An $O(M(n) \log n)$ algorithm for the Jacobi symbol

Richard P. Brent, ANU

Paul Zimmermann, INRIA, Nancy

6 December 2010

Definitions

The *Legendre symbol* $(a|p)$ is defined for $a, p \in \mathbb{Z}$, where p is an *odd prime*.

$$(a|p) = \begin{cases} 0 \text{ if } a = 0 \bmod p, \text{ else} \\ +1 \text{ if } a \text{ is a quadratic residue (mod } p), \\ -1 \text{ otherwise.} \end{cases}$$

By Euler's criterion, $(a|p) = a^{(p-1)/2} \bmod p$.

The *Jacobi symbol* $(a|n)$ is a generalisation where n does not have to be prime (but must still be odd and positive):

$$(a|p_1^{\alpha_1} \cdots p_k^{\alpha_k}) = (a|p_1)^{\alpha_1} \cdots (a|p_k)^{\alpha_k}$$

This talk is about an algorithm for computing $(a|n)$ quickly, without needing to factor n.

Connection with the GCD

The greatest common divisor (GCD) of two integers b, a (not both zero) can be computed by (many different variants of) the Euclidean algorithm, using the facts that:

$$\gcd(b, a) = \gcd(b \bmod a, a),$$

$$\gcd(b, a) = \gcd(a, b),\ \gcd(a, 0) = a.$$

Identities satisfied by the Jacobi symbol $(b \mid a)$ are similar:

$$(b \mid a) = (b \bmod a \mid a),$$

$$(b \mid a) = (a \mid b)(-1)^{(a-1)(b-1)/4} \text{ for } b \text{ odd positive},$$

$$(-1 \mid a) = (-1)^{(a-1)/2},\ (2 \mid a) = (-1)^{(a^2-1)/8},$$

$$(b \mid a) = 0 \text{ if } \gcd(a, b) \neq 1.$$

Computing the Jacobi symbol

The similarity of the identities satisfied by $\gcd(b, a)$ and the Jacobi symbol $(b \mid a)$ suggest that we could compute $(b \mid a)$ while computing $\gcd(b, a)$, just keeping track of the sign changes and making sure that everything is well-defined.

This is true, if the GCD is computed via the classical Euclidean algorithm (or via the binary Euclidean algorithm), and leads to a quadratic algorithm for computing the Jacobi symbol.

For large inputs, the GCD can be computed even faster, as first shown by Knuth (1970) and Schönhage (1971). Can we speed up computation of the Jacobi symbol as well?

Complexity of algorithms

Assume the inputs are n-bit integers.

A cubic algorithm runs in time $O(n^3)$.

A quadratic algorithm runs in time $O(n^2)$.

A subquadratic algorithm runs in time $o(n^2)$.

All subquadratic algorithms considered in this talk run in time $O(M(n) \log n)$, where

$$M(n) = O(n \log n \log \log n)$$

is the time required to multiply n-bit integers.

Motivation

From: Steven Galbraith
Date: 17 April 2009
To: Paul Zimmermann, ...

Hi Paul, ...

The usual algorithm to compute the Legendre (or Jacobi) symbol is closely related to Euclid's algorithm. There are variants of Euclid for n-bit integers which run in O(M(n) log(n)) bit operations. Hence it is natural to expect a O(M(n) log(n)) algorithm for Legendre symbols.

I don't see this statement anywhere in the literature. Is this:

(a) in the literature somewhere
(b) so obvious no-one ever wrote it down
(c) false due to some subtle reason.

Thanks for your help.

Regards
Steven

Answer: (b) so obvious no-one ever wrote it down (?)

This is what we first thought.

However we soon realized it was not so easy...

