483340568317543019584575635895742560438771105058321655238562613083979651479555788009994557822024565226932906295208262756822275663694111

事实上，我们有，在方法迭代至第379轮时就可以通过计算最大公因数得到非平凡因子. 在我的个人计算机上分解这个1024比特的数甚至只需要0.1秒！

从上面两个例子中我们感受到Pollard 方法运行时间不依赖于带分解的合数本身的大小，而是的最大素因子.

## Williams 方法

Williams 方法（以下简称方法）是一种特殊的因子分解方法，由H. C. Wiliams于1984年提出[24]. 该方法和Pollard 有着相似的特点，其运行效率不取决于被分解的合数本身，而是依赖于合数最小素因子的特性. 如果的所有素因子都比较“小”，方法就可以在短时间内分解出素数.

### Lucas序列简介

方法的核心原理与Lucas序列的特殊性质有关. 我们在介绍方法之前，需要给出Lucas序列的一些性质和常用结论. 限于篇幅，这里不给出定理和结论的具体证明，有兴趣的读者可以参考[23][24].

Lucas序列是指下面的整数序列[23]

和

其中是一元二次方程的两个根，.

我们记判别式. 显然有. Lucas序列满足下列的递推关系[23]：

|  |  |
| --- | --- |
|  | （2-4） |
|  | （2-5） |

由归纳法可以得出Lucas序列和都是整数序列（尽管方程的根可能不是整数），著名的Fibonacci数列就是时的特例.

我们列出Lucas序列如下的常见性质[24]：

|  |  |
| --- | --- |
|  | （2-6） |
|  | （2-7） |
|  | （2-8） |

这些性质在快速计算Lucas序列某一项中有非常重要的作用. 最后我们给出一个定理，它是方法的理论基础.

**定理**[24] 如果是奇素数，不整除，令为关于的Legendre符号，则

|  |  |
| --- | --- |
|  | （2-9） |

### 方法原理与描述

接下来我们说明方法是如何利用Lucas序列分解整数的. 假设待分解的合数有一个素因子. 根据上一节的定理，假设且，如果找到一个正整数，使得，那么就有，于是.

找到这样的正整数的思路和Pollard 方法完全一致，设

|  |  |
| --- | --- |
|  | （2-10） |

其中表示第个素数，为非负整数. 对于任意的整数，对于上述，有，令

我们就找到了这样的.

方便起见，我们可以令，定义，，这样上一节的定理中的（2-9）就描述为

|  |  |
| --- | --- |
|  | （2-11） |

我们就有，于是计算就可以得到的一个非平凡因子. 我们注意到，由式（2-8），序列还有这样的性质：

|  |  |
| --- | --- |
|  | （2-12） |

这一性质使得序列可以迭代地进行计算，因此对于快速计算Lucas序列来说，的性质比要好得多，方法利用的就是序列.

我们接下来给出方法的伪代码[24]：

|  |
| --- |
| **算法2.4 Williams 方法** |
| **输入：**待分解的合数，光滑界  **输出：**的一个非平凡因子  1. 构建素数表  2. 随机选取使得，  3. 对于  ，表示第个素数  ，（也就说）  （这里）    如果，则进入下一个  如果，则输出并终止算法  如果，则返回步骤1并选取另一个 |

事实上，方法不是名副其实的“方法”. 由于我们不知道素因子具体的值，因此在选取时，有可能会使得Legendre符号，. 此时，这就使得Williams 方法变成了另一种形式的“方法”，即当时，有，计算得到非平凡因子.

Williams 方法在实践中比Pollard 方法要慢[24]. 其运算瓶颈在于计算Lucas序列的第项，如果只是使用递推公式，时间复杂度为，这在实践中是不可取的. 在下一节我们给出一个方法可以在的时间复杂度内计算.

### 快速计算Lucas序列的方法

为了快速计算，我们要利用下述结论[24]：

|  |  |
| --- | --- |
|  | （2-13） |

令

|  |  |
| --- | --- |
|  | （2-14） |

根据秦九韶算法，令，，于是.

由于，我们便可以通过递推计算[24]

|  |  |
| --- | --- |
|  | （2-15） |

得到.

下面，我们给出伪代码描述[24]：

|  |
| --- |
| **算法2.5 Lucas序列快速算法** |
| **输入**：初始值，项数  **输出**：序列第项  1. 将表示为  2. 如果，则返回2  3. 如果，则返回  4. 对于  如果，那么      否则，那么      5. 返回 |

### 适用范围与实际测试

我们给出若干个随机正奇数，并且这些数没有小于的素因子，用来测试Williams 方法在实际应用中的效果.

