计算机在数论中的应用

英国约克大学数学系　颜松远

摘要　计算机与数论有着十分密切的联系。一方面，计算机在数论中有着广泛的应用；另一方面，数论也在计算机科学中有着深入的应用；在本文中，我们力图提纲挈领地、简明扼要地介绍计算机在数论中的应用。具体而言，我们要介绍计算机在素数测试、整数分解、孙子定理以及哥德巴赫猜想等数论领域中的应用。

关键词　计算机　数论　应用　素数　整数　孙子定理　哥德巴赫猜想

一、引言　数论是数学中最古老、最纯粹、最重要的一个分支，它的初等部分便是大家所熟知的算术。在今天的社会里，即便是一个文盲，也略懂一些算术知识。相对于数学而言，计算机科学则是一个十分年轻的学科，它从出现到至今也不过短短四、五十年的时间。不过，尽管它们的"年龄"相差悬殊，但它们却有着十分密切的联系，并且这种联系完全是双向的。一方面，计算机在数论中有着广泛的应用，如在素数测试、整数分解、孙子定理以及哥德巴赫猜想等许多问题的研究中，都直接或间接地要依赖于计算机的帮助。另一方面，数论也在计算机科学中有着深入的应用，如在计算机算术、计算机设计、可计算性理论以及计算复杂性理论等计算机科学领域中，都需要大量用到数论的知识。更为重要的是，计算机科学与数论互相渗透、交叉影响，并逐步地改变着各自的面貌。比如目前蓬勃兴起的学科—计算数论，就是计算机科学 与数论相互渗透、互相影响而产生的一门新兴的边缘学科。这正是：

"算机数论两兄弟，　形影相吊不分离；

算中有论根基深，　论中的有算虎添翼"。

当然，在这篇短文中，我们不可能面面俱到地介绍计算机在数论中的所有应用，也不可能讨论计算数论中的所有内容。事实上，我们只是选择几个具有重要意义的问题，如素数测试，整数分解、孙子定理以及哥德巴赫猜想等，介绍计算机在数论中的应用。

二、素数测试　素数测试是当代数论中、尤其是当代计算数论中的一个十分重要的研究领域。在诸如计算机代数、数学密码、保密计算以及网络安全等应用领域中，素数测试几乎是不可缺少的一个重要工具。同时，它还是连结纯粹数学与应用学科的一座重要桥梁。因此，素数测试无论是在纯粹学科中，还是在应用学科中，都具有十分重要的意义。但是，素数测试又是一个很困难的问题，具有计算上的难解性之特点，因此千百年以来，许多一流数学家以及近代的许多优秀计算机科学家都曾涉足这个领域，这其中包括十七世纪法国某著名数学家麦什涅和费尔马、十八世纪著名瑞士数学家欧拉，以及享有数学王子之 称的十九世纪德国数学家高斯。可是，尽管有这么多的名人来从事这项工作，但素数测试问题始终没有得到彻底解决。

所谓素数测试(或素性测试)，就是测定一个整数的素合性，也即给定一个大于1的整数N，要求测定出该N到底是素数还是合数。根据数论的基本知识，我们知道，对于任意一个大于1的整数N，它要么是素数，要么是合数，二者必居其一且仅居其一。用比较形式一点的话讲，就是：

输入：给定一个大于1的整数N；

输出：如果N为素数，输出Yes；否则输出No.

这个问题从表面上看，似乎极简单，但它却是一道著名数学难题。

最早期最简单的素数测试算法大概要算"试除法"。所谓试除法，就是用1与N之间的整数去遍除N。即(我们将余数为零记之为0，将余数不为零记之为1)：

$$\frac{n}{2}\Rightarrow 1/0$$

$$\frac{n}{3}\Rightarrow 1/0$$

$$\cdots$$

$$\frac{n}{n-1}\Rightarrow 1/0$$

只要其中有一个除式的余数等于零，N就是合数。如果所有的除式之余数都不为零，那么N便是素数。当然，在实际上，我们完全不必要测试到n－1。下面我们引出一条重要的数论定理，据此可以设计出较好的试除算法来。

定理1：设$n>1$。如果n没有小于或等于根号n的素因子，那么n就是素数。

根据这个定理，如果我们要测试整数n是不是素数，那么我们不必测试在1与根号n之前的所有

整数，而仅需测试在1与根号n之间的素数便可。例如，测试任何一个小于1000000的整数，我们至多仅需进行168次试除，因为根号1000000等于1000，而在1000之内只有168个素数。由此可知，这个方法还是比较快的。从数学上讲，这个方法不仅是可靠的，而且也是完备的(也即它说n是素数，n就一定是素数；它说n不是素数，n就一定不是素数)。值得一提的是，这个定理不仅可以用来测试素数，而且可以用来指导生成在n之前的所有素数。著名的素数生成算法—Erathosthenes 筛法，实际上与这个定理的思想是一致的。

下面这个适合于在计算机上实现的算法，就是根据上述定理而设计的：

算法1(试除测试算法)：

```
begin
    while d_k ≤ √n
      begin
        if d_k | n
Output n is composite, goto Exit
else k←k+1
end
output n is prime
Exit:
end
```

算法中的d[k]为已知的从2到根号n的素数集合。我们用此集中的数去遍除n.

