acm PRESS

SIGSAM Bulletin

A Quarterly Publication of the
Special Interest Group on Symbolic & Algebraic Manipulation

Volume 27, Number 1 January, 1993 Issue Number 103

Research Articles

Departments

A Binary Algorithm for the Jacobi Symbol

Jeffrey Shallit*
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
shallit@graceland.uwaterloo.ca

Jonathan Sorenson†
Department of Mathematics and Computer Science
Butler University
4600 Sunset Avenue
Indianapolis, IN 46208 USA
sorenson@butleru.bitnet

September 14, 1992

Abstract

We present a new algorithm to compute the Jacobi symbol, based on Stein's binary algorithm for the greatest common divisor, and we determine the worst-case behavior of this algorithm. Our implementation of the algorithm runs approximately 7–25% faster than traditional methods on inputs of size 100–1000 decimal digits.

1 Introduction

Efficient computation of the Jacobi symbol $\left(\frac{a}{n}\right)$ is an important component of the Monte Carlo primality test of Solovay and Strassen [9]. Algorithms for computing the Jacobi symbol can also be found on symbolic algebra systems such as Mathematica and Maple.

Several efficient algorithms modeled on Euclid's algorithm for computing the greatest common divisor (gcd) have been proposed and analyzed; see, for example, [12, 3, 8]. Indeed,

*Supported in part by a grant from NSERC.
†Supported in part by a Butler University Fellowship and a grant from NSF.

it is possible to compute $\left(\frac{a}{n}\right)$ in $O((\log a)(\log n))$ bit operations using the "naive arithmetic" model. Using Schönhage's result [7], it is possible (see [1]) to compute $\left(\frac{a}{n}\right)$ (for $0 < a < n$) using $O((\log n)(\log\log n)^2(\log\log\log n))$ bit operations, but this is not likely to be useful in practice.

Stein [11] proposed a "binary" algorithm for computing $\gcd(u, v)$; this algorithm is based on the binary expansion of the numbers, and performs no divisions (other than divisions by 2). It has a particularly efficient implementation on binary digital computers, where division by 2 can be done quickly by using a "shift". Although it is not surprising that it is possible to adapt Stein's algorithm to compute the Jacobi symbol, it does not seem to have been remarked before that this method is actually simpler and more efficient in practice than familiar methods. We have implemented this method in C++, and our results show that the new algorithm runs approximately 7–25% faster than the traditional methods.

This paper is organized as follows: first, we present Stein's binary algorithm and determine its worst-case behavior. Next, we present pseudo-code for two familiar algorithms for computing the Jacobi symbol, and our new binary Jacobi symbol algorithm. We determine the worst case of the binary Jacobi symbol algorithm. Finally, we present our timing results.

2 Stein's binary gcd algorithm

We first present Stein's binary gcd algorithm:

```
Binary Gcd(u,v)   /* inputs are integers u, v > 0 */
g := 1;

while (u mod 2 = 0) and (v mod 2 = 0) do
     u := u/2;
     v := v/2;
     g := 2*g;

while (u <> 0) do
     if (u mod 2 = 0) then u := u/2
     else if (v mod 2 = 0) then v := v/2
     else
(1)       t := abs((u-v)/2);
          if (u >= v) then u := t else v := t;

return(g*v).
```

In an environment which supports bit-shifting operations, division and multiplication by 2 are relatively inexpensive operations, compared to step (1). We call an execution of step (1) a *subtraction step*.

We now determine the worst case of this algorithm, that is, the lexicographically least pair (u,v) that forces the algorithm to perform n subtraction steps. (We say a pair (u,v) is *lexicographically less* than a pair (u',v') if $u < u'$, or if $u = u'$ and $v < v'$.)

Lemma 1 *Let $R(u,v)$ denote the number of subtraction steps performed by* `Binary Gcd` *on input (u,v). Then $R(u,v) \leq \lfloor \log_2(u+v) \rfloor$.*

Proof.

It suffices to prove that $R(u,v) \leq \log_2(u+v)$; from this the result follows, since $R(u,v)$ is an integer. The proof is by induction on $u+v$. It is clearly true if $u = v = 1$, for then the algorithm performs one subtraction step. Otherwise, we proceed by induction. If both u and v are even, then $R(u,v) = R(u/2, v/2) \leq \log_2(u/2 + v/2) \leq \log_2(u+v)$. A similar inequality holds when u is even and v is odd, and when u is odd and v is even.

Finally, assume both u and v are odd. Then if $u \geq v$, the algorithm performs a subtraction step and replaces u with $(u-v)/2$. Hence we have

$$R(u,v) = 1 + R((u-v)/2, v) \leq 1 + \log_2((u-v)/2 + v) = 1 + \log_2((u+v)/2) = \log_2(u+v).$$

The same thing occurs if $u < v$. ■

We note that a similar, though weaker, result has been given by D. E. Knuth [5, Exercise 4.5.2.28]. For heuristic average-case analysis of the binary algorithm, see Brent [2].

