Comparing Several GCD Algorithms

T. Jebelean

RISC–Linz, A-4040 Austria
tjebelea@risc.uni-linz.ac.at

Abstract

We compare the execution times of several algorithms for computing the GCD of arbitrary precision integers. These algorithms are the known ones (Euclidean, binary, plus-minus), and the improved variants of these for multidigit computation (Lehmer and similar), as well as new algorithms introduced by the author: an improved Lehmer algorithm using two digits in partial cosequence computation, and a generalization of the binary algorithm using a new concept of "modular conjugates". The last two algorithms prove to be the fastest of all, giving a speed-up of 6 to 8 times over the classical Euclidean scheme, and 2 times over the best currently known algorithms. Also, the generalized binary algorithm is suitable for systolic parallelization, in "least-significant digits first" pipelined manner.

1 Introduction

Computation of the Greatest Common Divisor (GCD) of long integers is heavily used in computer algebra systems, because it occurs in normalization of rational numbers and other important subalgorithms. According to our experiments [5], in typical algebraic computations more than half of the time is spent for calculating GCD of long integers. For instance, in Gröbner Bases computation [4], calculating GCD takes 53% of the total time if the length of the input coefficients is 5 decimal digits and 70% if the length is 50.

We report here on the computing time of multiprecision GCD computation by the following algorithms:

- **Euclid**, **I-Euclid**, **I-I-Euclid**: The classical Euclidean scheme, the improvement of it for multidigit integers ([13]), and a further improvement by the author using two digits in partial cosequence computation ([8]).
- **binary**, **I-binary**: The binary GCD algorithm ([16]) and its improvement for multidigit integers (Gosper, see [12]).
- **PlusMinus**, **I-PlusMinus**: The plus-minus scheme introduced in [2] and its improvement for multidigit computation.
- **G-binary**, **I-G-binary**: A new algorithm for multiprecision GCD, which generalizes the binary and plus-minus GCD algorithms and its improvement by using two digits in computation of cofactors [10].

We did not consider the GCD algorithms based on FFT multiplication scheme ([15], [14]), which are asymptotically faster, but are not expected to give a practical speed-up for the range of integers we are interested in (up to 100 words of 32 bits).

The results of the experiment show the following speed-up over the raw Euclidean scheme, for random pairs of integers with 100 words of 32 bits:

- 2 times for **binary** and **PlusMinus**;
- 3.5 times for **I-Euclid** and **I-PlusMinus**;
- 5 times for **I-binary** and **G-binary**;
- 6.5 times for **I-I-Euclid**;
- 8 times for **I-G-binary**.

The investigation presented here is part of a more general research aimed at speeding-up algebraic computations by systolic parallelization of arbitrary precision rational arithmetic. In this context, it is important to note that **I-PlusMinus**, **G-binary** and **I-G-Binary** are suitable for this kind of parallelization. Also, these three algorithms work "least–significant digits first" (LSF), hence they are suitable for pipelined aggregation with other LSF systolic algorithms (multiplication: [1], exact division: [11]).

1063-6889/93 $03.00 © 1993 IEEE

2 Description of algorithms

We present here the outline of the GCD algorithms which were measured, and indicate the appropriate literature for the readers which are interested in more details concerning the correctness proofs and complexity analysis.

2.1 Euclid

Starting with two positive integers $A_0 \geq A_1$, one computes the remainder sequence $\{A_k\}_{1 \leq k \leq n+1}$ defined by the relations:

$$A_{k+2} = A_k \bmod A_{k+1}, \quad A_{n+1} = 0.$$

and then one has: $GCD(A_0, A_1) = A_n$.

An extensive discussion on Euclidean algorithm is presented in [12].

2.2 I-Euclid

The *extended* Euclidean algorithm (also in [12]) consists in computing the additional sequences $\{q_k\}_{1 \leq k \leq n}, \{u_k, v_k\}_{0 \leq k \leq n+1}$ defined by:

$$\begin{array}{r} q_{k+1} = \lfloor A_k / A_{k+1} \rfloor, \\ u_0 = 1, v_0 = 0, u_1 = 0, v_1 = 1, \\ u_{k+2} = u_k + q_{k+1} * u_{k+1}, \\ v_{k+2} = v_k + q_{k+1} * v_{k+1}. \end{array} \qquad (1)$$

which have the properties:

$$A_k = \begin{cases} u_k * A_0 - v_k * A_1, & \text{if } k \text{ even}, \\ -u_k * A_0 + v_k * A_1, & \text{if } k \text{ odd}, \end{cases} \qquad (2)$$

The sequences $\{u_k, v_k\}$ of *cofactors* are called *cosequences* of $\{A_k\}$.