作为一个例子，我们来看看如何用这种试除法来确定一个大于1的整数n是不是素数。假定欲被测定的数是198371，我们用从1至445(即根号198371之近似值)之间的素数去除198371。首先，2不能整除198371，3、5、7、11等素数都不能，因此我们必须继续试，一直试到443为止，因为443是最接近445的素数。可是当我们试到163这个素数时，它可整除198371。这也就是说，198371除以163之余数为零。因此我们断定198371是一个合数而不是一个素数。相反，如果我们欲测定整数3989，那么，我们必须用在1与63之间的素数去试除3989。由于从2到61为止的所有素数都不能整除3989，因此3989是一个素数。

定理2(素数定理)：设n>1。如果n是一个很大的整数，则其小于或等于n的素数的个数$\pi(n)$大约等于$\frac{n}{\ln n}$。更精确的讲，我们有下列的式子：

$$\lim_{n\to\infty}\frac{\frac{\pi(n)}{n}}{\frac{1}{\ln n}}=1$$

定理2所给定的素数个数的估计既非常简单又比较精确。下面我们给出十个根据定理2得出的估计值，据此读者可以作一番比较：

n	$\frac{n}{\ln n}$	$\pi(n)$
10^1	4.3…	4
10_2	21.7…	25
10^3	144.7…	168
10^4	1085.7…	1239
10^5	8685.8…	9592
10^6	72382.4…	78498
10^7	620420.7…	644579
10^8	5428681.0…	5761455
10^9	48254942.4…	50847478
10^{10}	434294481.9…	455052512
…	…	…

根据定理1，为了测定n的素合性，我们需要用小于等于根号n的所有素数去遍除n，再根据定理2，我们知道小于等于根号n的素数大约有如下这么多个：$\frac{\sqrt{n}}{\ln\sqrt{n}}=\frac{2\sqrt{n}}{\ln n}$

假定我们在最现代化的计算机上计算，那么这种计算机执行一个试除至多仅需$\frac{\ln n}{10^6}$秒钟。因此，我们最多仅需

$\frac{2\sqrt{n}}{\ln n}\times\frac{\ln n}{10^6}=\frac{2\sqrt{n}}{10^6}$秒钟去测定n的素合性。

直至本世纪七十年代，试除法一直是最基本最常用的一种素数测定法。就是在今天，试除法也还有一定市场。不过当被测之整数很大时，这种方法是无效的。例如，当被测之整数n是一个30位的整数时，用这种方法大约要63年才能知道n是不是素数。

下面我们介绍一种稍好点的算法—费尔马法。

所谓费尔马法，就是基于费尔马小定理的一种测试素数的方法(费尔马小定理是法国数学家费尔马在1640年发明的)。

定理3(费尔马小定理)：如果P为素数，且gcd(a,p)=1，则$a^{p-1}\equiv 1(mod\ p)$. 根据这个定理，我们可以很容易地得出如下推论：

推论1：如果gcd(a,N)=1，且$a^{N-1}\not\equiv 1(mod\ N)$. 则$N$为合数。

再根据这个推论，我们可以很容易地确定出N是不是合数。可能读者会问，当N很大时，计算$a^{N-1}(modN)$之复杂性是不是也会很大呢？有幸的是，在计算机上计算$a^{N-1}(modN)$之复杂性仅为$O((log_2 n)$，由于这是一个对数级的复杂性，因此这当然没有任何计算上的困难性。注意，前面刚介绍的试除法之计算复杂性为$O(2^{(log_2 n)/2})$，由于这是一个指数复杂性的算法，因此当N很大时，它无法在计算机上实现。

作为费尔马法的一个例子，我们来看看它如何确定 1736 这个整数是不是合数。首先我们选定 $a=2$。由于：

$$2^{1763-1}\equiv 742 (mod\ 1763)$$
$$\neq 1 (mod\ 1763)$$

因此 1763 是一个合数而不是一个素数。如果我们用试除法来确定 1763 的素合性，那么必须用 1763 去遍除 2,3,5,7,…，直至 41，才能知道 1763 是合数而不是素数，因为 41 是第一个可以整除 1763 的素因子。

从数学上讲，费尔马法是可靠的，但不是完备的。也即它说整数 n 是合数，那么 n 肯定是合数；但若它说 n 是素数（即不是合数），那么我们就要打个问号了，因为有时它会指鹿为马，不是素数它也说是素数，关于这一点，很多人不理解，他们认为反正对于一个大于 1 的整数 n，它不是素数就是合数，不是合数就是素数，就象一个数的奇偶性一样，它不是奇数就是偶数，不是偶数就是奇数。不过我们要强调的是，数的素合性可比奇偶性复杂多了。我们还是通过例子来说明这个问题吧，比如我们要测定 1387。由于：

$2^{1387-1}\neq 1 (mod\ 1387)$，因此根据费尔马小定理，这不是一个合数而是一个素数。可是事实上 1387 根据不是素数而是合数，如果用试除法我们可以很容易得到 1387 的第一个素因子 19。因此我们称 1387 是伪素数（或基于 2 的伪素数），也即它看起来象素数可其实它并不是素数。由此我们可以得出一个基本结论，即如用费尔马法测定 n 是合数，那么 n 一定是合数；如用费尔马法测定 n 为素数，那么 n 可能真是素数，也可能不是素数。这完全是一个概率，也许正因为费尔马法具有一定的概率性，因此，它也可被称之为概率法，不过尽管此法有一定概率性，但在未能找到有效的素数测试算法之前，它仍不失为是一个好方法。并且，相对来说，出现这种概率性的错误机会是很小的。即便有错，我们还有一定的手段与办法来补救它。事实上，很多近代的一些基于概率的素数测试方法如强伪素数测试和随机测试等方法，都是在此基础上发展壮大起来的，并且这些方法的基本出发点都是为了增强其可靠性与完备性。若用现代概率测试法来测定 n，如果 n 是一个具有 100 个十进制位的整数，那么它仅需 33 秒钟，200 位的整数需 8 分钟，就是 1000 位的整数，也只不过一个星期。由此可知，尽管概率法不太准确，但其速度倒确实是很快的，是试除法所望尘莫及的！当然，概率法并非最好的素数测试方法。事实上，我们今天已经有了很多基于深刻数学理论（如代数几何）的方法。