这里选择的参数如下：初始值，光滑界. 为雅可比符号，. 当时，方法事实上变为“方法”.

表2.4 Williams 方法实际效果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | 光滑 | 光滑 |
| 119896964274265910591 | 8011921 |  |  | **○** |
| 62451287400060090371459 | 2490617 |  | **○** |  |
| 3810890154018073410744853 | 1597817021 |  |  | **○** |
| 3898469093163961729117703 | 3220197143067809 |  |  | **○** |
| 1618097150196588399492881 | 12772868993 |  | **○** |  |
| 2980364646793620971041249 | 27377573 |  | **○** |  |
| 9543143972000854056103261 | 601708603 |  | **○** |  |
| 230596284855113908406849 | 34780345181 |  |  | **○** |
| 11087673216746767240321 | 31742712012937 |  | **○** |  |
| 3464162395610077675776881 | 1085980307 |  | **○** |  |
| 69732843355871203507 | 277103821 |  |  | **○** |
| 90821508504100871295431 | 1448111886301 |  |  | **○** |
| 244499761362073466267 | 3873768581 |  |  | **○** |
| 1965269570777866154148833 | 133074904208921 |  |  | **○** |
| 784596057928418607629839 | 10930127027 |  | **○** |  |
| 418844635229100014267 | 8178177193 |  | **○** |  |
| 284071363754547251 | 361742123 |  | **○** |  |
| 69732843355871203507 | 277103821 |  |  | **○** |
| 307729831547528105489 | 65364564383 |  | **○** |  |

注：如果分解出来的素因子p满足p+1的分解式全是小因子，在“p+1光滑”这一列标上记号“**○**”，如果素因子p满足p-1的分解式全是小因子，在“p-1光滑”这一列标上记号“**○**”.

通过上述的测试，我们发现该方法虽然名字叫做方法，但实质上应该叫做“方法”，这取决于初始值，由于素因子是未知的，雅可比符号的取值也是未知的，只能通过多次更换初始值来尝试.

与Pollard 方法相比，Williams 方法在计算Lucas序列上所花费的时间要比Pollard 方法的模幂运算多出1倍，在加上更换初始值（至少3次才能有的概率使得），消耗时间会更久[24].

因此，尽管Williams 方法的效率不如Pollard 方法，但Williams 方法的适用范围更加广泛，Pollard 方法可以分解的数，理论上Williams 方法也可以分解. 只要是光滑的，Williams 方法都有用武之地.

## 椭圆曲线整数分解算法（ECM方法）

椭圆曲线整数分解算法（ECM方法）是一个概率性整数分解算法，由H. W. Lenstra在1987年提出，可以快速地找到一个合数的“小”素因子[20]. ECM方法是典型的特殊整数分解方法，它的运行的时间只和素因子有关，和待分解的合数无关.

ECM方法的渐进时间复杂度为 ，其中是一个和具体算法实现细节有关的常数[12]. 从形式上看，它和后面将要介绍的二次筛法一样，是一个亚指数算法（只不过二次筛法的时间复杂度的参数不是素因子，而是整数），但在时，ECM方法可以更快地找到素因子，而且ECM方法实现起来更加简单，是一个相当实用的整数分解方法.

ECM方法本质上其实是Pollard 方法的一种“变体”，Pollard 方法从技术角度来看，利用的是乘法群的阶的光滑性，而Lenstra的ECM方法把方法中的乘法群换成椭圆曲线上的加法群，扩大了适用范围，提高了运行效率. ECM方法目前是除了数域筛法和二次筛法以外，使用较广泛的整数分解算法，其运行效率仅次于上述两种算法.

### 椭圆曲线上的点加群

在介绍ECM方法之前，首先要了解有限域上椭圆曲线的点加群. 我们使用Weierstra形式的椭圆曲线方程，并将无穷远点记为，于是有限域上椭圆曲线的点可以定义为[20]：

接下来，我们给出上的加法定义[20].

设任意非无穷远点且则：

（1）加法单位元就是无穷远点，即，

（2）和它的逆元素关于轴对称，即

（3）如图2.2，设过点，的直线交椭圆曲线于另外一点（如果，直线就是椭圆曲线在点处的切线），定义，



图2.2 椭圆曲线点加法示意图

|  |  |
| --- | --- |
|  | （2-16） |

其中

|  |  |
| --- | --- |
|  | （2-17） |

有了上的加法定义，我们继续给出“数乘”的规定[20]：

设为整数，当时，；

当时，；

当时，.

设为合数，在ECM方法中，我们并不是在（也就是）上做椭圆曲线的点加法和数乘，而是在上. 上椭圆曲线的点加法和数乘的定义和类似，只要把改为即可，但这样自然会遇到问题——的分母在模的意义下不一定可逆，下椭圆曲线的点加法可能失效，此时，就是的一个非平凡因子. ECM方法的核心就是利用上椭圆曲线的点加法和数乘的“失效”，来分解整数的.

