Journal of Computer Applications
计算机应用,2015,35(6):1673 - 1677,1697
ISSN 1001-9081
CODEN JYIIDU
2015-06-10
http://www.joca.cn

文章编号:1001-9081(2015)06-1673-05
doi:10.11772/j.issn.1001-9081.2015.06.1673

基于 *k*-ary 消减的快速最大公约数算法

王广赛[1,2*],曾　光[1,2],韩文报[1,2],李永光[1,2]
(1.信息工程大学,郑州 450001;　2.数学工程与先进计算国家重点实验室,郑州 450001)
(*通信作者电子邮箱 sswang1990@yeah.net)

摘　要: 最大公约数(GCD)算法中,对于输入 B 和 C,利用 Sorenson 的右移 k-ary 消减思想提出一个算法用于寻找整数 x 和 y,使得 x 和 y 满足 $Bx - Cy$ 在二进制表示下低比特位部分为 0,即 $Bx - Cy = 0 \pmod{2^e}$,其中 e 是常数正整数。利用该算法能够右移较多比特并大规模降低循环次数。再结合模算法,提出了快速 GCD 算法,其输入规模为 n 比特时最差复杂度仍然是 $O(n^2)$,但最好的情况下复杂度能达到 $O(n \log^2 n \log\log n)$。实验数据表明,对于 20 万以上比特规模的输入,快速 GCD 算法比 Binary GCD 算法速度快;对 100 万比特规模的输入,快速 GCD 算法速度是 Binary GCD 算法的两倍。

关键词: 最大公约数算法;欧几里得算法;二进制最大公约数算法;右移 k-ary 消减;整数最大公约数算法
中图分类号: TP309　　**文献标志码:** A

Fast greatest common divisor algorithm based on *k*-ary reduction

WANG Guangsai[1,2*], ZENG Guang[1,2], HAN Wenbao[1,2], LI Yongguang[1,2]
(1. *Information Engineering University, Zhengzhou Henan* 450001, *China*;
2. *State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou Henan* 450001, *China*)

Abstract: Greatest Common Divisor (GCD) is one of the basic subjects in computational number theory. It has a wide application in encryption and analysis of cryptography. For inputting B and C, an algorithm based on right-shift k-ary reduction proposed by Sorenson was presented for finding the integers x and y which satisfy the least significant bits of $Bx - Cy$ were 0, i.e., $Bx - Cy = 0 \pmod{2^e}$ where positive integer e was a constant. It could do a lot of right shifts and reduce a large number of cycles with taking advantage of the algorithm for finding the integers x and y. A fast GCD algorithm was proposed combined with modulus algorithm. When the size of the input was n bits, the worst complexity of the fast GCD algorithm was still $O(n^2)$. In the best case, the complexity of the proposed algorithm could achieve $O(n \log^2 n \log\log n)$. The experimental data show that actual implementations given input about more than 200 000 bits, the fast GCD algorithm is faster than the Binary GCD algorithm, and the fast GCD algorithm is twice as fast as the Binary GCD algorithm for 1 million bits of input.

Key words: Greatest Common Divisor (GCD) algorithm; Euclidean algorithm; binary Greatest Common Divisor (GCD) algorithm; right-shift k-ary reduction; integer greatest common divisor algorithm

0　引言

求最大公约数(Greatest Common Divisor,GCD)算法在计算代数、密码学中都有广泛的应用。文献[1]中指出,在典型的代数计算中有一多半的时间都花在大数的 GCD 计算上。在密码学中,GCD 算法主要出现在大数分解和素性判别中,GCD 算法的扩展算法普遍被用于求逆。

GCD 算法从提出至今已经被广泛研究。在 GCD 算法中比较著名的算法是辗转相除法,也叫作 Euclidean 算法,该经典算法被 D. Knuth 称为所有算法的祖先[2]。二进制 GCD(Binary GCD)算法主要适合于二进制算数,由 Stein[3] 于 1961 年提出,对于小整数的 GCD 计算有很好的实现效率。PM GCD(PlusMinus GCD)算法[4] 是 Binary GCD 算法的一个推广,它在硬件实现上有很大的优势。对于输入规模为 n 比特的两个整数,这些算法平均复杂度都是 $O(n^2)$。

GCD 计算的第一个准线性时间算法由 D. Knuth 于 1970 年提出,这个 GCD 算法基于快速傅里叶变换(Fast Fourier Transformation,FFT)乘法,能在 $O(n \log^5 n \log\log n)$ 的时间内计算两个 n 位整数的 GCD,该算法的复杂度被 Schönhage 改进为 $O(n \log^2 n \log\log n)$[5]。虽然 Knuth-Schönhage 算法被认为是最快的 GCD 算法,但是在软件实现时过于复杂不实用且由于基于 FFT 乘法导致只能对大规模输入加速。2004 年,Stehlé 等[6] 提出一种迭代 GCD 算法,该算法在复杂度方面上没有提高,但实现了对 Knuth-Schönhage 算法的简化。

近二十年来,许多针对 GCD 算法的改进被提出。1994 年,Sorenson[7] 提出了对 Binary GCD 算法思想的推广的右移 k-ary 消减方法,它有许多不同的版本,包括 Sorenson[7] 的左移、右移,以及 Jebelean[8] 和 Weber[9] 的右移 k-ary GCD 算法。

收稿日期:2015-01-05;**修回日期:**2015-03-29。
基金项目:国家自然科学基金资助项目(61003291);数学工程与先进计算国家重点实验室开放课题基金资助项目(2013A03,2013A10)。
作者简介:王广赛(1990 -),男,河南鹿邑人,硕士研究生,CCF 会员,主要研究方向:公钥密码;　曾光(1980 -),男,吉林吉林人,副教授,博士,主要研究方向:网络密码;　韩文报(1963 -),男,河北邯郸人,教授,博士生导师,主要研究方向:密码理论、网络安全;　李永光(1988 -),男,河南淮阳人,硕士研究生,主要研究方向:分组密码。