前面介绍的试除法及费尔马法是属于通用型方法，对一些具有特殊形式的整数，如麦什涅数，我们当然不必用通用型方法测试，而应采用特殊方法测试。下面我们简单介绍适应于麦什涅素数测试的 Lucas—Lehmer 方法：

所谓麦什涅数，是指形为如下之数。即：$M_p=2^p-1$，其中 p 为素数，如果形为 $M_p=2^p-1$ 的麦什涅数是一个素数，那么，我们就称其为麦什涅素数。

Lucas—Lehmer 方法主要基于如下定理：

定理 4：设 q 为奇素数，并用如下规则定义序列 $<Ln>$：

$L_0=4$，$L_{n+1}=(L_n^2-2) mod (2^q-1)$。

则 2^q-1 为素数且仅当 $L_{q-2}=0$。

根据这个定理，我们可以很容易地设计出测试麦什涅素数的算法，即：

```
L=4
For i =3 to p do
    L=(L²-2)mod(2^p-1)
    If L=0(mod2^p-1)
    then 2^p-1 is prime
    else 2^p-1 is composite
```

这是一个很有效的测试麦什涅素数的算法。在具体计算时，为了加快其速度，一般都采用 Schonhage—Strassen 之快速富里埃速乘法来计算 L^2（L^2 是最影响速度的一个计算项目）。

到目前为止，人们已经发现了前 32 个麦什涅素数。即（我们仅列出 2^p-1 中 p 之值）：

$p=2,3,,5,7,13,17,19,31,67,89,107,127,521,607,1279,2203,2281,3217,4253,4423,9689,9941,11213,19937,21701,23209,44497,86243,110503,132049,216091,756839.$

其中 $p=756839$ 这个麦什涅素数是刚于 1992 年 3 月间在英国 *Cray* 研究中心的 *Cray*－2 巨型计算机上发现的，这是当今世界上最大的素数，它具有 227832 个十进制数位。上面这个表列中所列出的比较大的几个素数，基本上都是用 *Lucas—Lehmer* 方法发现的。不过有一点需要强调的是，*Lucas—Lehmer* 法只适合于麦什涅素数的测试，对于具有一般形式的整数的素性测试，它是无能为力的。因此，对于一般形式的整数，就目前来讲，主要还得靠概率法或其它更现代化的方法。

解决素数测试问题的关键是研制出具有多项式复杂性的测试算法。可是遗憾的是，到目前为止，人们还没有找到这样的一种算法。当然，也无人能否定说没有这种算法存在的可能性。不过理论性的结果已经表明：在广义黎曼假设的情况下，素数测试问题属于 p（即多项式）；如果消掉这个假设，那么它属于

$co-RP$(即随机多项式的补集)和 NP(即非确定型多项式);另外,它还可以用确定型算法在 $O(n^{O(loglogn)})$计算步骤内得到解决。最新的结果则更表明素数测试问题在 RP 与 $ZPP(ZPP=R\cap co-R)$之间。这也就是说,素数测试问题可以在概率多项式时间内得到解决。总而言之,虽然我们还没有找到确定型多项式复杂性的素数测试算法,但我们找到了测试素数的随机(即概率)多项式算法,尽管随机多项式算法(RP)没有确定型多项式算法(P)好,但它却比非确定型多项式算法(NP)及指数复杂性算法要好得多。由此可知,尽管素数测试问题还没有得到彻底解决,但也是比较接近彻底解决了。

由于素数有无穷多个(根据欧几里德定理),因此寻找特大的素数、尤其是寻找具有一般形式的特大素数,是当前素数测试的一个重要研究方向,因为具有一般形式的特大素数在密码设计等领域中有重大的应用价值。

三、整数分解 整数分解是当前计算数论中最重要的研究课题之一。就其重要性、应用性以及影响力而言,数论中所有与计算机相关的问题,没有一个可以与整数分解相提并论。

大家知道,数论中的许多问题,都具有一个显著的特点,即看起来似乎很容易但解起来却非常的困难。整数分解就是属于这类问题中的一个。到目前为止,人们还没有找到一种分解整数的有效的算法,但又没有人能否定这种法算存在的可能性。也许正因如此,从古至今,这个问题才吸引了许许多多的数学家以及计算机科学家,其中包括历史上的名家费尔马、欧拉以及高斯。与我们前面刚介绍的素数测试相比,整数分解可就要更困难得多、更重要得多了。

早在两千年前的欧几里德时代,人们就知道任何一个大于1的整数,都可以表示成素数的乘积形式。当然这个问题的最终的精确描述与理论证明,则是1801年在数学王子高斯手上完成的。这就是我们今天人所皆知的"算术基本定理"。