Potential difficulty

Known fast (subquadratic) GCD algorithms work in the following way. A recursive procedure `halfGCD`(a, b) returns a matrix R such that, if

$$\begin{pmatrix} a' \\ b' \end{pmatrix} = R \begin{pmatrix} a \\ b \end{pmatrix},$$

where $\max(|a'|, |b'|)$ is significantly smaller than $\max(|a|, |b|)$, but the GCD is preserved, i.e. $\gcd(a', b') = \gcd(a, b)$.

In `halfGCD`(a, b) we (usually) work with the most significant bits of a and b. This means that we might not have all the information required to update the Jacobi symbol, which depends on the least significant bits.

Examples on some computer algebra systems

Magma V2.16-10 on 2.83Ghz Core 2:

```
> a:=3^209590; b:=5^143067;
> time c := Gcd(a,b);
Time: 0.080
> time d := JacobiSymbol(a,b);
Time: 2.390
```

Sage 4.4.4 on 2.83Ghz Core 2:

```
sage: a=3^209590; b=5^143067
sage: a.ndigits(), b.ndigits()
(100000, 100000)
sage: %timeit a.gcd(b)
5 loops, best of 3: 49.9 ms per loop
sage: %timeit a.jacobi(b)
5 loops, best of 3: 2.04 s per loop
```

GMP 5.0.1 and GP/PARI 2.4.3 give similar results.

Answer: (a) in the literature somewhere (?)

Yes and no. The literature is incomplete and confusing.

There are two MSB (most significant bits first) algorithms:

- Bach and Shallit, "Algorithmic Number Theory" (1996), solution of Exercise 5.52 [sketch only, attributed to Gauss (1817/18), Bachmann (1902), Schönhage (1971)], also mentioned briefly in Bach (1990);
- a *different* algorithm mentioned by Schönhage in his "Turing machine" book (1994), but without details. This algorithm does not use the identity of Gauss.

As far as we know, no subquadratic implementation exists, except that of Schönhage in the TP language, which shows how to implement it on a multi-tape Turing machine, but is not immediately relevant to Maple, Magma, Sage, etc.

Answer: (c) false due to some subtle reason (?)

No, it is possible, although nontrivial.

It can be done using a fast version of either:

- a “most significant bit first” (MSB) Euclidean algorithm, e.g. Schönhage/Möller,
- or a “least significant bit first” (LSB) algorithm, e.g. Stehlé and Zimmermann (2004).

The LSB algorithm is simpler and easier to justify.

It does not seem possible to adapt Shallit and Sorenson’s quadratic “binary” algorithm (1993) to give a subquadratic Jacobi algorithm.

Outline of remainder of the talk

- Binary (MSB and LSB) division for GCD computation
- A cubic (quadratic?) LSB algorithm for the Jacobi symbol
- A provably quadratic LSB algorithm
- A subquadratic LSB algorithm (details omitted)
- Implementation and timings
- Annotated list of references

LSB algorithms

We propose an LSB (least significant bit) algorithm, that can be implemented with time bound $O(M(n)\log n)$ by modifying an LSB gcd algorithm.

We assume a is odd positive, b is even positive.

- if b is negative, use $(b|a) = (-1)^{(a-1)/2}(-b|a)$.
- if b is odd, use $(b|a) = (b+a|a)$.

For $a \in \mathbb{Z}$, the notation $\nu(a)$ denotes the 2-adic valuation $\nu_2(a)$ of a, that is the maximum k such that $2^k|a$, or $+\infty$ if $a = 0$.

(LSB) Binary division

$$a = 935 = (1110100111)_2$$

$$b = 714 = (1011001010)_2$$

- divide b by the largest possible power of two:

$$b/2 = 357 = (101100101)_2$$

- now choose in $[a + b/2, a - b/2]$ the number $a + qb/2$ with most trailing zeros:

$$a + b/2 = 1292 = (10100001100)_2$$

$$a - b/2 = 578 = (1001000010)_2$$

Reference: Stehlé and Zimmermann, *A binary recursive gcd algorithm*, Proc. ANTS VI, 2004.