基于右移 k-ary 消减思想提出的算法中，Weber[9] 的算法加速效果最好，可以对 Bianry GCD 算法提速 4.5 倍，然而这种提速只是针对 4 096 比特规模的输入。在考虑混合 Binary GCD 算法和 Euclidean 算法的算法中，文献[10]提出了结合两种算法优势的算法；但是该算法并不能对算法的复杂度 $O(n^2)$ 进行优化，而且在解决较大规模的输入上并不能做到实现上的优化，更适合于并行实现。Mohamed[11] 提出一种混合算法，该算法的优点是便于实现且当减法操作与除法操作时间花费相比较小时该算法表现最优；但是该算法在速度方面上并不比已有算法有显著提高。对于更大规模的输入，Knuth-Schönhage 算法复杂度较高；但是在软件实现时不实用且由于基于 FFT 乘法导致只能对大规模输入加速。同样地，虽然文献[6]中算法实现了对 Knuth-Schönhage 算法的简化，但是在软件实现上依然比较复杂。本文基于右移 k-ary 消减思想提出一种实现起来相对简单的快速 GCD 算法，而且该算法能对 20 万比特以上的输入实现对 Binary GCD 算法的加速。

右移 k-ary 消减想法是寻找整数 x、y 使得对于给定的输入 B 和 C 满足 $Bx - Cy \equiv 0 \bmod k$，要求 B、C 和 k 都互素，k 是正整数。对于这样 的 x、y，由于 $Bx - Cy$ 被 k 整除，如果 x、y 小于 $\sqrt{k}$，那么 $(Bx - Cy)/k$ 的比特长度比 B 减少约 0.5 lb k 比特。本文的思路是在 Binary GCD 的设计基础上进一步利用右移 k-ary 消减，可以假定输入是奇数，如果输入不是奇数则可以通过简单的预处理转换为求两个奇数的最大公约数，所以本文考虑的输入都是奇数，在计算过程中产生的偶数都可以右移转换为奇数继续计算下去，这一点可以由 Binary GCD 的理论分析保证。那么当 k 为 2 的方幂时就满足了 B、C 和 k 互素的条件。

Euclid 算法是从最高比特位优先的角度减小输入整数的规模，Binary GCD 则是从最低比特位考虑。本文将结合 Binary GCD 和 Euclid 算法的基本思想，同时考虑输入整数的最低位和最高位，给出一个快速 GCD 算法，为方便说明，记作 fast_gcd 算法。fast_gcd 算法的主要难点在于寻找辅助因子 x 和 y 的算法，记作 findxy_gcd 算法。findxy_gcd 算法结合模算法提出 fast_gcd 算法。本文的算法不同于 Binary GCD 算法和 PM GCD 算法的一次循环内有一次相减或相加操作和一次右移操作，在一次循环内计算 $Bx - Cy$ 并将其右移以减少更多比特。对于相同规模的输入，fast_gcd 算法能大规模减少循环量，并且利用模算法来提高算法速度，这是因为对于两个不同长度的输入，模算法能快速降低输入的规模。

本文所有操作都在二进制上进行。对任意正整数 A，记 $Val(A)$ 为满足 $A = ((A >> x) << x)$ 的 x 的最大值，这里 x 为正整数。记 $L(A)$ 为 A 以二进制表示时的比特长度。

1 findxy_gcd 算法

本章首先提出递归算法(recursive_gcd 算法)，该算法能寻找至少一组合适的 x、y，使得 $Bx - Cy$ 的低比特部分为 0。接着该算法以寻找一组合适的 x、y，使得 $Bx - Cy$ 尽量多的低比特部分为 0，这是为了能让 $Bx - Cy$ 尽量右移，大规模减少 B 的长度，虽然 $Bx - Cy$ 越多的低比特部分为 0，x、y 的长度也随之增加，但是相比 x、y 长度的增加，$Bx - Cy$ 能右移规模增加更多，这样就达到了更快地降低输入规模的目的。

虽然 recursive_gcd 算法能返回至少一组辅助因子 x 和 y，但是由于其失败情形的出现，导致不能最快返回辅助因子，所以才需要 findxy_gcd 算法寻找辅助因子 x 和 y 算法以最快速度返回最合适的 x 和 y。

1.1 递归算法：recursive_gcd 算法

记 recursive_gcd 算法的输入为 op_1、op_2 和 n，对于输入的要求是 $n \geqslant L(op_1) \approx L(op_2)$。recursive_gcd 算法能返回两组值 (x_1, y_1)、(x_2, y_2)。因为要求 recursive_gcd 算法的返回值能被递归调用下去，所以这里返回的两组值只有两种情况：第一种情况是 $(x_2, y_2) = (0, 0)$，(x_1, y_1) 满足 $Val(op_1 \cdot x_1 - op_2 \cdot y_1) \approx 2L(x_1)$ 或 $Val(op_1 \cdot x_1 - op_2 \cdot y_1) > 2L(x_1)$，因没有能找到两组非零值以递归下去，所以称这种情况是失败的；第二种情况是 (x_1, y_1)、(x_2, y_2) 满足 $x_1 \cdot y_2 \neq x_2 \cdot y_1$，$Val(op_1 \cdot x_1 - op_2 \cdot y_1) \geqslant n$，$Val(op_1 \cdot x_2 - op_2 \cdot y_2) \geqslant n$，且还满足对于 $x = x_1, y_1, x_2$ 或 y_2，都有 $L(x) \approx n/2$ 或 $L(x) \leqslant n/2$，因为这种情况找到两组非零值可以递归下去，所以称这种情况是成功的。

1.1.1 辅助因子的存在性理论证明

为了说明 recursive_gcd 算法理论上是可行的，需要下面的推论 2。

引理 1[7,12] 令 $k > 1$ 为一个正整数，对于整数 u 和 v，必然存在一对整数 a 和 b，满足 $0 < |a| + |b| \leqslant 2\lceil\sqrt{k}\rceil$，使得 $au + bv \equiv 0 \pmod k$。