### ECM方法的描述

设待分解的合数为，是的一个素因子，. 我们已经知道，上椭圆曲线的点加法和数乘可能会“失效”. 事实上，我们可以“人为”地制造数乘失效的情况，当，时，上椭圆曲线点的数乘就会失效！

因此，在ECM方法中，我们关注椭圆曲线上有限域上椭圆曲线的点的个数，也就是群的阶，记作（也有的文献记作）. 根据Hasse定理，我们有，也就是说和的大小相差不大[10]. 设

|  |  |
| --- | --- |
|  | （2-18） |

其中表示第个素数，为非负整数. 随机选取若干条椭圆曲线，并选择一个合适的上界，如果足够幸运，使得，对任意的，，那么

就有，计算就很可能失效，然后就可以通过欧几里得算法得到的一个非平凡因子. 下面给出ECM方法的伪代码[12]：

|  |
| --- |
| **算法2.6 ECM方法** |
| **输入**：待分解的奇合数（不是素数的方幂，），光滑界  **输出**：的一个非平凡因子  1. 随机选取，确定椭圆曲线，，点  2. 构建素数表  3. 对于每个      计算，如果数乘“失效”，找到计算过程中出现的不可逆元，返回，终止算法；否则置，如果，算法失败，返回步骤1取另一个椭圆曲线；反之，取下一个，继续循环. |

举例来说，，选取椭圆曲线，点，则：

但是，在计算时，的分母与不互素，计算“失效”，我们就可以得到的一个非平凡因子.

### 并行改进与实现

到目前为止，我们只是从理论层面描述了ECM方法，如果真的“一板一眼”地照着上面的伪代码在计算机上实现ECM的话，效率是相当低的. 通常确定了光滑界后，不可能幸运到一下子就找到了一条椭圆曲线，满足的最大素因子的方幂小于，因此可以采取一次同时选取多个参数，相当于一次选取多条椭圆曲线，通过下面的算法求多个数模的逆，或者. 这种并行算法的优势在于，它只需要使用一次扩展的欧几里得算法就能得到个数模的逆. 下面的算法[10]在一次计算多个椭圆曲线上的模逆运算中非常实用.

|  |
| --- |
| **算法2.7 并行模逆算法** |
| **输入：**正整数，个正整数并且不被整除  **输出：**模的逆，或者的一个非平凡因子  1.[初始化] 置，对，置  2.[扩展的欧几里得算法] 使用扩展的欧几里得算法，计算满足：  如果，转到步骤3；否则，如果，依次计算，，直到或，返回的非平凡因子，终止算法.  3.[求逆] 对于，计算，并置；最后，令，终止算法. |

值得注意的是，除了上述的并行计算来改进ECM方法外，还可以在算法失败后，添加第二阶段的基于“生日悖论”原理的算法来提高成功率，这里就不介绍了.

### 计算机实现与实际测试

我在自己的个人计算机上实现了一个并行版本的ECM方法，算法每次迭代都计算多条椭圆曲线上的点加法和数乘，但通过并行模逆算法，每次执行多条椭圆曲线的点加法都只需要一次扩展的欧几里得算法，大大提高了效率.

经过实际测试，这个程序可以轻松地找到十进制10~20位的素因子. ECM方法在具体实现时，对光滑界和椭圆曲线数量这两个参数比较敏感，参数和和素因子的规模有关，根据文献[21]的建议，设置的参数如下表：

表2.5 ECM方法参数设置

|  |  |  |
| --- | --- | --- |
| 素因子规模 | 光滑界 | 椭圆曲线数量 |
| 12 | 125 | 50 |
| 13 | 250 | 53 |
| 14 | 500 | 57 |
| 15 | 830 | 62 |
| 16 | 1500 | 68 |
| 17 | 2500 | 75 |
| 18 | 4200 | 85 |
| 19 | 6500 | 100 |
| 20 | 10000 | 120 |
| 21 | 15000 | 145 |
| 22 | 22000 | 175 |
| 23 | 32000 | 210 |
| 24 | 45000 | 250 |
| 25 | 65000 | 300 |
| 26 | 85000 | 400 |
| 27 | 115000 | 500 |
| 28 | 155000 | 650 |
| 29 | 205000 | 750 |
| 30 | 275000 | 950 |

下面是使用我自己编写的ECM分解出20位十进制素因子的部分测试数据.