(LSB) Binary division: another example

$$a = 935 = (1110100111)_2$$

$$b = 716 = (1011001100)_2$$

$$a + b/4 = 1114 = (10001011010)_2$$

$$a - b/4 = 756 = (1011110100)_2$$

$$a + 3b/4 = 1472 = (10111000000)_2$$

$$a - 3b/4 = 398 = (110001110)_2$$

Here we choose $a + 3b/4$ as next term.

Theory of LSB binary division

Suppose $a, b \in \mathbb{Z}$ with $j := \nu(b) - \nu(a) > 0$.

There is a unique $q \in (-2^j, 2^j)$ such that

$$r = a + qb/2^j \text{ and } \nu(r) > \nu(b).$$

q is the *binary quotient* of a by b.
r is the *binary remainder* of a by b.

Rationale: if a, b each have n bits, $b' = b/2^j$ has $n - j$ bits, and qb' has about n bits, thus r has about the same bit-size as a, but at least $j + 1$ more zeros in the LSBs.

Also, $\gcd(b, r) = \gcd(a, b)$ (as for MSB binary division).

Computation

$$j = \nu(b) - \nu(a) > 0$$

$$b' = b/2^j$$

$$q \equiv -a/b' \bmod 2^{j+1} \quad \text{(centered)}$$

Iterating, we get a binary remainder sequence $a, b, r, \ldots$ with

$$\nu(a) < \nu(b) < \nu(r) < \cdots$$

Using binary (LSB) division for GCD computation

Binary (LSB) division forces 0's in the LSBs:

$$
\begin{array}{rrr}
 & 935 & 1110100111 \\
 & 714 & 1011001010 \\
935 + 714/2 = & 1292 & 10100001100 \\
714 + 2 \times 1292/2^2 = & 1360 & 10101010000 \\
1292 + 4 \times 1360/2^4 = & 1632 & 11001100000 \\
1360 + 16 \times 1632/2^5 = & 2176 & 100010000000 \\
1632 - 96 \times 2176/2^7 = & 0 & 000000000000
\end{array}
$$

Conclusion: $\gcd(935, 714) = (10001)_2 = 17 = 2176/2^7$

Comparison – using MSB division

Classical (MSB) division forces zeros in the MSBs:

decimal	*binary*
935	1110100111
714	1011001010
$835 - 714 = 221$	0011011101
$714 - 3 \times 221 = 51$	0000110011
$221 - 4 \times 51 = 17$	0000010001
$51 - 3 \times 17 = 0$	0000000000

Conclusion: $\gcd(935, 714) = (10001)_2 = 17$

Advantages of LSB binary division

- ⊕ simpler to compute, at least in software (division mod 2^{j+1} instead of MSB division);
- ⊕ no "repair step" in the subquadratic GCD;
- ⊕ an average reduction of two LSB bits per iteration;
- ⊖ an average increase of 0.05 MSB bit per iteration (analyzed precisely by Daireaux, Maume-Deschamps and Vallée, DMTCS, 2005).

Using binary (LSB) division for the Jacobi symbol (?)

It seems easy, using $b' = b/2^j$ odd, via the identities:

$$(b|a) = (-1)^{j(a^2-1)/8}(b'|a)$$

$$(b'|a) = (-1)^{(a-1)(b'-1)/4}(a|b')$$

$$(a|b') = (a + qb'|b') = (r|b')$$

$$(r|b') = (-1)^{j(b'^2-1)/8}(r/2^j|b')$$

However r can be negative!

Example: 935, 738, 1304, -240, 1184, -832, 768, -1024, 0.

This is incompatible with the definition of the Jacobi symbol, which requires a odd positive.

Binary (LSB) division with positive quotient

Solution: use positive instead of centred quotient – instead of taking $q = a/(b/2^j) \mod 2^{j+1}$ in $(-2^j, 2^j)$, take it in $(0, 2^{j+1})$.

Since $q > 0$, if $a, b > 0$, then $r = a + qb/2^j > 0$, so all terms in the binary remainder sequence are non-negative.