容易由引理 1 得到推论 1，这里将不再证明。推论 2 可以由推论 1 简单证明得到。

推论 1 令 $k = 2^n$，对于奇数 u 和 v，$u < 2^n$，$0 < v < u$，必然存在一对整数 a 和 b，满足：

$$0 < L(|a|) \leqslant (n+1)/2 + 1$$
$$0 < L(|b|) \leqslant (n+1)/2 + 1$$

使得 $au + bv \equiv 0 \pmod k$。

推论 2 对于奇数 u 和 v，$u > v > 0$，$n \geqslant L(u)$，必然存在一对奇数 x 和 y，满足 $L(|x|) \leqslant (n+1)/2 + 1$ 和 $L(|y|) \leqslant (n+1)/2 + 1$，使得 $ux + vy \equiv 0 \pmod{2^{L(|x|)+\lfloor(n-3)/2\rfloor}}$。

证明 由推论 1 知道存在整数 a 和 b，满足：

$$0 < L(|a|) \leqslant (n+1)/2 + 1$$
$$0 < L(|b|) \leqslant (n+1)/2 + 1$$

使得 $au + bv \equiv 0 \pmod{2^n}$，由于 u 和 v 为奇数，则 $uv^{-1}a \equiv b \pmod{2^n}$。令 $x = a/2^{Val(a)}$，$L(|x|) \leqslant (n+1)/2 + 1$。则 $uv^{-1}x \cdot 2^{Val(a)} \equiv b \pmod{2^n}$，由于 $Val(a) < n$，那么 $Val(b) = Val(a)$。再令 $y = b/2^{Val(a)}$，$L(|y|) \leqslant (n+1)/2 + 1$，则：

$$uv^{-1}x \cdot 2^{Val(a)} \equiv y \cdot 2^{Val(a)} \pmod{2^n}$$

那么：

$$uv^{-1}x \equiv y \pmod{2^{n-Val(a)}}$$

又知：

$$0 < L(|a|) \leqslant (n+1)/2 + 1$$

即 $0 < L(|x|) + Val(a) \leqslant (n+1)/2 + 1$，则：

$$L(|x|) + \lfloor(n-3)/2\rfloor \leqslant L(|x|) + \lfloor(n-3)/2\rfloor \leqslant n - Val(a)$$

那么 $uv^{-1}x \equiv y \pmod{2^{L(|x|)+\lfloor(n-3)/2\rfloor}}$。 证毕。

推论2说明对于两个奇数B和C,$B>C$,$n=L(|B|)$,必然存在奇数x和y,满足:

$L(|x|)\leqslant(n+1)/2+1$和$L(|y|)\leqslant(n+1)/2+1$使得:$Bx+Cy\equiv 0(\bmod 2^{L(|x|)+\lfloor(n-3)/2\rfloor})$或$Bx-Cy\equiv 0(\bmod 2^{L(|x|)+\lfloor(n-3)/2\rfloor})$。

那么对于 recursive_gcd 算法理论上至少可以返回一组非零值,也就说明了 recursive_gcd 算法理论上是可行的。

1.1.2　递归算法的实现

实现 recursive_gcd 算法,也就是该算法的递归部分。

输入　op_1,op_2,n,op_1和op_2为正奇数,$n\geqslant L(op_1)$。

输出　rop_1,rop_2,rop_3,rop_4,返回值只返回两种情况,一种是失败的情况,rop_1和rop_2非零,$rop_3=rop_4=0$;另一种情况是返回两组值,满足$rop_1\cdot rop_4\neq rop_2\cdot rop_3$。

1) if $n<cons$,穷举搜索并返回$(rop_1,rop_2,rop_3,rop_4)$;　//cons 为某常数
2) $hn=(n+3)/2$;　// 称 hn 为截断比特长度
3) $B\leftarrow op_1;C\leftarrow op_2;b=B\bmod 2^{hn};c=C\bmod 2^{hn}$;
4) $(x_{1b},y_{1b},x_{2b},y_{2b})=$ recursive_gcd(b,c,hn);
5) if $x_{2b}==0$,返回$(x_{1b},y_{1b},0,0)$;　// 只能返回一组非零值
6) $b=\text{abs}(B\cdot x_{1b}-C\cdot y_{1b});c=\text{abs}(B\cdot x_{2b}-C\cdot y_{2b})$;　//abs 表示绝对值函数,返回正整数
7) if $b==0$或$c==0$或$b==c$,返回$(x_{1b},y_{1b},0,0)$;　// 只能返回一组非零值
8) $b=b/Val(b);c=c/Val(c);b_1=b\bmod 2^{hn};c_1=c\bmod 2^{hn}$;
9) $(x_{1b_1},y_{1b_1},x_{2b_1},y_{2b_1})=$ recursive_gcd(b_1,c_1,hn);
10) if $Val(B_1)>Val(C_1)$且$B\cdot x_{1b}-C\cdot y_{1b}>0$且$B\cdot x_{2b}-C\cdot y_{2b}>0$,令:
11) $x_{1B}=x_{1b_1}\cdot x_{1b}-y_{1b_1}\cdot x_{2b}\cdot 2^{Val(B_1)-Val(C_1)}$,
$y_{1B}=x_{1b_1}\cdot y_{1b}-y_{1b_1}\cdot y_{2b}\cdot 2^{Val(B_1)-Val(C_1)}$,
12) $x_{2B}=x_{2b_1}\cdot x_{1b}-y_{2b_1}\cdot x_{2b}\cdot 2^{Val(B_1)-Val(C_1)}$,
$y_{2B}=x_{2b_1}\cdot y_{1b}-y_{2b_1}\cdot y_{2b}\cdot 2^{Val(B_1)-Val(C_1)}$;
13) 其他情况可类似得到$x_{1B},y_{1B},x_{2B},y_{2B}$;
14) if $x_{2b_1}==0$,令$rop_1\leftarrow x_{1B};rop_2\leftarrow y_{1B}$;
15) 返回$(rop_1,rop_2,0,0)$;　// 只能返回一组非零值
16) $rop_1\leftarrow x_{1B};rop_2\leftarrow y_{1B};rop_3\leftarrow x_{2B};rop_4\leftarrow y_{2B}$;
17) if $rop_1\cdot rop_4==rop_2\cdot rop_3$,令$rop_3=0;rop_4=0$;
18) 返回$(rop_1,rop_2,rop_3,rop_4)$。