If A_0 and A_1 are multiprecision integers, then let a_0 and a_1 be the most significant 32 bits of A_0 and the corresponding bits of A_1. Lehmer [13] noticed that part of the sequence $\{q_k\}$ can be computed by:

$$q'_{k+1} = \lfloor a_k / a_{k+1} \rfloor, \quad a_{k+2} = a_k \bmod a_{k+1}, \qquad (3)$$

and as long as $q'_k = q_k$, the sequences $\{u_k, v_k\}$ computed as in (1) are the correct ones. Hence, the algorithm **I-Euclid** simulates several steps of the Euclidean algorithm by using only simple precision arithmetic. This process is called *digit partial cosequence computation* (see [6]). When $q'_{k+2} \neq q_{k+2}$, then A_k and A_{k+1} are recovered using (2), and the cycle can start again.

The algorithm needs a sufficient condition for $q'_{k+1} = q_k$. We have used:

$$a_{k+1} \geq v_{k+1} \textbf{ and } (a_k - a_{k+1}) \geq (v_k + v_{k+1}), \qquad (4)$$

which was developed in [6].

Recovering A_k, A_{k+1} involves 4 multiplications of a single digit number by a multidigit number, and this is the most time consuming part of the whole computation. Experimentally, one notices that final cofactors are usually shorter than 16 bits. Therefore, if partial cosequences would be computed for pairs of double digits, then the recovering step would require the same computational effort, but will occur (roughly) two times less frequently. This idea suggests the next improvement of the Euclidean algorithm.

2.3 I-I-Euclid

The basic structure of the algorithm is the same as above, but we use a new condition for $q'_k = q_k$, which also ensures that cofactors are smaller than one word (32 bits). For developing this condition, we use the *continuant polynomials* (see also [12]) defined by:

$$\begin{cases} Q_0() = 1, \\ Q_1(x_1) = x_1, \\ Q_{k+2}(x_1, \ldots, x_{k+2}) = \\ \quad Q_k(x_1, \ldots, x_k) + x_{k+2} * Q_{k+1}(x_1, \ldots, x_{k+1}) \end{cases} \qquad (5)$$

which are known to enjoy the symmetry:

$$Q_k(x_1, \ldots, x_k) = Q_k(x_k, \ldots, x_1). \qquad (6)$$

By comparing the recurrence relations (1) and (5) one notes:

$$\begin{array}{r} u_k = Q_{k-2}(q_2, \ldots, q_{k-1}), \\ v_k = Q_{k-1}(q_1, \ldots, q_{k-1}). \end{array} \qquad (7)$$

Also, by transforming (3) into:

$$a_k = a_{k+2} + q_{k+1} * a_{k+1},$$

and using $a_k > a_{k+1}$, one can prove:

$$\begin{array}{r} a_0 \geq a_k * Q_k(q_k, \ldots, q_1), \\ a_1 \geq a_k * Q_{k-1}(q_k, \ldots, q_2). \end{array}$$

Hence by (6) and (7) one has:

$$\begin{array}{r} v_{k+1} \leq a_0 / a_k, \\ u_{k+1} \leq a_1 / a_k. \end{array} \qquad (8)$$

The previous relations allow us to develop an alternative to (4), which is:

$$a_{k+2} \geq \sqrt{a_0}. \qquad (9)$$

Indeed, one has:

$$a_k > a_{k+1} > a_{k+2} \geq \sqrt{a_0},$$

$$v_{k+1} \leq a_0/a_k < a_0/\sqrt{a_0} = \sqrt{a_0} < a_{k+1},$$

and:

$$a_k - a_{k+1} \geq a_k - q_{k+1} * a_{k+1} = a_{k+2} \geq \sqrt{a_0},$$

$$v_k + v_{k+1} \leq v_k + q_{k+1} * v_{k+1} = v_{k+2} \leq$$
$$\leq a_0/a_{k+1} < a_0/\sqrt{a_0} = \sqrt{a_0}.$$

Also, note that when (9) holds:

$$\begin{array}{l} v_k < v_{k+1} \leq a_0/a_k < \sqrt{a_0}, \\ u_k < u_{k+1} \leq a_1/a_k < \sqrt{a_0}. \end{array} \qquad (10)$$

For the practical implementation, if a_0, a_1 are double words (64 bits), then condition (9) is satisfied if the higher word of a_{k+2} is nonzero, and (10) implies $u_k, v_k, u_{k+1}, v_{k+1}$ are at most one word (32 bits) long.