Stopping GCD criterion: $a/2^{\nu(a)} = b/2^{\nu(b)}$.

Notation: $(q, r) = \text{BinaryDividePos}(a, b)$.

Example: $935, 714 = 357 \cdot 2, 1292 = 323 \cdot 2^2, 1360 = 85 \cdot 2^4, 1632 = 51 \cdot 2^5, 2176 = 17 \cdot 2^7, 4352 = 17 \cdot 2^8$.

Disadvantage: Slower convergence, compared to using the centred quotient.

A cubic (quadratic?) LSB algorithm

Algorithm CubicBinaryJacobi.

Input: $a, b \in \mathbb{N}$ with $\nu(a) = 0 < \nu(b)$

Output: Jacobi symbol $(b|a)$

1: $s \leftarrow 0$
2: $j \leftarrow \nu(b)$
3: **while** $2^j a \neq b$ **do**
4: $\quad b' \leftarrow b/2^j$
5: $\quad (q, r) \leftarrow \mathrm{BinaryDividePos}(a, b)$
6: $\quad s \leftarrow (s + \frac{j(a^2-1)}{8} + \frac{(a-1)(b'-1)}{4} + \frac{j(b'^2-1)}{8}) \bmod 2$
7: $\quad (a, b) \leftarrow (b', r/2^j)$
8: $\quad j \leftarrow \nu(b)$
9: **if** $a = 1$ **then** return $(-1)^s$ **else** return 0

(lines in red are added to the LSB GCD-algorithm)

Cost of the cubic (quadratic?) algorithm

Let n be the bit-size of the inputs a, b.

Each iteration costs $O(n)$.

The number of iterations is $O(n^2)$ (conjectured to be $O(n)$).

Thus the total cost is $O(n^3)$ (conjectured to be $O(n^2)$).

A provably quadratic LSB algorithm

Lemma

The quantity $a + 2b$ is non-increasing in CubicBinaryJacobi.

Proof. At each iteration, $a + 2b$ becomes:

$$\frac{2a}{2^j} + \left(1 + \frac{2q}{2^j}\right)\frac{b}{2^j}.$$

If $j \geq 2$, $a + 2b$ is multiplied by a factor at most $9/16$:
call this a *good* iteration.

If $j = 1$ and $q = 1$, $a + 2b$ decreases, but with a factor that can be arbitrarily close to 1: *bad* iteration.

If $j = 1$ and $q = 3$, $a + 2b$ remains unchanged: *ugly* iteration.
(Ugly iterations never occur with centred LSB division.)

Examples

Good iteration: $a = 9, b = 4$ gives $j = 2, q = 7$, $b' = 1, r/2^j = 4$, $a + 2b = 17$ becomes 9.

Bad iteration: $a = 9, b = 6$ gives $b' = 3, r/2^j = 6$, $a + 2b = 21$ becomes 15.

Ugly iteration: $a = 9, b = 10$ gives $b' = 5, r/2^j = 12$, $a + 2b = 29$ remains 29.

Sequences of ugly iterations

Lemma

If $\mu = \nu(a - b/2)$, there are exactly $\lfloor \mu/2 \rfloor$ ugly iterations starting from (a, b), followed by a good iteration if μ is even, otherwise by a bad iteration.

Example 1: $a - b/2 = 64 = 2^6$

$$(85, 42) \underbrace{\to}_{ugly} (21, 74) \underbrace{\to}_{ugly} (37, 66) \underbrace{\to}_{ugly} (33, 68) \underbrace{\to}_{good} (34, 38) \cdots$$

Example 2: $a - b/2 = 128 = 2^7$

$$(149, 42) \underbrace{\to}_{ugly} (21, 106) \underbrace{\to}_{ugly} (53, 90) \underbrace{\to}_{ugly} (45, 94) \underbrace{\to}_{bad} (47, 46) \cdots$$

Corollary and conjecture

Corollary

The worst-case running time of Algorithm CubicBinaryJacobi for n-bit inputs is $O(n^3)$.