表 2.6 ECM方法测试数据

|  |  |
| --- | --- |
| 符号约定：正整数N，素因子P，用时T（单位：秒） | |
| 第1组 | N = 6908590239850559602011876387531041010521 (40位)  P = 94577040848583019007 (20位)  T = 44.82 |
| 第2组 | N = 578562323970236801552978068523747766271 (39位)  P = 28831031353205586683 (20位)  T = 23.51 |
| 第3组 | N = 37581325427622622667438173346265231847360045506863 (50位)  P = 56139683050163382749 (20位)  T = 449.35 |
| 第4组 | N = 6199891197853790537561319511323553272909410690621 (49位)  P = 23197376900593042033 (20位)  T = 337.08 |
| 第5组 | N = 4883666612596499574181334751069294180831402451043 (49位)  P = 28589395631471598961 (20位)  T = 198.80 |

目前常见的ECM方法主要是linux平台下的gmp-ecm软件包，它是用C语言编写的，用gmp-ecm找到的最大素因子有83位（十进制下），在2013年被发现，它是的一个素因子. 这其实是一个非常惊人的结果，因为一般来说，ECM方法找到30位以上的素因子其实相当困难.

## 特殊整数分解算法的综合测试

我使用python实现了本章所介绍的Pollard 方法，ECM方法和方法，并对这些算法在我的个人笔记本电脑上进行了测试. 我的笔记本电脑的CPU为Intel Core i7-8550U，内存大小为16G. 由于在这一章中测试的数不算太大，因此电脑的配置不是决定性因素，关键在于代码自身的优化程度.

Pollard 方法和ECM方法都是寻找“小”的素因子的特殊分解方法，因此我们将方法和ECM方法作为一组测试对象. Wiliams 方法在选择特定参数的情况下，也可以分解Pollard 方法针对的素因子. 我们将方法和方法作为另一组测试对象.

首先比较方法和ECM方法的效率. 我们特意准备了7个用来测试的整数：

18269739683304744001

186381227182485763130761

645417815134960752258959257

26544338966722965136175979076061

250171128077165861042136131271837083

1009510520099489142611466356676300230681

73176966795773815265747287825533308144595403

它们分别拥有十进制10位、12位、14位、16位、18位、20位和22位的素因子. 由于我的代码并没有经过高度的优化，ECM方法只实现了第一阶段，效率不太稳定，最终记录下的时间数据是多次取平均值的结果. Pollard 方法实现的版本则是Brent改进版本. 测试结果如图2.3：

图2.3 ECM方法与ρ方法比较

当素因子的位数大于16时，Pollard 方法的用时就会变得很长，此时使用该方法就没有意义了. 从实验结果来看，使用Pollard 方法寻找十进制12位以下的素因子所用的时间很少. 如果实现得当，ECM方法寻找十进制10~20位之间的素因子是非常轻松的.

接下来比较Pollard 方法和Williams 方法. 通过2.4节的讨论，我们知道Pollard 方法能够分解的数，使用Williams 方法也可以分解（只要参数选择正确）. 下面我们测试一下，分解同样的整数，Pollard 方法和Williams 方法哪一个更快一些. 我们选取六个素数：

P1 = 784440582794765917351

P2 = 865130977464802032781

P3 = 533699432609757591013

P4 = 833028960742355107769

P5 = 331630727321919020131

P6 = 350202979272346202333

用这些素数分别乘以十进制37位的素数1803164397518375171384717924112312493得到六个合数，然后测试两种方法找到上述素因子所需要的时间. 测试结果如下图：

图2.4 p+1方法与p-1方法比较

从实际测试结果看，Pollard 方法的运算速度是Williams 方法的两倍！根本原因在于，方法计算Lucas序列需要的时间为，Pollard 方法中的快速模幂运算需要的时间为.

这里还是在参数选择最优的前期下的测试结果. 实际使用过程中Williams 方法的失败率也明显高于Pollard 方法. 因此，方法的实用性并不高.

# 一般整数分解算法

“一般整数分解算法”指的是整数分解算法的渐进时间复杂度只和待分解合数的大小有关. 所谓的“一般”，表现在无论合数拥有怎样“特殊”的结构和“小”的素因子，算法的运行效率都不会产生显著的变化. 实际应用中，只有在特殊整数分解方法很难找到素因子的情况下，才使用一般整数分解方法. 常见的一般整数分解方法有Lehman方法、SQUFOF方法、连分数因子分解方法、二次筛法和数域筛法等等[12]. 在这一章中，我们介绍两个非常著名的整数分解算法——二次筛法与数域筛法.