A more detailed description of the theoretical background and of the implementation can be found in [8].

2.4 Binary and PlusMinus

The **binary** GCD algorithm ([16], [12], [3]) is based on the relations:

$$\begin{array}{l} GCD(A,B) = GCD(A-B,B), \\ \text{If } A \text{ odd, } B \text{ even, then} \\ \qquad GCD(A,B) = GCD(A,B/2). \end{array} \qquad (11)$$

The algorithm begins by shifting A_0, A_1 rightwise as many positions as there are zero trailing bits, and stores the number of common zero bits for being incorporated into the resulting GCD at end of computation. After shifting A_0, A_1 are odd, hence $A_2 = |A_0 - A_1|$ is even. A_2 is then shifted rightwise for skipping the zero bits and the cycle is repeated with A_2 and $\min(A_0, A_1)$. Note that it is necessary to know at each step which of A_k, A_{k+1} is the largest. When $A_k = 0$, then the GCD is A_{k-1} shifted leftward with the number of common zero bits of A_0, A_1.

The multidigit version **I-binary** (see [12]) is developed from **binary** in the same way **I-Euclid** is developed from **Euclid**. For a certain number of steps, one can decide which shifts/subtractions are needed by only looking at the most–significant and least–significant digits of A_0, A_1. These shifts and subtractions are encoded into cofactors which allow the recovery of A_k, A_{k+1} when single digit operation is not possible anymore.

The need to compare A_k, A_{k+1} at each step prevents efficient parallelization of **binary**. In order to overcome this, the **PlusMinus** algorithm was developed in [2]. This algorithm is based on (11) and supplementary relations:

$$\begin{array}{l} GCD(A,B) = GCD(A+B,B), \\ \text{If } A,B \text{ odd, then } 4|(A+B) \text{ or } 4|(A-B). \end{array}$$

Therefore, each step consists of choosing $A'_{k+2} = A_k + A_{k+1}$ or $A'_{k+2} = A_k - A_{k+1}$ such that $4|A'_{k+2}$, and then setting A_{k+2} to A'_{k+2} shifted rightward to skip the zero bits. Note that in this case it is not important anymore to know which of A_k, A_{k+1} is the largest. The scheme works even if one (or both) of the operands become negative.

This makes easier the implementation of **I-PlusMinus**, since there is no need to keep track of the most–significant digits of A_0, A_1.

Note also that further improvement of **I-binary** and **I-PlusMinus** is not possible in the way we did it for **I-Euclid**, because in this case the cofactors are not bound by half-word size.

2.5 G-Binary

A natural generalization of the plus-minus scheme is the following:

Let $m \geq 1$ be a constant. Given positive long integers A, B, let a, b be the least significant $2m$ bits. Find x, y with at most m bits, such that:

$$2^{2m} \mid (x * a + y * b) \text{ or } 2^{2m} \mid (x * a - y * b) \qquad (12)$$

We shall call x, y the *modular conjugates* of a, b.

Note that for $m = 1$ the plus-minus scheme is obtained.

It is interesting that, if a, b are odd, then such modular conjugates always exist. Indeed, since b is odd, there exists $b^{-1} \bmod 2^{2m}$. Then (12) is equivalent to:

$$(x * c \pm y) \bmod 2^{2m} = 0, \qquad (13)$$

where:

$$c = (a * b^{-1}) \bmod 2^{2m}.$$

If one applies the extended Euclidean algorithm for $a_0 = 2^{2m}, a_1 = c$, one obtains $GCD(2^{2m}, c) = 1 < 2^m$, since c is odd. Let us consider that k for which $a_{k-1} \geq 2^m$ and $a_k < 2^m$. According to (8), we also have: $v_k < \sqrt{a_0} = 2^m$. If we set $x = v_k$ and $y = a_k$, then by (2):

$$\pm u_k * 2^{2m} \mp x * c = y,$$

which implies (13), hence these are the conjugates we need.

Now let A, B be two multidigit odd integers. By applying the scheme above we get $C = (x*A \pm y*B)/2^{2m}$ which is (roughly) m bits shorter than $\max(A, B)$. This is efficient when (length(A) - length(B)) is small (for $m = 64$, we experimentally observed that the best threshold is 8). Otherwise, it is more efficient to bring the lengths closer by another scheme – for instance, by division. However, division is not suitable for parallelization, and it is also a relatively slow operation. We applied instead the "exact division" scheme described in [9], which works like this:

> Let be d = length(A) - length(B) and a, b the trailing d bits of A, B.
> Set $c = (a * b^{-1}) \bmod 2^d$.
> Then $C = (A - c * B)/2^d$ is (roughly) d bits shorter than A.