Conjecture.

The worst-case running time of Algorithm CubicBinaryJacobi on n-bit inputs is $O(n^2)$.

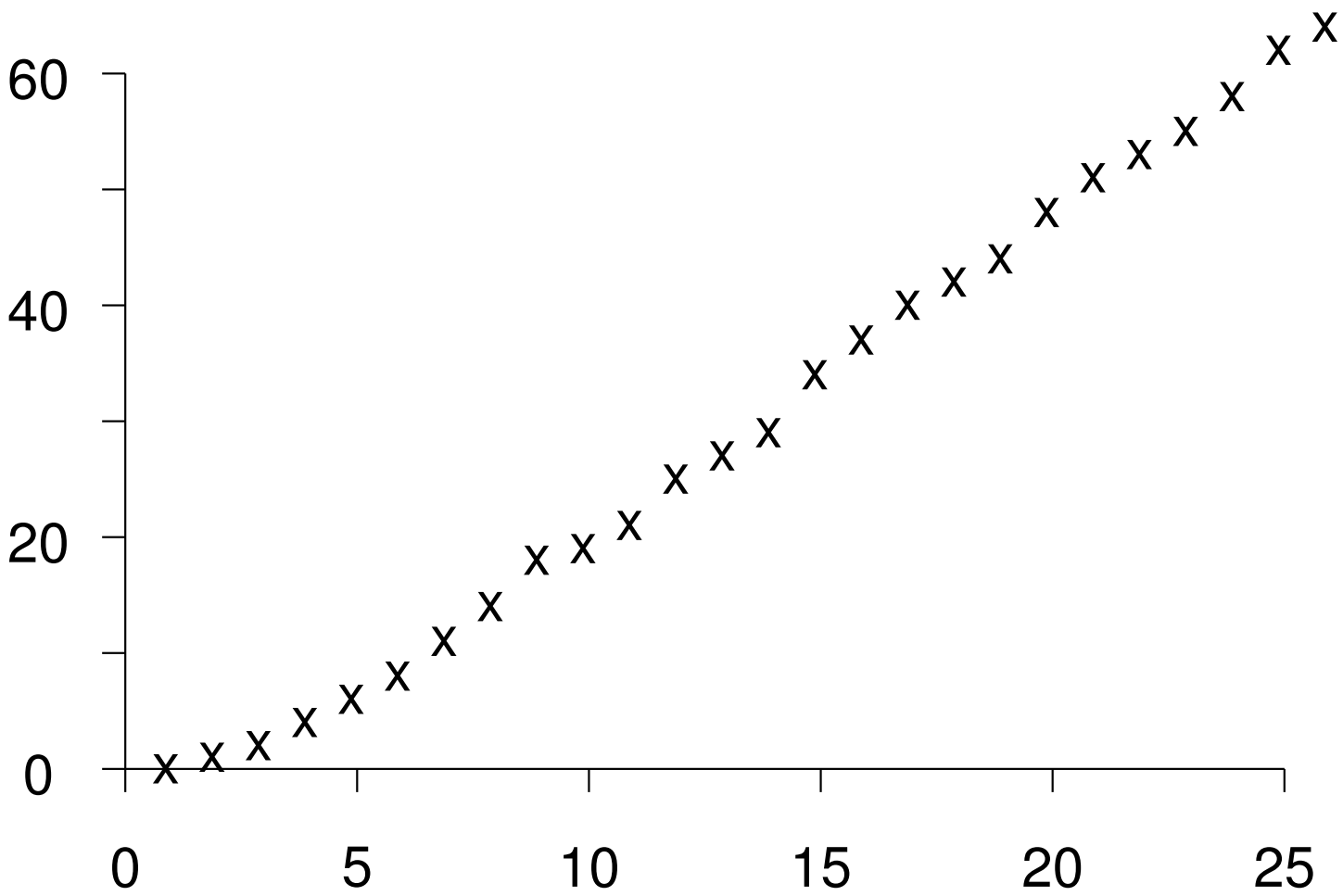
Evidence.