定理5(算术基本定理):任何一个大于1的整数,都可以唯一地表示成如下之素数的乘积形式,即:$n=p_1^{e_1}p_2^{e_2}\cdots p_r^{e_r}$,其中,$p_1<p_2\cdots<p_r$ 为素数,$e_1<e_2<\cdots e_r$ 为正整数。

例如,123456789是一个大于1的整数,因此,它可以唯一地表示成如下的素数的乘积形式,即:

$$123456789=3^2\cdot 3607^1\cdot 3803^1$$

其中3,3607和3803均为素数。

在此值得特别一提的是,当大于1的这个整数是素数时,那么我们认为它已被分解成了素数的乘积形式,此乘积形式只有一个因子,就是它自己。例如,素数1234567之素数乘积形式可以简单地写成123457=123457。当整数 n 不是很大时,比如说只是一个具有五、六位或七、八位十进制数的整数,我们可以手算。但是当这个数是一个比较大的整数时,要手算是很困难的,甚至是不可能的。因此,我们必须依靠计算机,必须研究出适合于在计算机上实现的有效分解算法。

因此,根据算术基本定理,所谓整数分解,就是要将一个大于1的整数分解成素数的乘积形式,即:

输入:给定一个大于1的整数 N;

输出:如果 N 为合数,输出 N 的所有素因子;

否则指出 N 是素数。

与素数测试类似,最简单最原始的计算机分解整数的算法,也就是基于如下定理的试除法;

定理6. 如果 n 为合数,那么其素因子必定都小于或等于根号 n.

根据定理6,我们可以将算法设计成:假定欲被分解的那个整数为 n。该法用从2开始直至根号 n 为止的所有素数去遍除 n,只要该素数能被 n 整除,那么这个素数就是 n 的一个因子。当然,在具体实现上,还要把握住两个循环,即一个外循环和一个内循环。外循环当然是指从2到根号 n 的那个循环,而内循环则在被找到的那个因子上。例如,对于16,首先找到因子2,此时并不必急于从2跳到3上去执行外循环,而应在2上执行内循环。事实上,对于16024只需在2上再执行9次内循环,便可得到 $1024=2^{10}$。外循环它实际上只执行了一次。

下面是关于这个算法的形式描述:

算法2(试除分解算法):

```
input n                          (输入欲被分解的整数)
k=0,t=0,                         (置初值)
A:if n =1 go to Exit             (如果n=1,则算法终止)
B:q= n/d[k]                      (q为n除以d[k]之商)
r=n mod d[k]                     (r为n除以d[k]之余数)
if r 1 = 0 go to c
else t =t+1                      (找到一个因子)
p[t]=d[k]
q=n
print p[t]
go to A
C: if q>0
k=k+1
go to B
t=t+1
p[t]=n                           (n是一个素数)
print p[t]
Exit:                            (算法之出口)
```

在这个算法中，我们假定 $d[0]=2, d[1]=3, d[3]=5, \cdots, d[k] \leqslant \sqrt{n}$ 为至少直至根号 n 的所有素数。在具体实现上，这些素数可以作为一个外部文件读入程序中。由于这种算法采用试凑性的方法用所有可能的素因子去遍除 N，因此该法一般被称之为试除法。该算法的运行时间与 $\max(p_{t-1}, \sqrt{pt})$ 成比例，即其算法复杂性为 $O(\max(p_{t-1}, \sqrt{p_t}))$.

试除分解算法的可靠性与完备性是显然的，只要时间空间足够，它就可以将 N 的因子全部分解出来。但问题是当被分解的数的位数增多时，其算法的复杂性以指数倍数增长。根据计算复杂性的理论，计算机只能执行具有多项式或小于多项式复杂性的算法。因此，对于这个算法，它没有能力分解那些比如说大于 20 位的整数。由此看来，对于整数分解，更重要的问题不在于找到一个算法，而在于找到一个有效的算法。所谓有效的算法，是指那些具有多项式复杂性的算法，简称为多项式算法。当然，这种算法是不太好找的。事实上，到目前为止，人们还没有找到一种多项式的整数分解算法，但人们又不能否定这种算法存在的可能性。但有一点是可以肯定的，我们至少可以找到比算法 1 更精细更有效的算法。比较现代一点的算法包括基于概率论的 RHO 方法、基于连分数的 CFRAC 方法、基于代数几何的 ECM 法以及基于代数数论的数域筛法 NFS 等等。我们在此便不多叙。

前面我们曾提及，到目前为止，人们还没有找到一种适合于在计算机上执行的“有效”的整数分解算法(即确定型多项式算法)。就今天的算法与机器条件而言，人们分解一个随机的 100 位的整数问题不大(注意，直到本世纪七十年代，人类分解整数的能力只限于 40 位以下)，但对于一个 150 位的整数，就已经相当困难了，当然并非不可能，但对于一个 200 位的整数，目前人们是无论如何也分解不出来的，这也就是说，目前人们还没有找到一种有效的整数分解算法。但是，目前没有找到并不意味着就不存在。事实上，数学家们一直在寻求这种有效的整数分解算法，而这种寻求工作又突出地集中在一种称之为费尔马的数上，因这种数具有特殊形式，相对来讲分解起来要容易些。事实上，数学家们是想以费尔马数为突破口，通过分解这种特殊型式的数寻找规律，研究技巧，为从特殊到一般奠定基础，并期望最终找到一种通用的有效的整数分解算法。