首先分析对于 recursive_gcd 算法成功的情形是如何保证递归成功的。

对于输入B、C和$L(B)$,假设对于长度小于$L(B)$的任意一组输入 recursive_gcd 算法都能够返回两组非零值。

令$b=B\bmod 2^{(L(B)+1)/2}$,$c=C\bmod 2^{(L(B)+1)/2}$,调用 recursive_gcd 算法,输入为b,c和$(L(B)+1)/2$,得到(x_{1b},y_{1b}),(x_{2b},y_{2b}),都满足$L(x)\leqslant L(B)/4+2$(这里$L(x)\leqslant L(B)/4+2$并不是算法的要求,实际上算法要求是$L(x)\leqslant L(B)/4$或$L(x)\approx L(B)/4$,这里这么要求只是为了说明方便)。由假设知:

$$Val(|x_{1b}\cdot b-y_{1b}\cdot c|)\geqslant(L(B)+1)/2$$
$$Val(|x_{2b}\cdot b-y_{2b}\cdot c|)\geqslant(L(B)+1)/2$$

记$B=B'\cdot 2^{(L(B)+1)/2}+b$,$C=C'\cdot 2^{(L(B)+1)/2}+c$。容易得到:

$$Val(|x_{1b}\cdot B-y_{1b}\cdot C|)\geqslant(L(B)+1)/2$$
$$Val(|x_{2b}\cdot B-y_{2b}\cdot C|)\geqslant(L(B)+1)/2。$$

为了讨论方便,不妨设$x_{1b}\cdot B-y_{1b}\cdot C>0$,$x_{2b}\cdot B-y_{2b}\cdot C>0$,令:

$$B_1=|x_{1b}\cdot B-y_{1b}\cdot C|$$
$$C_1=|x_{2b}\cdot B-y_{2b}\cdot C|$$
$$b_1=(B_1/2^{Val(B_1)})\bmod 2^{(L(B)+1)/2}$$
$$c_1=(C_1/2^{Val(C_1)})\bmod 2^{(L(C)+1)/2}$$

再把b_1和c_1和$L(b_1)$当作 recursive_gcd 算法的输入得到两组值,记为(x_{1b_1},y_{1b_1})、(x_{2b_1},y_{2b_1}),都满足$L(x)\leqslant L(B)/4+2$。则:

$$Val(|x_{1b_1}\cdot b_1-y_{1b_1}\cdot c_1|)=Val(|x_{1b_1}\cdot B_1/2^{Val(B_1)}-y_{1b_1}\cdot C_1/2^{Val(C_1)}|)\geqslant(L(B)+1)/2$$
$$Val(|x_{2b_1}\cdot b_1-y_{2b_1}\cdot c_1|)=Val(|x_{2b_1}\cdot B_1/2^{Val(B_1)}-y_{2b_1}\cdot C_1/2^{Val(C_1)}|)\geqslant(L(B)+1)/2$$

那么可以得到:

$$Val(B_1)=Val(|x_{1b}\cdot B-y_{1b}\cdot C|)\geqslant(L(B)+1)/2$$
$$Val(C_1)\geqslant(L(B)+1)/2$$

不妨设$Val(B_1)\geqslant Val(C_1)$,则:

$$Val(|x_{1b_1}\cdot B_1-y_{1b_1}\cdot C_1\cdot 2^{Val(B_1)-Val(C_1)}|)\geqslant L(B)$$
$$Val(|x_{2b_1}\cdot B_1-y_{2b_1}\cdot C_1\cdot 2^{Val(B_1)-Val(C_1)}|)>L(B)$$

综上,令:

$$x_{1B}=x_{1b_1}\cdot x_{1b}-y_{1b_1}\cdot x_{2b}\cdot 2^{Val(B_1)-Val(C_1)}$$
$$y_{1B}=x_{1b_1}\cdot y_{1b}-y_{1b_1}\cdot y_{2b}\cdot 2^{Val(B_1)-Val(C_1)}$$
$$x_{2B}=x_{2b_1}\cdot x_{1b}-y_{2b_1}\cdot x_{2b}\cdot 2^{Val(B_1)-Val(C_1)}$$
$$y_{2B}=x_{2b_1}\cdot y_{1b}-y_{2b_1}\cdot y_{2b}\cdot 2^{Val(B_1)-Val(C_1)}$$

可以得到:

$$Val(|x_{1B}\cdot B-y_{1B}\cdot C|)\geqslant L(B)$$
$$Val(|x_{2B}\cdot B-y_{2B}\cdot C|)\geqslant L(B)$$

而x_{1B}满足:

$$L(x_{1B})\leqslant\max\{L(x_{1b_1})+L(x_{1b}),L(y_{1b_1})+L(x_{2b})+Val(B_1)-Val(C_1)\}\leqslant L(B)/2+4+Val(B_1)-Val(C_1)$$

一般情况下$Val(B_1)-Val(C_1)$比较小,而当$Val(B_1)-Val(C_1)$较大时$L(y_{1b_1})$又会比较小,从而得到$L(x_{1B})\approx L(B)/2$或$L(x_{1B})\leqslant L(B)/2+con$,$con$是指一个小常数,与$L(B)$无关,$L(y_{1B})$、$L(x_{2B})$和$L(y_{2B})$也有类似性质。那么该结果可用于被递归调用,完成了 recursive_gcd 算法的全部过程。

对于第一种情形,也即失败的情形,当递归调用的返回值导致$x_{1B}\cdot y_{2B}=y_{1B}\cdot x_{2B}$出现时,因为无法为 recursive_gcd 算法的接下来的计算提供可用的两组输入,所以只能返回一组非零值和一组零。在被调用时也会导致调用函数失败继续返回一组非零值和一组零,从而完成 recursive_gcd 算法的全部过程。