## 二次筛法（Quadratic Sieve）

二次筛法（Quadratic Sieve，以下简称QS方法）是目前为止，较为常见的一个整数分解算法，有许多数学软件和整数分解工具都实现QS方法的某些改进版本的变体，来提供大整数分解服务. QS方法的渐进时间复杂度为，其中是一个和具体算法实现有关的常数[12]. 一般来说. 这是一个亚指数算法，其参数是待分解的合数. 不同于之前介绍的特殊整数分解方法，QS方法的运行时间只和的规模有关，哪怕的素因子全是“小”数，QS方法也不会因此而更快地给出结果. 因此，QS方法最合适的使用场景就是分解RSA数——由两个规模相近的奇素数相乘得到的奇合数[29].

在实际应用中，各个数学软件和整数分解工具实现的都是QS方法的改进版本MPQS（Multiple Polynomial Quadratic Sieve）或者SIQS（Self-Initializing Quadratic Sieve）. 对于一个随机给出的合数，直接套用QS方法是不明智的. 我们需要先尽可能地找到“小”的素因子，然后再使用QS方法分解剩余的部分，这样才能高效地利用QS方法.

尽管QS方法在渐进时间复杂度上和分解能力上不如数域筛法，但是QS方法在整数分解领域依旧有着不可替代的地位，分解十进制100位以下的RSA数，数域筛法的实际运行速度不如QS方法，而且QS方法实现起来更加简单. 因此，在绝大多数应用场景下，使用QS方法的分解能力绰绰有余了.

### Fermat方法和Dixon方法

从算法的技术细节往前追溯，QS方法可以看作是Dixon方法的改进，而这两个算法和更古老的Fermat方法的思想一脉相承，甚至是技术上更加复杂的数域筛法，其最终的目标也和Fermat方法一样——对于给定的奇合数，找到两个正整数使得. 从同余的观点来看，正整数满足：

|  |  |
| --- | --- |
|  | （3-1） |

如果能够找到这样的两个正整数，那么和就是的两个非平凡因子.

Fermat方法粗看上去十分“野蛮”，依次令，直到表达式是一个完全平方数，此时. 当然，我们也可以依次令，直到是一个完全平方数，此时

下面我们分析Fermat方法的时间复杂度. 设奇合数，为两个不同的奇素数，不妨设，则

|  |  |
| --- | --- |
|  | （3-2） |

如果我们穷举，大约需要迭代次后，才能找到非平凡因子，如果我们穷举，大约需要迭代次后，才能找到非平凡因子. 无论怎样，Fermat方法只有在时才能奏效.

事实上，随机选取，同余式的右边往往不是一个完全平凡数，但是如果选取多个数得到多个同余式

|  |  |
| --- | --- |
|  | （3-3） |

将这些同余式组合起来，就能得到平方数：

|  |  |
| --- | --- |
|  | （3-4） |

此时只要令，就找到了我们想要的平方差关系.

Dxion方法[16]（还有更早的连分数因子分解算法）在此基础上引入了因子基的概念，因子基，为所有素数构成的集合. 我们设因子基的大小为，Dixon方法依次选取大于的整数，计算. 如果的所有素因子都在因子基中，我们就保留同余式

|  |  |
| --- | --- |
|  | （3-5） |

当我们收集了足够多的这样的同余式以后，我们就可以组合出平方关系了. 很容易发现：

当且仅当

也就是

于是，我们在给定同余式中寻找平方关系就转化为了上的线性代数问题. 我们在存储每一个同余式时，只需要记录指数模2向量

|  |  |
| --- | --- |
|  | （3-6） |

只要我们搜集到的指数向量的数量，下面的方程

|  |  |
| --- | --- |
|  | （3-7） |

一定有非零解（或者说线性相关）其中. 接下来，对于每一组非零解中使得的下标，就有

|  |  |
| --- | --- |
|  | （3-8） |

我们可以举个简单的例子来帮助理解，设待分解的数，因子基，搜集光滑数如下：

表3.1 Dxion方法演示

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | 模2指数向量 |
| 1 | 296 |  |  |
| 2 | 301 |  |  |
| 3 | 311 |  |  |
| 4 | 314 |  |  |
| 5 | 379 |  |  |

选取的同余式，组合出如下的平方关系：

即

于是，我们就可以得到两个非平凡因子：

和

总结来说，Dxion方法一共分为两个阶段：一、搜集同余关系式；二、求解上的齐次线性方程组. Dxion方法其实与二次筛法已经很接近了，它们的唯一区别只在第一阶段，Dxion方法的瓶颈在于需要用穷举和试除法来寻找足够多的能够在因子基中完全分解的，对于较大的，这种方法是不可取的. 二次筛法使用一元二次多项式通过筛法在给定区间内找到可以在因子基中完全分解的数，不需要对每一个数都使用试除法，大大提高了效率.