The worst-case number of iterations for Algorithm CubicBinaryJacobi on n-bit inputs is as follows:

n	5	10	15	20	25	26
iterations	6	19	34	48	62	64

This seems to be linear in n (implying quadratic running time).

The evidence – max iterations vs bit-size



A quadratic LSB algorithm

Main idea: from the 2-valuation of $a - b/2$, compute the number $m > 0$ of consecutive ugly iterations, and apply them all at once: call this a *harmless* iteration.

The Jacobi symbol can be updated efficiently for a harmless iteration (details omitted).

Now we have only good (G), bad (B), or harmless (H) iterations, where HH is forbidden.

Algorithm QuadraticBinaryJacobi

1: $s \leftarrow 0, \quad j \leftarrow \nu(b), \quad b' \leftarrow b/2^j$
2: **while** $a \neq b'$ **do**
3: $\quad s \leftarrow (s + j(a^2 - 1)/8) \bmod 2$
4: $\quad (q, r) \leftarrow \text{BinaryDividePos}(a, b)$
5: $\quad$ **if** $(j, q) = (1, 3)$ **then** ▷ harmless iteration
6: $\quad\quad d \leftarrow a - b'$
7: $\quad\quad m \leftarrow \nu(d) \text{ div } 2$
8: $\quad\quad c \leftarrow (d - (-1)^m d/4^m)/5$
9: $\quad\quad s \leftarrow (s + m(a-1)/2) \bmod 2$
10: $\quad\quad (a, b) \leftarrow (a - 4c, b + 2c)$
11: $\quad$ **else** ▷ good or bad iteration
12: $\quad\quad s \leftarrow (s + (a-1)(b'-1)/4) \bmod 2$
13: $\quad\quad (a, b) \leftarrow (b', r/2^j)$
14: $\quad s \leftarrow (s + j(a^2 - 1)/8) \bmod 2, \quad j \leftarrow \nu(b), \quad b' \leftarrow b/2^j$
15: **if** $a = 1$ **then** return $(-1)^s$ **else** return 0

Analysis of the quadratic algorithm

Lemma

Algorithm QuadraticBinaryJacobi needs $O(n)$ iterations.

Proof.

Consider a block of three iterations (G, B, or H):

- G multiplies $a + 2b$ by at most $9/16 < 5/8$;
- HH is forbidden, thus we have either $HB = U^m B$ or BB;
- UB multiplies $a + 2b$ by at most $5/8$, and U^{m-1} leaves it unchanged;
- BB multiplies $a + 2b$ by at most $1/2 < 5/8$.

Thus each three iterations multiply $a + 2b$ by at most $5/8$, thus the number of iterations if $cn + O(1)$, where $c = 3/\log_2(8/5) \approx 4.4243$. □

A subquadratic LSB algorithm for the Jacobi symbol

We can modify Algorithm QuadraticBinaryJacobi to get a subquadratic algorithm for the Jacobi symbol, following the general ideas of the subquadratic LSB GCD algorithm of Stehlé and Zimmermann.

Details are given in Brent and Zimmermann, *Proc. ANTS-IX* (Nancy, July 2010) – preprint available from my website.

Computational results for large inputs

Timings on a 2.83Ghz Core 2 with GMP 4.3.1, with inputs of one million 64-bit words.

GMP's fast gcd takes 45.8s.