所谓费尔马数，就是以十七世纪法国著名数学家费尔马的名字命名的，具有如下形式的数：

$F_n=2^{2^n}+1, n \geq 0$

早在 1640 年，费尔马便曾猜测所有这种形式的数均为素数，因为他已经知道 F_n 的前五个数全为素数。即：

$F_0=2^{2^0}+1=3$,

$F_1=2^{2^1}+1=17$,

$F_2=2^{2^2}+1=17$,

$F_3=2^{2^3}+1=25$,

$F_4=2^{2^4}+1=65537$,

全为素数。

当然，我们今天都知道，费尔马的猜测是错误的，因为早在 1732 年，瑞士著名数学大师欧拉就已验证 F_5 是合数而不是素数，因为他将 F_5 分解成了素数的乘积形式。即：

$F_5=2^{2^5}+1=4294967297=641 \cdot 6700417$

事实上，就人们今天所掌握的情况而言，我们也就仅知道这前五个费尔马数是素数。至于其它的费尔马数，它们要么是合数，要么其素合性至今仍确定不了。在欧拉后的 148 年间(即 1880 年)，法国数学家 Landry 和 Lasseur 彻底分解了第六个费尔马数 F_6。即他们得到：

$F_6=2^{2^6}+1=2^{64}+1=274177 \cdot 67280421310721$

不过他们是用什么方法分解出这个数的，人们并不清楚，因为他们没有发表他们所用的方法。

第七个费尔马数 F_7 是美国数学家 Morrison 和 Brillhart 在 1970 年应用连分数算法(即 CFRAC 法)分解出来的。

即：$F_7=2^{2^7}+1$

$=2^{128}+1$

$=59649589127497217 \cdot 5704689200685129054721$

连分数法是一个比较通用的分解算法，它也适用于其它形式的整数的分解。

对于第八个费尔马数，即：

$F_8=2^{2^8}+1=2^{256}+1=1238926361552897 \times$

$93461639715357977769163558199606896584051237541638188518028032 1$

人们早在 1909 年就知道它是一个合数，但直到 1980 年才由 Brent 和 Pollard 应用一种基于概率论方法的蒙特卡罗算法(即改进的 rho 方法)将它完全分解成素数的乘积形式的。

时隔十年后(即 1990 年)，英美荷等好几个国家的数学家联合作战，并采用当代最先进的基于代数数论的数域筛法(即 NFS 法)，以及计算机网络技术，尤其是电子邮件技术(即 e－mail)将分布在世界各地的 700 多台计算机相联，最终又在一台巨型机上汇总，前后总共用了 4 个多月，才彻底分解了第九个费尔马数。即：

$F_9=2^{2^9}+1=2^{512}+1$

$=2424833.p_{49}\cdot p_{99}$（$p_{49}$与$p_{99}$分别为具有49位与99位的素数）

在这里值得一提的是，分解F_9所用的NFS算法虽然是一种特殊的方法，但它可望扩展成一般的方法，使之适合于分解一般形式的整数。

第11个费尔马数是澳大利亚计算机科学家Brent于1988年采用椭圆曲线法（即ECM法）在VP100巨型机上彻底分解出来的。即：

$F_{11}=2^{2^{11}}+1$

$=2^{2048}+1=319489.974849.$

$167988556341760475137.3560841906445833$

$920513.p_{564}$（p_{564}为一具有564位数的素数）

目前未被分解的最小的费尔马数，便是第十个费尔马数F_{10}，因此，分解第十个费尔马数是当今国际计算数论界的一个主攻课题。

四、"孙子定理" 我国是一个具有几千年历史的文明古国，是数学的故乡。我们中华民族的先哲为古代数学的创造与发明做出了杰出的贡献，写出了许多优秀的、最早期的数学著作，如【周髀算经】以及【九章算术】等等。【孙子算经】也是我国古代的一部数学巨著。【孙子算经】共分上、中、下三卷。上卷叙述筹算的乘除法，中卷叙述筹算的分数算法及开方方法。下卷则收集了一些算术难题，如已知头数和足数的"鸡兔同笼"问题等等。不过在【孙子算经】中，最有名的要算下卷中的第26题，即所谓的"孙子问题"。这个问题的原文为：

"今有物不知其数，三三数之余二，五五数之余三，七七数之余二，问物几何？"

这就是名扬天下、誉满全球的"孙子定理"之肇源。"孙子定理"在国外则被尊称为"中国剩余定理"或"中国余数定理"。事实上，只要打开国际上任何一本标准数论讲义，都少不了有一个专门的章节来讲述"孙子定理"。稍微现代一点的数论著作则更要大书"孙子定理"在快速计算、计算机设计、密码设计、数值分析、数据库设计以及查询检索等方面的应用。由此可知，"孙子定理"在当代国际数学界中的重要地位。

"孙子问题"可以用代数语言来作更一般的描述，即：

"现有一数X，以m_1除之余r_1，以m_2除之余r_2，以m_3除之余r_3，问X是多少？"

假如我们用同余式$x\equiv r_1(\mathrm{mod}\ m_1)$来表示x，以$m_1$除余$r_1$，则这个问题相当于求解一个一次同余方程组，也即求出满足如下方程：

$$\begin{cases}x\equiv r_1(\mathrm{mod}\ m_1)\\ x\equiv r_2(\mathrm{mod}\ m_2)\\ x\equiv r_3(\mathrm{mod}\ m_3)\end{cases}$$