### 光滑数与筛法

在正式介绍二次筛法之前，首先要了解一个重要的概念——光滑数[10]（smooth number）. 我们在介绍方法和ECM方法时已经接触过这个概念，当一个正整数全部都由小的素因子构成时，就称它是“光滑数”. 现在，我们可以定量地给出一个整数的光滑性. 设是一个正整数，如果正整数的最大素因子，那么就称正整数是-光滑（-smooth）的[10]. 如果正整数的任意素数幂因子，则称正整数是-强光滑（-powersmooth，这里的power其实是指“幂”的意思，但我认为意译为-强光滑更好听）的[10]. 我们将整数称为正整数的光滑界.

在下面的讨论中，设因子基的上确界为. 从光滑数的角度看，Dxion方法就是寻找-光滑的. 二次筛法则是寻找足够多的使得多项式是-光滑的.

需要注意的是，二次筛法中的因子基中的素数应该满足Legendre符号，这样二次同余式才有解. 换句话说，的素数不可能是的素因子.

二次筛法中的“二次”指的就是二次多项式，“筛法”指的是一种类似于Eratosthenes筛法的技术来快速筛选光滑数的方法. 我们可以将Eratosthenes筛法看作是在线性序列上用素数对每一个合数进行标记，具体来说，就是对于每一个素数，首先对下标为的项进行标记，然后依次标记下标为的项. 在线性序列中的某个合数如果有个素因子，它就会被标记次，如果把Eratosthenes筛法中每次的标记操作变为除以当前的素数直到无法整除，然后用这个值替换序列中原来的. 这样，当时，说明是-光滑的数.

现在，只需要简单改造Eratosthenes筛法就能得到一个快速寻找区间上-光滑数的算法[29]. 尽管这方法需要预计算出小于的素数表，其实它和Eratosthenes筛法非常很相似.

|  |
| --- |
| **算法3.1 区间上-光滑数筛选算法** |
| **输入**：正整数，光滑界  **输出**：上所有的-光滑数  1. 构建素数表，  2. 初始化数组，满足，置变量  3. 如果，则置；否则，变量取下一个素数. 如果，跳转到步骤6；否则就有，那么置，跳转到步骤4.  4. 置，直到不整除  5. 置，如果，跳转到步骤4；否则，跳转到步骤3  6. 对于数组，从2到依次取值，如果，则是-光滑的；否则就不是光滑数. |

在一个区间上寻找光滑数还不是我们的最终目的. 确切来说，我们是在一个二次序列上寻找光滑数，因此仍然需要改造上面的算法. 在区间上寻找光滑数就是在线性序列上寻找光滑数. 筛法之所以能成立的原因，在于线性序列满足：如果对于某个素数和某个下标使得，则一定有，其中为正整数. 这样所有是素数的倍数的都能被找到.

显然，二次序列也满足这样的性质：

|  |  |
| --- | --- |
|  | （3-9） |
|  |
|  |
|  |

只要找到上的解，也就是求解同余方程，然后就可以通过下标每次增加来找到上使得为素数的倍数的所有下标. 一般地，如果要筛选素数幂因子，由于，我们可以通过求解，下标每次递增寻找是倍数的. 由于对于较大的素数，这种情况出现的概率很小，一般在可以限制的大小的前提下，只对小素数进行这样的筛法.

下面给出二次筛法中的“筛法”[18]伪代码：

|  |
| --- |
| **算法3.2 多项式上-光滑数筛选算法** |
| **输入**：正整数，光滑界，区间  **输出**：中所有的-光滑数  1. 构建素数表，  2. 初始化数组，满足，置变量  3. 如果，则置；否则，变量取下一个素数. 如果，跳转到步骤8；否则就有，跳转到步骤4.  4. 求解出二次同余方程全部的解，设满足  5. 置  6. 置，直到不整除  7. 置，如果，跳转到步骤5；否则，跳转到步骤3  8. 对于数组，从到依次取值，如果，则是-光滑的；否则就不是光滑数. |

上面的算法其实也只是理论层面的，真正在计算机实现时，我们可以在步骤2初始化数组，这里不需要精确的对数运算，只需要取整即可，这样步骤6的试除法就变为，在最后的步骤8中，如果在一定的误差内和“0”非常接近，就用试除法验证是否-光滑[19]. 尽管这样做可能遗漏一小部分光滑数，但由于只对少数做试除法，能够显著提升二次筛法的效率.