Hence, the generalized binary algorithm consists in alternating the "exact division" step with "inter-reduction by modular conjugates" step. After each alternation, the two operands become (roughly) m bits shorter ($m = 16$ for **G-binary** and $m = 32$ for **I-G-binary**). Note that two such steps need 3 multiplications of a simple precision integer by a multiprecision integer (vs. 4 multiplications in **I-Euclid**), but nevertheless the same reduction of the two operands is obtained.

The algorithm terminates when a 0 is obtained. If 0 is obtained after an exact division step, then $G' = B$, and if 0 is obtained after an inter-reduction step, then $G' = (A * GCD(x, y))/y = (B * GCD(x, y))/x$, where G' is the approximative GCD of initial A, B. However, G' is in general different from $G = GCD(A, B)$, because

$$GCD(A, B) \mid GCD(B, x * A \pm y * B),$$

but not the other way around. A "noise" factor may be introduced at each inter-reduction step, and the combined noise must be eliminated after finding G' by:

$$\begin{aligned} GCD(A, B) &= GCD(G', A, B) \\ &= GCD(GCD(G', A \bmod G'), B \bmod G'). \end{aligned} \tag{14}$$

This "noise" is nevertheless small in the average case (see [10]), and we experimentally noticed that the operations (14) take less than 5% of the total GCD computation time, in average.

We also note that the operation $x^{-1} \bmod 2^{2m}$, which is quite costly when performed via the extended Euclidean algorithm, was implemented using a scheme developed in [9]:

> Let be $x = x_1 * \beta + x_0$. Then:
>
> $$x^{-1} \bmod \beta^2 = (((1 - x * a') * x') + x') \bmod \beta^2$$
>
> $$\text{where } \; x' = (x_0)^{-1} \bmod \beta$$

Using this relation, the computation of modular inverse of a [double] word was reduced to the modular inverse of a half-word, which was done by look-up in a precomputed table.

More details concerning the theoretical background and the implementation of this algorithm can be found in [10].

Finally, let us note that this algorithm is suitable for systolic implementation in the "least–significant digits first" manner, because all the decisions on the procedure are taken using only the lowest digits of the operands.

3 Experiment settings and results

We implemented the algorithms using the GNU multiprecision arithmetic library [7], under the GNU optimizing C compiler. The experiments were done using a the Digital DECstation 5000/200 (RISC architecture). For each length, each of the algorithms was applied to 1000 pairs of random integers.

The figures on the next page present the absolute timings in milliseconds and the speed-up over the raw Euclidean algorithm (**Euclid**). The absolute timings for 100 word operands are 317 milliseconds for the raw Euclidean algorithm and 39 milliseconds for improved generalized binary (**I-G-binary**).

Acknowledgements

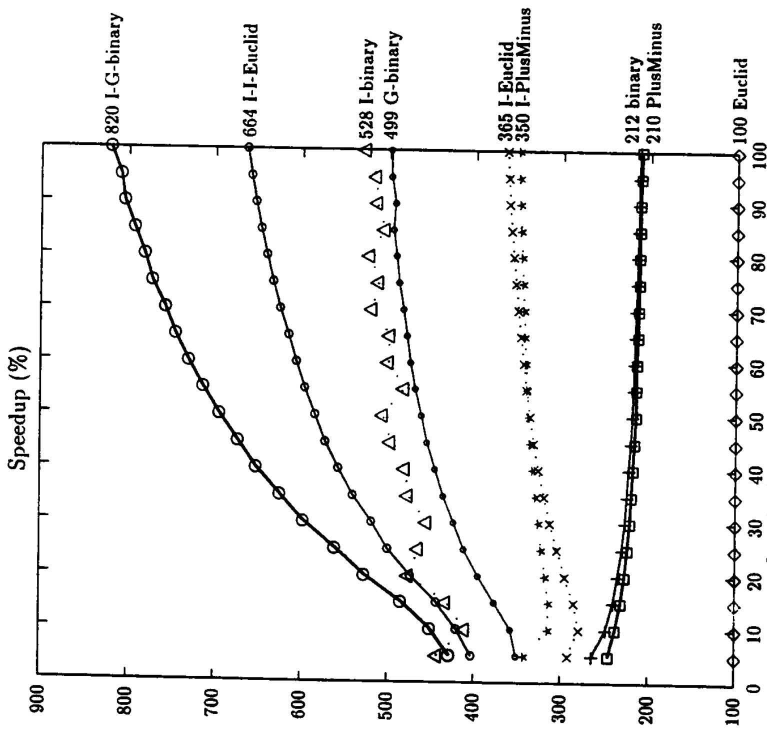
Austrian Forschungsförderungsfonds, project S5302-PHY (Parallel Symbolic Computation).