的解X来。

"孙子问题"在我国民间流传甚广，并有"秦王暗点兵"、"韩信点兵"、"鬼谷算"、"剪管术"、"神奇妙算"、"物不数"、"大衍求一术"，以及"隔墙算"等许多不同的名字。在宋代（420—479年间）的一本算术笔记中，还曾把它的解法写成下面四句诗：

"三岁孩儿七十稀，五留二一事尤奇，

七度上元重相会，寒食清明便可知。"

到了1583年，明代数学家程大位则在他的数学著名【算法统宗】中进一步将其叙述为如下的四句诗，即：

三人同行七十稀，五树梅花廿一枝，

七子团圆月正半，除百零五便得知。"

如把这首诗翻译成白话文，便是：

"以三三除之的余数乘七十，

五五除之的余数乘二十一，

七七除之的余数乘十五，

总加之并减去105的倍数，

便为所求。"

若列成算式，则为：

$$2.70+3.21+2.15-210=23.$$

程大位的这个解法是非常奇妙的，尤其是70，21，15这三个数，更是设计得巧妙极了，真可谓是神奇妙算！不过，程氏的这个解法并不是凭空捏造的，而是以一定的天文现象作背景的，这大概就是为什么我国古代的很多数学家，如祖冲之父子等，同时也是天文学家之缘故吧！另外，用诗的形式来解答数学问题，这也算是我们祖先的一大发明！

"孙子问题"的进一步发展，是将它总结成如下这个"孙子定理"。

定理7（孙子定理）：设$m_1,m_2,\cdots m_n$均为>1的两两互素的正数，$a_1,a_2\cdots a_n$为任意整数，则下列同余方程组：

$$\begin{cases}x\equiv a_1(\mathrm{mod}\ m_1),\\ x\equiv a_2(\mathrm{mod}\ m_2),\\ \cdots\cdots,\\ \cdots\cdots,\\ \cdots\cdots,\\ x\equiv a_n(\mathrm{mod}\ m_n)\end{cases}$$

具有一个解x，如果有两个解x和x'的话，那么$x\equiv x'(\mathrm{mod}\ M)$，其中，$M=m_1m_2\cdots m_n$.

关于这个定理的证明，读者可以查阅任何一本数论书，我们便不在此赘述。关于这个定理的重要性，我们要强调三点：

1. 虽然孙子本人并没有形式地提出这个定理，但这个定理的思路完全是从"孙子问题"中得来的，

因此我们完全可以说这个定理是孙子(注:国外文献中尊称孙子为孙大师)发明的。

2. 这个定理的证明的本身就提供了一种求解同余方程组的完整算法,并且这种证明完全可以是构造性的,因此它极易在计算机上实现。从这个角度讲,孙子也是一名出色的计算机科学家(因为计算机科学的核心是研究算法),尽管那时还没有计算机。美国著名计算机科学家 Knuth 称古巴比伦人是优秀的计算机科学家,事实上,我们中华民族的先哲也是优秀的计算机科学家。

3. 孙子定理具有很强的应用性,它在数值分析,计算机设计,计算机算术,密码设计,最优 hash 函数设计,数据库保密,快速计算以及环论等许多领域中都有重要应用。例如当代最有名的 RSA 密码技术,实际上它翻来复去就是用孙子定理而已,孙子定理成了它不可缺少的一个重要工具。据作者统计分析,孙子定理完全可以与古希腊人发明的著名的欧几理德算法相提并论、平起平坐,甚至比它还重要,还常用。

下面,我们给出"孙子定理"的一个 C 语言的程序实现:

```
#include <stdio.h>
#include<stdlib.h>
#define MAXi 100
int inv(a,n)
int a,n;
{
int i,b ,t ,X,Y;
int g[MAX i];
int x[MAX i];
int y[MAX i];
g[0]= n;g[1]=a;
x[0]=1;x[1]=0;
y[0]=0;y[1]=1;
i=1;
while (g [i]! =0)
{
t=g[i-1]/g[i];
g[i+1]=g[i-1]-t*g[i];
x[i+1]=x[i-1]-t*x[i];
y[i+1]=y[i-1]-t*y[i];
i=i+1;
}
x=y[i-1];
if (x)=0) y=x;
else Y=X+n;
return (Y);
}
main()
{
/*CRT(n,p[1],...,p[r],x[1],...x[r])
(return x=[0,n-1]) */
int i,t ,n,crt;
int x[MAXi];
int p[MAX i];
int y[MAX i];
while (1)
{
n=1;
crt=0;
printf("\n Chinese Remainder Theorem");
printf("\n Computing x=x[i] mod p[i]");
printf("\n ————————————\n");
printf(" Enter Integers t:")"
Scanf("%d",& t);
for (i=1;i<=t;i++)
{
printf("Enter Integers x[i]and p[i]:")
scanf("% d %d",&x[i],&p[i]);
n=n*p[i];
}
for (i=1;i<=t;i++)
{
y[i]=0;
}
for (i=1;i<=t ;i++)
{
y[i]=inv((n/p[i])%p[i],p[i]);
printf(" inv=%d\n",y[i]);
}
for (i=1;i<=t;i++)
crt=(crt + (n/p[i]*y[i]* x[i])%n;
printf(" crt =%d\n",crt);
}
}
```