1.1.3　recursive_gcd 算法实例

沿用 1.1.2 节中符号,令$B=62\,865\,867\,041$,$C=57\,811\,773\,995$,$L(B)=36$,可表示为$(B,C,n)=(62\,865\,867\,041,57\,811\,773\,995,36)$,见表1实例1。

表 1 recursive_gcd 算法实例

实例	实例 1	实例 2
(B,C,n)	(62 865 867 041,57 811 773 995,36)	(48 755 054 703, 18 677 581 191,36)
(b,c,hn)	(65 825,109 099,19)	(465 007, 345 479,19)
$(x_{1b},y_{1b}),(x_{2b},y_{2b})$	(51,569), (919, -27)	(521, 161), (755, -773)
(b_1,c_1,hn)	(28 313 389,113 171 863,19)	(192 671, 229 923,19)
$(x_{1b_1},y_{1b_1}),(x_{2b_1},y_{2b_1})$	(121,131), (0,0)	(261, 937), (425, -483)
$(x_{1B},y_{1B}),(x_{2B},y_{2B})$	(246 949,61 775), (0,0)	(2 693 759, -2 939 225),(1 680 085, -1 425 011)

验证表 1 中 $(x_{1B},y_{1B}),(x_{2B},y_{2B})$ 对 B 和 C 作用结果的正确性如下：

实例 1：

$$62\,865\,867\,041 \times 246\,949 - 57\,811\,773\,995 \times 61\,775 = 11\,953\,340\,661\,366\,784 = 21\,743 \times 2^{39};$$

$$62\,865\,867\,041 \times 0 - 57\,811\,773\,995 \times 0 = 0$$

实例 2：

$$48\,755\,054\,703 \times 2\,693\,759 - 18\,677\,581\,191 \times (-2\,939\,225) = 186\,231\,980\,977\,815\,552 = 169\,377 \times 2^{40};$$

$$48\,755\,054\,703 \times 1\,680\,085 - 18\,677\,581\,191 \times (-1\,425\,011) = 108\,528\,394\,731\,257\,856 = 49\,353 \times 2^{41}$$

1.2 findxy_gcd 算法

考虑 recursive_gcd 算法返回一组辅助因子 x 和 y，另一组均为 0，也就是失败的情形出现时，如果该结果再被 recursive_gcd 算法调用则不能返回更优的辅助因子，那么此时由 findxy_gcd 算法来判断何时停止递归调用以提高算法效率。那么 findxy_gcd 算法不仅返回了最优的辅助因子，而且省掉了只有 recursive_gcd 算法时花费在失败递归上面的时间。

在 recursive_gcd 算法基础上，考虑寻找一组合适的 x、y 使得 $Bx - Cy$ 尽量多的低比特部分为 0。记 $(x,y) = \text{findxy\_gcd}(B, C, L(B))$，输入为 B、C 和 $L(B)$，$B > C$，输出为 (x,y)，且满足 $Val(Bx - Cy) \geqslant 2 \cdot \max\{L(x),L(y)\}$。类似于 recursive_gcd 算法可以容易得到 findxy_gcd 算法。

输入 op_1,op_2,n，其中 n 为 op_1 的比特长度，op_2 的比特长度也约为 n，这里 n 远大于算法中提到的小常数 $cons$。

输出 rop_1,rop_2，一般情形下满足 $Val(op_1 \times rop_1 - op_2 \times rop_2) \approx 2L(rop_1)$，也存在 $Val(op_1 \times rop_1 - op_2 \times rop_2) > 2L(rop_1)$。

1) $hn = cons$; //$cons$ 是某个小常数
2) $B \leftarrow op_1; C \leftarrow op_2; b = B \bmod 2^{hn}; c = C \bmod 2^{hn}$;
3) $(x_{1b},y_{1b},x_{2b},y_{2b}) = \text{recursive\_gcd}(b, c, hn)$;
4) if $x_{2b} == 0$,返回 (x_{1b},y_{1b});
5) while $hn < n/2$ do
6) $hn = 2 \cdot hn; b_1 = B \bmod 2^{hn}; c_1 = C \bmod 2^{hn}$;
7) $b = \text{abs}(b_1x_{1b} - c_1y_{1b}); c = \text{abs}(b_1x_{2b} - c_1y_{2b})$;
8) if $b == 0$ 或 $c == 0$,返回 (x_{1b},y_{1b});
9) $b_1 = b/2^{Val(b)}; c_1 = c/2^{Val(c)}$;
10) $(x_{1b_1},y_{1b_1},x_{2b_1},y_{2b_1}) = \text{recursive\_gcd}(b_1, c_1, hn)$;
11) if $Val(b) > Val(c)$ 且 $b_1x_{1b} - c_1y_{1b} > 0$ 且 $b_1x_{2b} - c_1y_{2b} > 0$,令：
11) $x_{1B} = x_{1b_1} \cdot x_{1b} - y_{1b_1} \cdot x_{2b} \cdot 2^{Val(b)-Val(c)}$,
$y_{1B} = x_{1b_1} \cdot y_{1b} - y_{1b_1} \cdot y_{2b} \cdot 2^{Val(b)-Val(c)}$,
12) $x_{2B} = x_{2b_1} \cdot x_{1b} - y_{2b_1} \cdot x_{2b} \cdot 2^{Val(b)-Val(c)}$,
$y_{2B} = x_{2b_1} \cdot y_{1b} - y_{2b_1} \cdot y_{2b} \cdot 2^{Val(b)-Val(c)}$;
13) 其他情况可类似得到 $x_{1B},y_{1B},x_{2B},y_{2B}$;
14) if $x_{1B} \cdot y_{2B} == x_{2B} \cdot y_{1B}$ 返回 (x_{1B},y_{1B});
15) $(x_{1b},y_{1b},x_{2b},y_{2b}) = (x_{1B},y_{1B},x_{2B},y_{2B})$;
16) end while
17) 返回 (x_{1b},y_{1b})。