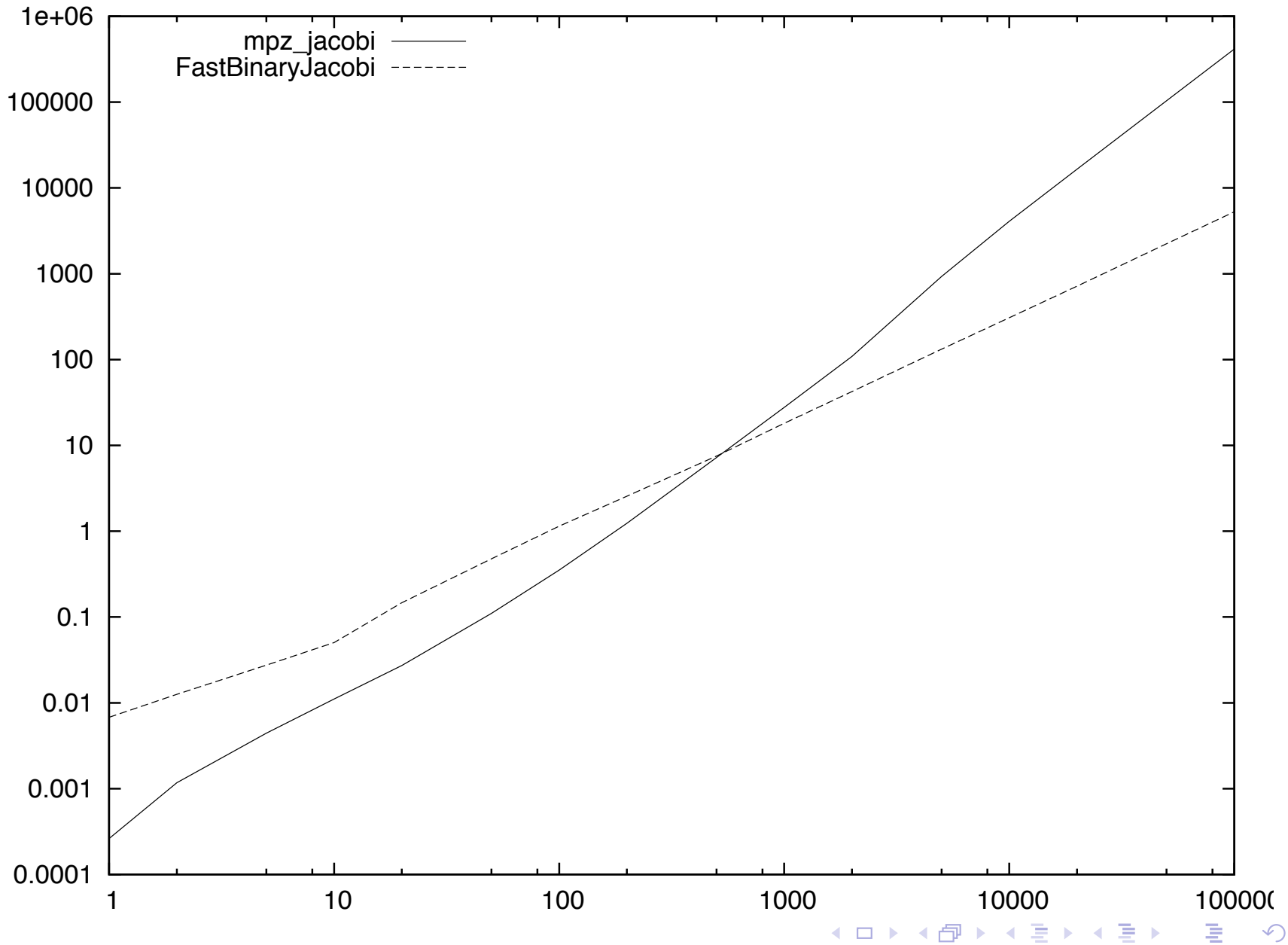
An implementation of the (fast) binary gcd takes 48.3s.

Our implementation FastBinaryJacobi takes 83.1s.

Our implementation is faster than GMP's $O(n^2)$ code from about 535 words (about 10,000 decimal digits).

See the following graph (note the log-log scale).