通过运行这个程序,我们可以求解各种各样的同余方程组。比如我们可以用它来求解【孙子算经】中的"孙子问题":

$$\begin{cases} x\equiv 2 \pmod 3, \\ x\equiv 3 \pmod 5, \\ x\equiv 2 \pmod 7。 \end{cases}$$

通过调用、执行上面的程序,我们可得如下结果:

```
Chinese Remainder Theorem
Computing x=x[i] mod p[i]
——————————
Enter Integers t:3
Enter Integers x[i] and p[i]:23
Enter Integers x[i] and p[i]:35
Enter Integers x[i] and p[i]:27
x=23
```

其中的 Enter 部分表示需要用户输入信息,如方程组中方程的个数 t,以及诸方程中 x 与模的值等,最

后一行则是所求x之值。

五、"哥德巴赫猜想" 早在1742年6月7日，德国数学家哥德巴赫便写信给瑞士数学家欧拉。在信中哥德巴赫猜测：任何一个大于4的偶数，都可以表示成两个素数之和。例如，下面这些等式显然满足哥德巴赫猜想：

6=3+3

8=3+5

……

14=3+11=7+7

100=3+97=11+89=17+83=29+71=41+59=47+53

……

以第一个式子为例，6(是一个偶数)可以表示成两个素数(即两个3，3是素数)之和。其余的式子均可依此类推。

欧拉并没有能够证明这个猜测的正确性，尽管欧拉号称是解难题的能手。在1922年，英国著名数学家哈代和李特伍德就曾指出：哥德巴赫的正确性是至信无疑的，但却谁也无法证明其正确性。

对于一般的科学家而言，如果我们做了成千上万次试验，结果都是一个，那么我们完全可以说这个结果是正确的。但是，对于数学家来讲，就是做了一百亿次试验，并且次次都正确，我们也不能说这个结果是正确的，因为在数学中，一百亿是个很小的数字。

到目前为止，关于哥德巴赫猜想的最好的理论结果是我国数学家陈景润在1966年得出的结果(尽管他的最终结果是迟至1973年才发表的)。他证明了任何一个充分大的偶数都可以表示成一个素数再加上两个素数之积(即所谓的1+2)。由于两个素数之积并不是一个素数，例如2乘以3等于6就不是一个素数，尽管2和3都是素数，况且这个充分大的偶数到底有多大、到底是多少，目前我们并不知道，因此，陈景润教授的结果与哥德巴赫的原本猜想还相距甚远。

目前，在解决哥德巴赫猜想方面，有两种主要的方法：一种是解析数论的方法，另一种是计算数论的方法。就解析数论来讲，常规的圆法与筛法对哥德巴赫猜想已是力不从心了，必须寻求新的方法。就计算数论来讲，目前最好的结果，是由三个荷兰数学家计算出来的，他们于1989年在巨型向量计算机CDC CYBER 205上验证了哥德巴赫猜想一直到2.10^{10}都正确。可是，在数学上，2.10^{10}是一个小的数字，对于比这个数还大的数是否也满足哥德巴赫猜想呢？目前人们并不知道。就是有一天人们能够突破(并不是很困难!)这个数字即(2.10^{10})而得到一个新的数字，但比这个新的数还大的那个数是否也满足哥德巴赫猜想呢?对于这些问题，目前人们可以说是根本无法回答。

为什么解决哥德巴赫猜想会如此之困难呢？首先，在数学理论上，人们还没有找到一种强有力的新的证明方法，该方法可以证明任何一个大于4的偶数都可以表示成两个素数之和。其次，在计算理论上，目前人们还没有找到一种有效的算法，该算法能够在多项式时间内，在计算机上予以解决。在这里，我们要特别强调，计算机只能执行具有多项式或比多项式还小如对数、线性复杂性的算法，而对于指数复杂性以及比指数复杂性还大(如阶乘复杂性)的算法，计算机是无能为力的。可是，目前人们所用于解决哥德巴赫猜想的一些计算机算法基本上都是基于穷举式的，其复杂性都远远超过了多项式，而在指数范围内，因此，这些算法当然不可能在现有的计算机条件下解决哥德巴赫猜想。

由于哥德巴赫猜想至少是具有计算上的难解性，即目前人们还没有找到具有多项式复杂性的计算机算法，因此，在人们还没有发明出新的强有力的数学工具之前、或在没有发明出新的有效的(即具有多项式复杂性的)计算机算法之前，哥德巴赫猜想是不可能得到解决的。在这里，我们认为值得一提的是，在数论中，一个问题两百多年没有得到解决是一点也不奇怪的，因为有的问题甚至花了两千多年才得到解决。

下面我们给出荷兰数学家们设计的几个计算哥德巴赫猜想的FORTRAN程序，供有兴趣的读者参考：

哥德巴赫算法Ⅰ：

```
C
C WE ASSUME THE INTEGER ARRAY PIND(N) HAS
BEEN INITIALIZED TO ZERO
C
DO 20 N =N1,N2,2
C WE SUPPOSE THAT N1 AND N2 ARE EVEN;
C WE SERACH FOR THE MINIMAL GOLDBACH PARTI-
TION OF N
C
DO 10 I=1,IMAX1
    IF (PRIME (N-PR(I))) THEN
PIND(N)=I
GO TO 20
ENDIF
10 CONTINUE
C NO GOLDBACH PARTITION N =P+Q FOUND WITH p
< =PR(IMAX1)
C INCREASE THE VALUE OF IMAX1
C
20 CONTINUE
```