Theorem 2 *Let $u > v > 0$, and let (u,v) be the lexicographically least pair such that Stein's binary gcd algorithm performs n subtraction steps on input (u,v). Then $u = 2^{n-1} + 1$, $v = 2^{n-1} - 1$ for $n \geq 2$.*

Proof.

Observe that $u \geq 2^{n-1}$, for otherwise by the lemma we have $R(u,v) < n$. But u cannot equal 2^{n-1}, for then the first step performed is to divide u by 2, and subsequent processing cannot result in more than $n-1$ subtraction steps. Thus $u \geq 2^{n-1} + 1$. It is easy to see that $R(2^{n-1}+1, 2^{n-1}-1) = n$ for $n \geq 2$. It remains to see that if $n \geq 3$, then $R(2^{n-1}+1, v) < n$ for all v with $1 \leq v < 2^{n-1} - 1$. It suffices to consider the case where v is odd, for when v is even, the next step of the algorithm changes v to $v/2$. Hence assume v is odd, and $1 \leq v < 2^{n-1} - 1$. After one subtraction step we are left with the pair $((2^{n-1} + 1 - v)/2, v)$, and by the lemma above we have $R((2^{n-1} + 1 - v)/2, v) \leq \lfloor \log_2(2^{n-1} - 1) \rfloor \leq n - 2$. Hence $R(2^{n-1} + 1, v) \leq n - 1$. ■

3 Three algorithms for computing the Jacobi symbol

In this section, we present three algorithms for computing the Jacobi symbol $\left(\frac{a}{n}\right)$. The first, which we call the "ordinary" algorithm, is essentially that given by Williams [12]. It has been analyzed by Collins and Loos [3] and Shallit [8].

We write the algorithms in a Pascal-like pseudocode. We do not use **begin/ends**, however, preferring to let the scope of the loops be specified by indentation.

All three algorithms assume that the inputs a, n are integers with n positive and odd. The first and third algorithms also assume that a is non-negative. The correctness of the algorithms follows from the following five identities, where m and n are positive odd integers:

$$\left(\frac{0}{n}\right) = \begin{cases} 1, & \text{if } n = 1; \\ 0, & \text{otherwise.} \end{cases}$$

$$\left(\frac{-1}{n}\right) = \begin{cases} 1, & \text{if } n \equiv 1 \pmod 4; \\ -1, & \text{if } n \equiv 3 \pmod 4 . \end{cases}$$

$$\left(\frac{2}{n}\right) = \begin{cases} 1, & \text{if } n \equiv 1, 7 \pmod 8; \\ -1, & \text{if } n \equiv 3, 5 \pmod 8. \end{cases}$$

$$\left(\frac{m}{n}\right) = \begin{cases} -\left(\frac{n}{m}\right), & \text{if } n, m \equiv 3 \pmod 4; \\ \left(\frac{n}{m}\right), & \text{otherwise.} \end{cases}$$

$$\left(\frac{m \bmod n}{n}\right) = \left(\frac{m}{n}\right).$$

The ordinary Jacobi symbol algorithm is similar to Euclid's algorithm for computing the gcd, except that powers of 2 are removed when possible:

```
Jacobi(a,n)  /* a >= 0; n > 0; n odd */

t := 1;
while (a <> 0) do

     while (a mod 2 = 0) do
          a := a/2;
          if (n mod 8 = 3) or (n mod 8 = 5) then t := -t;

     interchange(a, n);
     if (a mod 4 = 3) and (n mod 4 = 3) then t := -t;
     a := a mod n;

if (n = 1) then return(t) else return(0).
```

The next algorithm we present is due to Lebesgue [6]. It is similar to the least-remainder algorithm for computing the gcd.

```
LR Jacobi(a,n)  /* n > 0; n odd */

t := 1;
while (a <> 0) do

     if (a < 0) then
          a := -a;
          if (n mod 4 = 3) then t := -t;

     while (a mod 2 = 0) do
          a := a/2;
          if (n mod 8 = 3) or (n mod 8 = 5) then t := -t;

     interchange(a, n);
     if (a mod 4 = 3) and (n mod 4 = 3) then t := -t;
     a := a mod n;
     if (a > n/2) then a := a-n;

if (n = 1) then return(t) else return(0).
```

Finally, we present our new algorithm, based on Stein's binary algorithm discussed above:

```
Binary Jacobi(a,n);   /* a >= 0; n > 0; n odd */

t := 1;
while (a <> 0) do

     while (a mod 2 = 0) do
          a := a/2;
          if (n mod 8 = 3) or (n mod 8 = 5) then t := -t;

      if (a < n) then
          interchange(a,n);
          if (a mod 4 = 3) and (n mod 4 = 3) then t := -t;

(*)  a := (a-n)/2;
     if (n mod 8 = 3) or (n mod 8 = 5) then t := -t;

if (n = 1) then return(t) else return(0).
```