对于 $(x,y) = \text{findxy\_gcd}(B, C, L(B))$，$L(B) - L((Bx - Cy)/2^{Val(Bx-Cy)})$ 能很好地反映该算法一次能消减的比特数。记 $Qmb(B,C,x,y) = L(B) - L((Bx - Cy)/2^{Val(Bx-Cy)})$，这里称 $Qmb(B,C,x,y)$ 为比特长度消减量。

对于 findxy_gcd 算法，随机选取 10 000 组长度为 1 000 比特左右的输入，统计 $Qmb(B,C,x,y)$ 在不同区间出现的频率，a_1 代表区间[10,20)，a_2 代表区间[20,30)，如此下去，a_{14} 代表区间[140,150)，而 a_{15} 代表区间[150,207)，得到表 2。

表 2 比特长度消减量区间及其对应频率

区间	频率	区间	频率
a_1	1 658	a_9	131
a_2	2 509	a_{10}	79
a_3	2 340	a_{11}	51
a_4	1 283	a_{12}	33
a_5	828	a_{13}	19
a_6	510	a_{14}	12
a_7	321	a_{15}	20
a_8	206		

由表 2 可知 $Qmb(B,C,x,y) \in [10,20)$ 的出现次数为 1 658，也就是在 findxy_gcd 算法的第 4) 步返回值的概率约为 0.165 8。另由表 2 得到比特长度消减量的平均值约为 40，也就是计算一次 $Bx - Cy$ 和一次右移可以消减输入的 40 个比特。虽然这里一次消减 40 比特相比于输入规模并不大，但是，由下文分析可以看到 Binary GCD 算法一个循环内只能消减不到 3 个比特，由此可以看出该算法相比 Binary GCD 算法在输入规模较大时降低循环次数还是非常明显的，并以此提高 Binary GCD 的速度。由表 2 还可以看出来 findxy_gcd 算法能返回的一组非零值大小与其出现的频率。比特长度消减量以 2.1% 的概率属于[100,206]，那就说明该算法返回的 x、y 比特长度大于 100 的概率约为 2.1%，这是因为一般只有当 x、y 比特长度大于 100 比特时，才能使得 $Val(Bx - Cy) \approx 2L(x) \geqslant 200$，从而使得 $Qmb(B,C,x,y) \geqslant 100$。

2 fast_gcd 算法

下面就可以利用 findxy_gcd 算法得到的 x、y 来进一步结合 Euclid 算法思想来消减输入的比特长度。对于求偶数的最大公约数，可以知道容易转化为求奇数的最大公约数，这里就不叙述。fast_gcd 算法，即 fast_gcd 算法描述如下：

输入　op_1,op_2,op_1 和 op_2 为正奇数;
输出　$\gcd(op_1,op_2)$。
1)　$B \leftarrow op_1$;$C \leftarrow op_2$;
2)　if $B > C$,令 $B \leftarrow B \bmod C$ 并 swap(B, C);
3)　else $C \leftarrow C \bmod B$;
4)　while $C \neq 0$ do
5)　　(x, y) = findxy_gcd(B, C,$L(B)$);
6)　　$B \leftarrow (B \cdot x - C \cdot y)/2^{Val(B \cdot x - C \cdot y)}$;　　// 这里导致误差
7)　　if $B \neq 0$,$C \leftarrow C \bmod B$; swap(B, C);
8)　　else swap(B, C);
9)　end while
10)　$rop \leftarrow B$;
11)　返回 $\gcd(rop,op_1,op_2)$。

3　fast_gcd 算法正确性及复杂度分析

任意正整数 n_1 和 n_2,一般情况下 $\gcd(n_1,n_2)$ 都会比较小,这是因为对于正整数 n_1、n_2[9]:

$$\Pr(\gcd(n_1,n_2) \leqslant n) = \frac{6}{\pi^2}\sum_{d=1}^{n}\frac{1}{d^2}$$

举个例子:$\gcd(n_1,n_2)$ 以近乎 99.9% 的概率小于1 000。

当 x 与 C 互素就不会引起误差,否则 fast_gcd 算法每一次调用 findxy_gcd 算法都会导致一定的误差,这是因为 $\gcd(Bx - Cy,C) \neq \gcd(B,C)$,误差为 $\gcd(x,C) = \gcd(Bx - Cy,C)/\gcd(B,C)$。虽然 fast_gcd 算法中含有误差项的结果(也就是 fast_gcd 算法第 10)步中的 rop)比 $\gcd(B,C)$ 大,但是,实验发现,fast_gcd 算法中 rop 的比特长度相比输入规模并不大。对于 100 万比特、90 万比特、80 万比特规模的随机输入,rop 的长度分别为 8 905 比特、8 340 比特、7 453 比特。

对于输入为 op_1、op_2,fast_gcd 算法通过求 $\gcd(op_1,op_2,rop)$ 来得到正确的最大公约数。由于 rop 长度相对较小,那么就可以两次利用 Euclid 算法通过求 $\gcd(op_1,rop)$ 和 $\gcd(op_2,\gcd(op_1,rop))$ 来得到 $\gcd(op_1,op_2,rop) = \gcd(op_1,op_2)$,这是由于 rop 能被 $\gcd(op_1,op_2)$ 整除,所以 $\gcd(op_1,op_2,rop) = \gcd(op_1,op_2)$。那么 fast_gcd 算法,就能够快速且准确地得到最大公约数。

一般情况下 findxy_gcd 算法返回值的大小在 100 比特内,则相对于 n 比特规模输入而言,findxy_gcd 算法花费时间可以看作常数 c_f。fast_gcd 算法花费的时间主要在该算法中的第 6)行、第 7)行和第 11)行,其中第 6)行有 4 个操作,包括 2 个乘法操作,一般情况为一个 100 比特的数乘以 n 比特的数,其复杂度为 $O(n)$,1 个减法操作和 1 个右移操作,复杂度为 $O(n)$,可见第 6)步复杂度为 $O(n)$;第 7)步采用的模算法虽然复杂度为 $O(n^2)$,但是这里采用模算法是为了加速对于 Binary GCD 算法中 1 次相减、1 次右移的操作,这是因为对于一定规模的输入,一次模算法约消减输入的 40 比特,而 Binary GCD 算法要消减 40 比特,则需要约 13 次相减和 13 次右移操作,实际实现时不如模算法花费时间少,而当输入规模达到一定程度导致模算法时间代价更高时则换用一次相减一次右移的方法。则第 7)行的复杂度也可以看作 $O(n)$。那么由循环次数约为 $n/40$,再结合第 11)行中算法的复杂度为 $O(n^2)$ 得到 fast_gcd 算法的复杂度为 $O(n^2)$。