### 上的齐次线性方程组

二次筛法的第二步需要找到上的齐次线性方程组的非零解，很容易想到的方法就是著名的“高斯消元法”. 高斯消元法适用于任何数域上的线性方程组，但它的时间复杂度高达，其中为矩阵行数和列数的最大值. 如果分解的数不是很大，二次筛法搜集的光滑数也不会很多，求解线性方程组时的矩阵规模也比较小，使用高斯消元法没有问题. 但如果分解的数相当大，最后矩阵的规模就有上万行和上万列，而且大部分元素为“0”，这时使用高斯消元法就不合适了.

对于上的大型稀疏矩阵的处理，已经有一些相当成熟的处理算法了. 可选的比较好的算法有Block Lanczos算法，具体描述这个算法将会占据很大的篇幅，要详细了解可以阅读文献[7][27][28].

### 二次筛法的一些改进版本

二次筛法最大的瓶颈就是寻找光滑数的速度. 原始的二次筛法QS使用固定的多项式，通过更换不同的区间来寻找光滑数. 但这样寻找光滑数的效率太低下了. QS的改进版本MPQS和SIQS都使用了这样的改进思路[10]：

在固定的筛选区间上，通过不同的二次多项式来寻找光滑数，这里常数满足. 这样就有

|  |  |
| --- | --- |
|  | （3-10） |
|  |
|  |

于是，就有同余关系

|  |  |
| --- | --- |
|  | （3-11） |

MPQS和SIQS的区别在于多项式参数的选择上. MPQS的参数，其中是在因子基中最接近的素数. SIQS的参数则是由因子基中多个素数相乘得到，并保持. 接下来通过解同余式

|  |  |
| --- | --- |
|  | （3-12） |

得到参数（的值不止一个），接着通过在多项式上使用3.1.2中描述的筛法寻找光滑数. 当无法找到其他光滑数时，更换参数，区间不变，继续使用筛法. 上的大型稀疏矩阵的处理和原始的QS方法一样. MPQS和SIQS在渐进时间复杂度上和原始的QS方法一样，但实际应用中，大大加快了分解过程. 文献[18]还指出，SIQS的效率至少是MPQS的两倍. 常见的整数分解工具如msieve和yafu实现的二次筛法实际都是SIQS.

MPQS和SIQS还有许多实现和优化的细节，限于篇幅，没有办法在此处讨论，有兴趣的读者请参考论文[18].

## 数域筛法（Number Field Sieve）简介

数域筛法（简称NFS）是目前渐进时间复杂度意义下最快的整数分解算法，其时间复杂度为，常数与具体实现有关[12]. 对于特殊数域筛法SNFS（Special Number Field Sieve），常数. 对于一般数域筛法GNFS（General Number Field Sieve），常数稍微大一些，是.顾名思义，SNFS专门用来分解一类特殊的数，比如形如的梅森数、形如的费马数等等，而GNFS则可以分解任何整数[6].

NFS涉及到很深的数学原理，有代数数论的基础知识才能真正理解NFS的原理，这里只大致地描述NFS的整体流程. NFS大体上分为四个步骤[6]：

（1） 在环中选择多项式，使得整数满足，即

（2） 寻找的复数根，使得，定义环

（3） 定义环同态映射，并且

（4） 找到若干组整数对，使得

|  |  |  |
| --- | --- | --- |
|  | 且 |  |

设，于是

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  | （3-13） |

最后计算和得到的非平凡因子.

NFS和QS相比，将代数结构从单纯的扩展到了上，因此可以在更高次数的多项式上寻找光滑数. 要了解更多关于NFS的历史和细节，请阅读文献[22]. 这里不继续介绍了.

## 二次筛法与ECM方法的测试与比较

我们用python实现了二次筛法的改进版本——SIQS. 这里的实现比较“粗糙”，没有进行优化，因此不能充分利用现代计算机的计算能力. 经过反复测试，我们实现的SIQS最大分解能力在十进制50~60位之间.

下表比较了我们实现的ECM方法和SIQS在分解十进制20位、30位、40位和50位数所使用的时间. 在寻找小因子时，ECM方法依然有相当大的优势. SIQS方法适合分解没有“小”的素因子的大整数.

表 3.2 ECM方法与SIQS方法比较

|  |  |  |
| --- | --- | --- |
| 整数十进制位数 | ECM用时（单位：秒） | SIQS用时（单位：秒） |
| 20 | 0.11 | 0.53 |
| 30 | 4.11 | 1.41 |
| 40 | 32.68 | 11.98 |
| 50 | 583.64 | 634.04 |
| 60 | -- | 2417.42 |

二次筛法对因子基的大小很敏感，因子基较小，需要找的光滑数就少，但是寻找光滑数的速度很慢；因子基较大，寻找光滑数的速度很快，但要找的数量很多，构建的矩阵会很大. 尽管因子基的大小有理论的上界，但还是需要通过实验来确定最佳的因子基大小.