We call an execution of step (*) a *subtraction step*. Because of the similarity of this algorithm to Stein's binary gcd algorithm, it is easy to determine its worst-case behavior:

Theorem 3 *Let (a,n) (with $n > a > 0$, n odd) be the lexicographically least pair such that* `Binary Jacobi`(a,n) *performs m subtraction steps. Then $a = 2^m - 1$, $n = 2^m + 1$.*

Proof.
This follows immediately from Theorem 2. ■

4 Experimental Results

In this section, we present our experimental results. We programmed our algorithms in `C++`, using a package developed by the second author at Butler University.

We timed our algorithms on two different platforms; the source code was identical on each machine except for the system call used to obtain the current time. In addition, on each platform, we implemented the `interchange( , )` procedure in two different ways: by interchanging pointers and by interchanging values. These four sets of timing results are given below.

Average running times in CPU seconds on a CompuAdd 486/33 PC running MS-DOS 5.0 at Butler University are as follows:

Interchanging Pointers

Decimal Digits	`Jacobi`	`LR Jacobi`	`Binary Jacobi`
100	0.0405	0.0429	0.0413
250	0.1910	0.1838	0.1727
500	0.6610	0.6510	0.5745
1000	2.416	2.463	2.079

Interchanging Values

Decimal Digits	`Jacobi`	`LR Jacobi`	`Binary Jacobi`
100	0.0458	0.0461	0.0445
250	0.2103	0.2058	0.1907
500	0.7300	0.7360	0.6350
1000	2.675	2.780	2.307

Average running times in CPU seconds on a DECstation 5500 running Ultrix at the University of Waterloo are as follows:

Interchanging Pointers

Decimal Digits	`Jacobi`	`LR Jacobi`	`Binary Jacobi`
100	0.0183	0.0190	0.0170
250	0.0865	0.0900	0.0720
500	0.3060	0.3275	0.2385
1000	1.162	1.239	0.864

Interchanging Values

Decimal Digits	`Jacobi`	`LR Jacobi`	`Binary Jacobi`
100	0.0215	0.0220	0.0197
250	0.1023	0.1080	0.0870
500	0.3645	0.3960	0.3000
1000	1.387	1.503	1.115

Implementation notes:

- All three algorithms were given the same set of 10 input pairs for each input size. For each input pair, each algorithm was run multiple times. The time listed in the table is the overall average.
- All divisions by 2 were implemented using bit shifts, and the `mod 4` and `mod 8` operations were done by examining the least significant bits of the numbers (i.e. no divisions were performed).
- In our experiments, we found that the `LR Jacobi` algorithm performed the fewest iterations of its main loop (about 980 for 1000 digit inputs). The `Jacobi` algorithm performed roughly 20% more iterations. The `Binary Jacobi` algorithm performed almost twice as many iterations as the `Jacobi` algorithm.

5 Additional Remarks

Although we did not do this, it would be easy to modify our algorithm to compute the generalized Jacobi symbol called the *Kronecker symbol* (see, e.g., Hua [4].)

There is a "k-ary" Jacobi symbol algorithm along the lines of the k-ary GCD algorithm [10], which is likely to be efficient in practice.

References

[1] E. Bach and J. O. Shallit. *Algorithmic Number Theory.* The MIT Press, Cambridge, Mass., 1992. To be published.

[2] R. P. Brent. Analysis of the binary Euclidean algorithm. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 321–355. Academic Press, New York, 1976.

[3] G. E. Collins and R. G. K. Loos. The Jacobi symbol algorithm. *ACM SIGSAM Bull.*, 16(1):12–16, 1982.

[4] L. K. Hua. *Introduction to Number Theory.* Springer-Verlag, New York, 1982.

[5] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms.* Addison-Wesley, 1981. 2nd edition.

[6] V.-A. Lebesgue. Sur le symbole $(\frac{a}{b})$ et quelques-unes de ses applications. *J. Math. Pures Appl.*, 12:497–517, 1847.

[7] A. Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Infor.*, 1:139–144, 1971.

[8] J. O. Shallit. On the worst case of three algorithms for computing the Jacobi symbol. *J. Symbolic Comput.*, 10:593–610, 1990.

[9] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM J. Comput.*, 6:84–85, 1977. Also see erratum 7:118, 1978.

[10] J. Sorenson. *Algorithms in Number Theory.* PhD thesis, University of Wisconsin-Madison, June 1991. Available as Computer Sciences Technical Report #1027.

[11] J. Stein. Computational problems associated with Racah algebra. *J. Comput. Phys.*, 1:397–405, 1967.

[12] H. C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Trans. Inform. Theory*, IT-26:726–729, 1980.