假如每次调用 findxy_gcd 算法,输入输出表达为 (x,y) = findxy_gcd(B,C,$L(B)$),都满足 $L(B) \approx 2L(x)$。那么对于 n 比特规模的两个输入,findxy_gcd 算法所需时间,记为 $f(n)$,对于 n 比特规模的输入,可以得到 Schönhage-Strassen 乘法复杂度[13] 记为 $M(n) = O(n \log n \log\log n)$。为了便于分析复杂度,不妨设 $n = 2^m$,则由 findxy_gcd 算法知道:$f(2^m) = 2f(2^{m-1}) + cM(2^m)$($c$ 是常数),得到 $f(n) = O(n \log^2 n \log\log n)$。那么可以认为对于 n 比特规模的两个输入,findxy_gcd 算法所需时间即为 $f(n) = O(n \log^2 n \log\log n)$。对于 n 比特规模的两个输入,记 fast_gcd 算法的复杂度为 $F(n)$。由于每次消减输入比特规模特别大,使得循环次数特别小,导致 fast_gcd 算法的第 10)行中 rop 的值与正确的最大公约数相差不大,可以把第 11)行复杂度看作常数。由 fast_gcd 算法进行一次循环就降低一半的输入规模可以得到:$F(n) = F(n/2) + c_1 n \log^2 n \log\log n$($c_1$ 为常数),得到 $F(n) = O(n \log^2 n \log\log n)$,即当 findxy_gcd 算法的返回值都满足 $L(B) \approx 2L(x)$ 时,fast_gcd 算法的复杂度为 $O(n \log^2 n \log\log n)$。

4　fast_gcd 算法实现结果

在 Intel Core i5-3 470,主频为 3.2 GHz 的 CPU 的 PC 上,利用 GNU MP 4.1.2 版本大数包和 VC 6.0 分别实现 Binary GCD 和 fast_gcd 算法得出实验结果见图 1 和图 2。

图 1 里的 Binary GCD 算法的一次循环是指进行一次减法操作和一次右移操作,而 fast_gcd 算法的 1 次循环是指进行 1 次 findxy_gcd 算法的调用、2 次乘法操作、2 次减法操作和 1 次右移操作。

由图 1 看出 fast_gcd 算法进入循环的次数比 Binary GCD 算法要少,这也是 fast_gcd 算法比 Binary GCD 算法快的最主要原因。由表 2 知道 fast_gcd 算法一次循环调用一次 findxy_gcd 算法至少消减输入 10 比特,平均消减的比特数约为 40,再加上模算法消减的约 40 比特,那么一次循环平均能消减约 80 比特。而 Binary GCD 算法一次循环消减输入的比特数约为 3 比特,这一点可以从图 1 看出来,而且文献[14]指出对 lb N 比特的随机输入的循环次数的期望是 k lb N($N \to \infty$,$k \approx 0.706$)。那么可以看出 fast_gcd 算法明显少于 Binary GCD 算法进入循环的次数。但是由于一个循环内 fast_gcd 算法花费时间更多,所以对于小规模的输入 fast_gcd 算法并没有很大程度的提高。

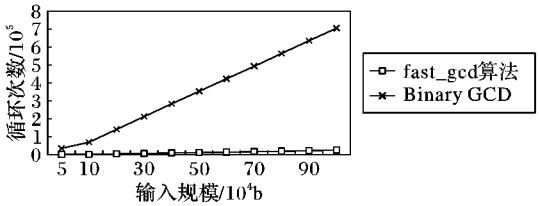

图 1　输入规模及其循环次数

由图 2 可以看出,当输入规模为 20 万比特时,两个算法所需时间一样,随着比特规模的增加,fast_gcd 算法的速度比 Binary GCD 算法越来越快。当输入规模达到 100 万比特规模时,fast_gcd 算法的速度是 Binary GCD 算法的两倍。

(下转第 1697 页)

Applications, 2010, 46(12): 79 - 81, 84. (徐小龙, 林亚平, 朱铁军, 等. 无线传感器网络中的一种正反馈信任模型[J]. 计算机工程与应用, 2010, 46(12): 79 - 81, 84.)

[5] PAN J, GAO J, XU Z, *et al*. nTRUST — A new trust scheme based on certainty theory in wireless sensor networks [J]. Chinese Journal of Sensors and Actautors, 2012, 25(2): 240 - 245. (潘巨龙, 高建桥, 徐展翼, 等. 一种基于确定性理论的无线传感器网络信任机制 nTRUST[J]. 传感技术学报, 2012, 25(2): 240 - 245.)

[6] CHEN J, DU R, WANG L, *et al*. A trust game method basing on probability model in networks [J]. Acta Electronica Sinica, 2010, 38(2): 427 - 433. (陈晶, 杜瑞颖, 王丽娜, 等. 网络环境下一种基于概率密度的信任博弈模型[J]. 电子学报, 2010, 38(2): 427 - 433.)

[7] LIU S, DANG Y, FANG Z, *et al*. Grey system theory and its application [M]. 5th ed. Beijing: Science Press, 2010: 53 - 78. (刘思峰, 党耀国, 方志耕, 等. 灰色系统理论及其应用[M]. 5 版. 北京: 科学出版社, 2010: 53 - 78.)

[8] CHEN W, OUYANG R, TANG G. A fine-grained trust model for P2P networks based on grey relation [J]. Computer Systems and Applications, 2010, 19(6): 153 - 157. (陈伟, 欧阳日, 汤光明. 一种细粒度的基于灰色关联度的 P2P 信任模型[J]. 计算机系统应用, 2010, 19(6): 153 - 157.)