哥德巴赫猜算法Ⅱ

```
C
C WE ASSUME THE INTEGER ARRAY PIND(N) AND
```

```
ODDPR HAVE BEEN INITIALIZED TO ZERO
C
DO 20 N=1,IMAX2
C WE SUPPOSE THAT N1 AND N2 ARE EVEN;
C WE SERACH FOR THE MINIMAL GOLDBACH PARTI-
TION OF N
C
DO 10 I=1,IMAX1
PRI =PR(I)
DO 10 I=N1,N2,2
IF (PIND(N) ,EQ. 0. AND. ODDPR(N-PRI). EQ. 1) PIND
(N)=I
PIND(N)=I
10 CONTINUE
20 CONTINUE
C TREAT THE EVEN N FOR WHICH PIND (N) IS STILL
ZERO,
C I. E. FOR WHICH NO GOLDBACH PARTITION HAS
BEEN FOUND YET
C
CO 40 I=N1,N2,2
IF (PIND(N). GT. 0)GOTO 40
DO 30 I=IMAX2+1,IMAX1
IF (ODDPR(N-PRI(I). EQ. 1)THEN
PIND(N)=I
GOTO 40
END IF
30 CONTINUE
40 CONTINUE
```

六、结束语　在本文中，我们论述了计算机与数论之间的联系，指出它们是互相依赖、互相联系、互相影响、互相渗透、互相促进的。计算机在数论中有着广泛的应用，数论也在计算机科学中有着深入的应用，而计算数论则又是这两门学科交叉渗透而形成的一门边缘学科。就数论本身来讲，除了传统的初等数论外，它已逐步形成了诸如组合数论，代数数论，解析数论，概率数论，几何数论等等，而计算数论则与所有这些数论分支都有关系，因为它们都少不了计算，都离不开计算机。因此，计算数论就象一座桥梁一样，沟通了数论诸分支之间的联系。同时，它又大量应用着计算机科学中的研究成果，尤其是并行算法，巨型机方面的研究成果。计算机在数论中的应用是如此之广泛，而计算数论的内容又是如此之丰富，使得我们不可能在这样一篇文章中介绍很多的内容。我们只能提纲挈领地、简明扼要地介绍计算机在素数测试、整数分解、孙子定理以及哥德巴赫猜想等数论领域中的应用。至于数论在计算机科学中的应用，我们则基本上没有提及，我们期望本文能对那些对数论感兴趣的计算机科学与应用人员有所帮助。我们在下面列出了一些参考文献，供有兴趣的读者去进一步参阅。就目前来讲，有关计算数论方面的资料，大都散发在不同的刊物(杂志)之中，并且技术性很强，不易阅读，从这个角度讲，本文作者所著的【计算数论导引】一书，很值得大家阅读、参考。

参考文献

1. L. Adleman and k. s. mcCurley, Open Problems in Number Theoretic Algorithms, in: Discret Algorithms and Complexity, Academic Press, 1987, 237－262.
2. E. Bach, Intractable Problems in Number Theory, in: Advances in Cryptology － － CRYPTO'89, Lecture Notes in Computer Science 403, Springer － Verlag, 1989, 77－93.
3. E. Bach, Number－Theoretical Algorithms, Annual Revivew of Computer Science, 1990, 119－172.
4. R. P. Brent, Parallel Algorithms for Integer Factorization, CMR－R49－89, Certer for Mathematical Analysis, Australian National University, Canberra, 1989
5. J. D. Dixon, Factorization and Primality Tests, The American Mathematical Monthly, 91, 1984, 333－352
6. A. Granville, J, van de Lune and H. J. J. te Riele, Checking the Goldbach Conjecture on a Vector Copmuter, Report NM－R8812, Dept of Numerical Mathemaics, CWI, Amsterdam, 1988.
7. G. H. Hardy and E. M. Wright, An Introduction to the Theory of Numers, Clarendos Press, Oxford, 5th Edition, 1979.
8. D. S. Johnson, The NP－Completeness Column: An Ongoing Guide, Journal of Algorithms, 7, 1986, 584－601.
9. A. K. Lenstra, H, W. Lenstra, Jr, M, S. Manasse and J. M. Pollard, The Number Field Sieve, in: proc. 22nd Annual ACM Symp. on Theory of Computation, Baltimore, May 14－16, 1990, 564－572.
10. A. K. Lenstra and M. S. Manasse, Factoring by Electronic Mail; in: Advances in Cryptology － － EUROCRYPT'89, Lecture Notes in Computer Science, Springer－Verlag, 1990, 355－371.
11. D. E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, 2nd Edition, Addison－Wesley, 1981.
12. H. J. J. te Riele, Parallel Processing in Number－Theoretical Problems, Report NM－R9010, Dept of Numerical Mathematics, CWI, Amsterdam, 1990.
13. R. L. Rivest, A. Shamir and L. Adleman, A Method for Obtaining Digital Signatures and Public Key Cryptosystems, Communications of the ACM, 21, 2, 1978, 120－126.
14. S. S. Wagstaff, Jr. and J., W. Smith, Methods of factoring Large Integers in: Number Theory, edited by D. V. Chudonvsky, G. V. Chudnovsky, H. Cohn and M. B. Nathanson, Lecture Notes in Mathematics, Springer－Verlag, 1984, 281－303.
15. S. Y. Yan, Introduction to Computational Number Theory, to be published in Gordon and Breach Science Publishers, London. (颜松远，计算数论导引，伦敦，待出版)