二次筛法也很容易估计实际运行时间，假如因子基的大小为3000，如果每秒可以找到1个光滑关系，大概需要50分钟才能分解这个数. 因此，二次筛法实现的快慢也常用每秒找到的光滑数数量来衡量.

目前最快的二次筛法实现版本yafu使用C语言编写，在实现SIQS本身的同时还运用了Two Large Primes技巧来处理不能在因子基中完全分解的整数，在编程技巧上使用了多线程编程方法充分利用现代计算机的多核优势，其分解能力可以在1小时左右的时间分解十进制100位的整数，2小时左右的时间分解十进制103位的整数.

# 参考文献

1. 全国高校密码数学挑战赛命题委员会.首届（2016）全国高校密码数学挑战赛——赛题三 [EB/OL].[2016-03-01]. http://www.cmsecc.com/xiazai/.
2. 柯召,孙琦.数论讲义[M].北京:高等教育出版社,2001.
3. 颜松远.整数分解:中小学数学问题, 大数学家难题[M].北京:科学出版社, 2009.
4. 姜建国,臧明相.数论算法[M].西安电子科技大学出版社,2014.
5. 裴定一,祝跃飞.算法数论[M].北京:科学出版社,2002.
6. 颜松远.整数分解新方向[J].计算机工程与科学,2013,35(01):1-14.
7. 应文娟. Lanczos算法在数域筛法中的应用与并行性研究[D].北京工业大学,2014.
8. Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to algorithms[M]. MIT press, 2009.
9. Knuth D E. Art of computer programming, volume 2: Seminumerical algorithms[M]. Addison-Wesley Professional, 2014.
10. Cohen H . A course in computational algebraic number theory[M]. World Pub. Co, 1997.
11. Rivest R L, Kaliski B. RSA Problem[J]. Encyclopedia of Cryptography & Security, 2005, 25:1065-1069.
12. Yan S Y . Primality testing and integer factorization in public-key cryptography /[M]. Springer US, 2009.
13. Pollard J M. A Monte Carlo method for factorization[J]. BIT Numerical Mathematics, 1975, 15(3): 331-334.
14. Brent R P. An improved Monte Carlo factorization algorithm[J]. BIT Numerical Mathematics, 1980, 20(2): 176-184.
15. Pollard J M. Theorems on factorization and primality testing[C]//Mathematical Proceedings of the Cambridge Philosophical Society. Cambridge University Press, 1974, 76(3): 521-528.
16. Dixon J D. Asymptotically fast factorization of integers[J]. Mathematics of computation, 1981, 36(153): 255-260.
17. Garrett S L. On the quadratic sieve[D]. The University of North Carolina at Greensboro, 2008.
18. Contini S P. Factoring integers with the self-initializing quadratic sieve[D]. The University of Georgia, 1997.
19. Pomerance C . The Quadratic Sieve Factoring Algorithm[C]// Workshop on the Theory and Application of of Cryptographic Techniques. Springer, Berlin, Heidelberg, 1984.
20. Lenstra H W, Jr. Factoring Integers with Elliptic Curves[J]. Annals of Mathematics (Second Series), 1987, 126(3):649-673.
21. Bosma W, Lenstra A K. An implementation of the elliptic curve integer factorization method[M]//Computational Algebra and Number Theory. Springer, Dordrecht, 1995: 119-136.
22. Lenstra A K, Hendrik Jr W. The development of the number field sieve[M]. Springer Science & Business Media, 1993.
23. Lehmer D H. An extended theory of Lucas' functions[J]. Annals of Mathematics, 1930: 419-448.
24. Williams H C. A 𝑝+ 1 method of factoring[J]. Mathematics of computation, 1982, 39(159): 225-234.
25. Rivest R L, Silverman R D. Are Strong Primes Needed for RSA?[C]//The 1997 RSA Laboratories Seminar Series, Seminars Proceedings. 1997.
26. Boneh D. Twenty years of attacks on the RSA cryptosystem[J]. Notices of the AMS, 1999, 46(2): 203-213.
27. Montgomery P L. A block Lanczos algorithm for finding dependencies over GF(2)[C]//International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 1995: 106-120.
28. Peterson M. Parallel block lanczos for solving large binary systems[D]. Texas Tech University, 2006.
29. Buhler J P , Stevenhagen P . Algorithmic number theory. Lattices, number fields, curves and cryptography. Reprint of the 2008 hardback ed.[M]// Algorithmic Number Theory. Lattices, Number Fields, Curves and Cryptography. Cambridge University Press, 2008.
30. Boneh D, Venkatesan R. Breaking RSA may not be equivalent to factoring[C]//International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 1998: 59-71.