[9] HE L, HUANG H. A distributed trust model based on grey system theory [J]. Journal of Beijing Jiaotong University, 2011, 35(3): 26 - 35. (贺利坚, 黄厚宽. 一种基于灰色系统理论的分布式信任模型[J]. 北京交通大学学报, 2011, 35(3): 26 - 35.)

[10] ZHOU H, YANG H, WU C. A power quality comprehensive evaluation method based on grey clustering [J]. Power System Protection and Control, 2012, 25(2): 240 - 245. (周辉, 杨洪耕, 吴传来. 基于灰色聚类的电能质量综合评估方法[J]. 电力系统保护与控制, 2012, 40(15): 70 - 75.)

[11] ZHOU C, LI X. Research on comprehensive evaluation of industrial wireless sensor network performance [J]. Computer Engineering, 2010, 36(16): 82 - 84, 87. (周婵, 李昕. 工业无线传感器网络性能综合评价研究[J]. 计算机工程, 2010, 36(16): 82 - 84, 87.)

[12] CHENG J, FENG R, XU X, *et al*. Trust evaluation model based on D-S evidence theory in wireless sensor networks [J]. Chinese Journal of Sensors and Actautors, 2009, 22(12): 1802 - 1807. (成坚, 冯仁剑, 许小丰, 等. 基于 D-S 证据理论的无线传感器网络信任评估模型[J]. 传感技术学报, 2009, 22(12): 1802 - 1807.)

[13] LIU S, XIE N. New grey evaluation method based on reformative triangular whitenization weight function [J]. Journal of System Engineering, 2011, 26(2): 244 - 250. (刘思峰, 谢乃明. 基于改进三角白化权函数的灰评估新方法[J]. 系统工程学报, 2011. 26(2): 244 - 250.)

[14] JING Q, TANG L, CHEN Z. Trust management in wireless sensor networks [J]. Journal of Software, 2008, 19(7): 1716 - 1730. (荆琦, 唐礼勇, 陈钟. 无线传感器网络中的信任管理[J]. 软件学报, 2008, 19(7): 1716 - 1730)

[15] LIU T, XIONG Y, HUANG W, *et al*. Trust computation model of nodes based on Bayes estimation in wireless sensor networks [J]. Computer Science, 2013, 40(10): 61 - 64. (刘涛, 熊焰, 黄文超, 等. 一种基于 Bayes 估计的 WSN 节点信任度计算模型[J]. 计算机科学, 2013, 40(10): 61 - 64.)

(上接第 1677 页)

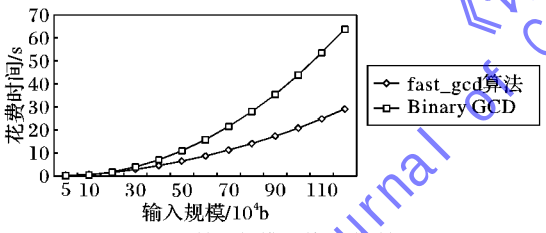

图 2　输入规模及其花费时间

5　结语

本文主要提出了寻找辅助因子 x 和 y 算法 findxy_gcd,并利用 findxy_gcd 算法对 Binary GCD 算法进行加速。由于 fast_gcd 算法的循环次数相比 Binary GCD 算法循环次数有大幅度的降低,所以使得 fast_gcd 算法速度更快。在以后的工作中考虑提高 recursive_gcd 算法返回两组非零值的概率以提高 $Qmb(B, C, x, y)$ 得到更大数值的概率,或者考虑解决对于一组返回值如何继续进行递归返回两组非零值的问题,这样在面对更大规模的输入时就可以更大幅度地提高 fast_gcd 算法速度。

参考文献:

[1] BUCHBERGER B, JEBELEAN T. Parallel rational arithmetic for computer algebra systems: motivating experiments [M]. Vienna: ACPC - Austrian Center for Parallel Computation, 1993: 1 - 3.

[2] LIOYD E K. The art of computer programming, vol. 2, seminumerical algorithms[J]. Software: Practice and Experience, 1982, 12(9): 883 - 884.

[3] STEIN J. Computational problems associated with Racah algebra [J]. Journal of Computational Physics, 1967, 1(3): 397 - 405.

[4] BRENT R P, KUNG H T. Systolic VLSI arrays for polynomial GCD computation [J]. IEEE Transactions on Computers, 1984, C-33(8): 731 - 736.

[5] YAP C K. Fundamental problems in algorithmic algebra [M]. Oxford: Oxford University Press, 2000: 43 - 76.

[6] STEHLÉ D, ZIMMERMANN P. A binary recursive gcd algorithm [C]// Proceedings of the 2004 6th International Symposium on Algorithmic Number Theory, LNCS 3076. Berlin: Springer, 2004: 411 - 425.

[7] SORENSON J. Two fast GCD algorithms [J]. Journal of Algorithms, 1994, 16(1): 110 - 144.

[8] JEBELEAN T. A generalization of the binary GCD algorithm [C]// Proceedings of the 1993 International Symposium on Symbolic and Algebraic Computation. New York: ACM, 1993: 111 - 116.

[9] WEBER K. The accelerated integer GCD algorithm [J]. ACM Transactions on Mathematical Software, 1995, 21(1): 111 - 122.

[10] SEDJELMACI S M. The mixed binary Euclid algorithm [J]. Electronic Notes in Discrete Mathematics, 2009, 35: 169 - 176.

[11] MOHAMED F K. A novel fast hybrid GCD computation algorithm [J]. International Journal of Computing Science and Mathematics, 2014, 5(1): 37 - 47.

[12] COHN H. Advanced number theory [M]. New York: Courier Dover Publications, 2012: 55 - 56.

[13] SCHÖNHAGE A, STRASSEN V. Schnelle multiplikation grosser zahlen [J]. Computing, 1971, 7(3/4): 281 - 292.

[14] BRENT R P. Analysis of the binary Euclidean algorithm [EB/OL]. [2014 - 12 - 02]. http://maths-people.anu.edu.au/~brent/pd/rpb037a.pdf.