Comparison with GMP 4.3.1



Summary

We have given:

- the first LSB algorithms for the Jacobi symbol;
- the first complete (description + code) subquadratic Jacobi algorithm;
- we do not need to compute the (LSB or MSB) quotient or remainder sequences;
- we introduced “harmless” iterations to circumvent the problem of “ugly” iterations, but conjecture that this trick is not necessary.

Acknowledgements

Thanks to:

- Steven Galbraith for asking the original question;
- Damien Stehlé for suggesting use of an LSB algorithm;
- Arnold Schönhage for his comments and pointers to earlier work;
- The ARC and the ANC Équipe Associée (INRIA) for their support.

References

- Eric Bach, A note on square roots in finite fields, *IEEE Trans. on Information Theory*, **36**, 6 (1990), 1494–1498. [First known mention in print of a subquadratic algorithm for the Jacobi symbol.]
- Eric Bach and Jeffrey O. Shallit, *Algorithmic Number Theory, Volume 1: Efficient Algorithms*, MIT Press, 1996. Solution to problem 5.52. [Sketches a subquadratic algorithm attributed to Schönhage.]
- Richard P. Brent and Paul Zimmermann, An $O(M(n) \log n)$ algorithm for the Jacobi symbol, *Proc. ANTS-IX*, LNCS **6197** (2010), 83–95. [Fills the gaps in this talk.]

References continued

- Richard P. Brent and Paul Zimmermann, *Modern Computer Arithmetic*, Cambridge University Press, 2010, §1.6.3. [For discussion of subquadratic algorithms.]
- Benoît Daireaux, Véronique Maume-Deschamps and Brigitte Vallée, The Lyapunov tortoise and the dyadic hare, *Proc. 2005 Internat. Conf. on Analysis of Algorithms*, DMTCS Proc. AD (2005), 71–94. [For rigorous average-case analysis of some relevant GCD algorithms.]
- C. F. Gauss, Theorematis fundamentalis in doctrina de residuis quadraticis, demonstrationes et ampliatones novæ, *Comm. Soc. Reg. Sci. Gottingensis Rec.* **4**, 1818. [Gives an identity necessary in Bach and Shallit's subquadratic Jacobi algorithm.]

References continued

- Donald E. Knuth, The analysis of algorithms, in *Actes du Congrès International des Mathématiciens de 1970*, vol. 3, Gauthiers-Villars, Paris, 269–274. [The first subquadratic GCD algorithm, but with a sub-optimal time bound.]
- Niels Möller, On Schönhage's algorithm and subquadratic integer GCD computation, *Math. Comp.* **77**, 261 (2008), 589–607. [Shows how to avoid "fixup" steps in fast MSB GCD algorithms.]
- Arnold Schönhage, Schnelle Berechnung von Kettenbruchentwicklungen, *Acta Informatica* **1** (1971), 139–144. [The first subquadratic (MSB) GCD algorithm with sharp time bound.]
- Arnold Schönhage, personal communication Dec. 2009. [Describes a subquadratic (MSB) Jacobi algorithm that does not use the identity of Gauss.]

References continued

- Arnold Schönhage, Andreas F. W. Grotefeld and Ekkehart Vetter, *Fast Algorithms: A Multitape Turing Machine Implementation*, BI-W, Mannheim, 1994. [Mentions, without details, a subquadratic (MSB) Jacobi algorithm.]
- Jeffrey Shallit and Jonathan Sorenson, A binary algorithm for the Jacobi symbol, *ACM SIGSAM Bulletin* **27**, 1 (January 1993), 4–11. [Adapts the binary GCD algorithm to give a quadratic algorithm for the Jacobi symbol.]
- Damien Stehlé and Paul Zimmermann, A binary recursive gcd algorithm, *Proc. ANTS-VI*, LNCS **3076** (2004), 411–425. [The first subquadratic (LSB) GCD algorithm. We adapted it to give a subquadratic Jacobi algorithm.]