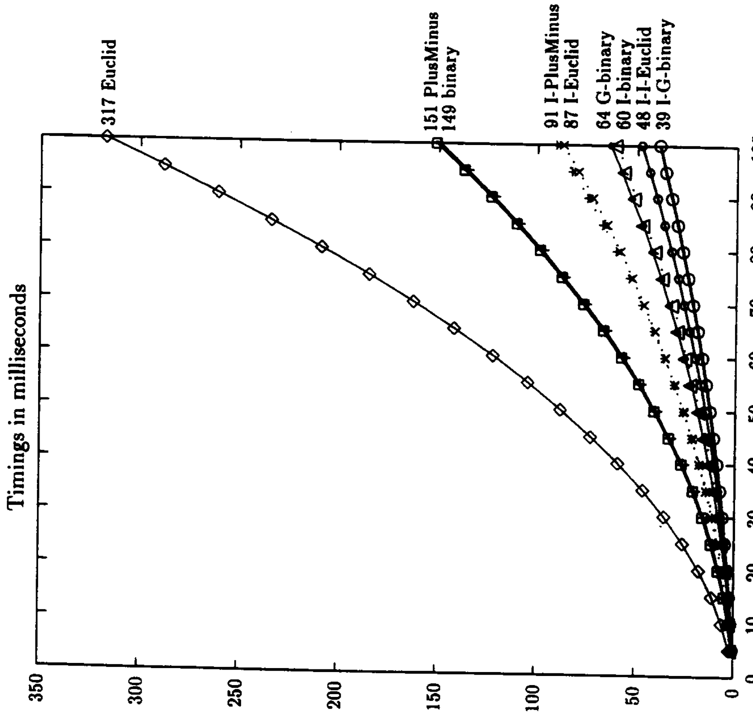
Austrian Ministry for Science and Research, project 613.523/3-27a/89 (Gröbner Bases) and doctoral scholarship.

POSSO project(Polynomial Systems Solving – ESPRIT III BRA 6846).

References

[1] A. J. Atrubin, "A one–dimensional iterative multiplier", *IEEE Trans. Computers*, Vol C-14, pp. 394-399, 1965.


Speedup (%)
820 I-G-binary
664 I-I-Euclid
528 I-binary
499 G-binary
365 I-Euclid
350 I-PlusMinus
212 binary
210 PlusMinus
100 Euclid
Length of operands (32 bit words)


Timings in milliseconds
317 Euclid
151 PlusMinus
149 binary
91 I-PlusMinus
87 I-Euclid
64 G-binary
60 I-binary
48 I-I-Euclid
39 I-G-binary
Length of operands (32 bit words)

[2] R. P. Brent, H. T. Kung, "Systolic VLSI arrays for linear-time GCD computation", in V. Anceau, E. J. Aas (eds.), *VLSI'83*, Elsevier (North-Holland), pp. 145 - 154, 1983.

[3] R. P. Brent, " Analysis of the binary Euclidean algorithm", in J. F. Traub (ed.), *New directions and recent results in algorithms and complexity*, Academic Press, pp. 321 - 355, 1976.

[4] B. Buchberger, "Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory", in N. K. Bose (ed.), *Multidimensional Systems Theory*, D. Reidel Publishing Co., 1985.

[5] B. Buchberger, T. Jebelean, "Parallel Rational Arithmetic for Computer Algebra systems: Motivating Experiments", RISC-Linz Report 92-29, May 1992.

[6] G. E. Collins, "Lecture notes on arithmetic algorithms", Univ. of Wiscousin, 1980.

[7] T. Granlund, "GNU MP: The GNU multiple precision arithmetic library", Free Software Foundation, 1991.

[8] T. Jebelean, "Improving the multiprecision Euclidean algorithm", RISC-Linz Report 92-69.

[9] T. Jebelean, "An algorithm for exact division", *J. Symbolic Computation*, Vol. 15, February 1993.

[10] T. Jebelean, "A generalization of the binary GCD algorithm", ISSAC'93 (Kiew, July 1993).

[11] T. Jebelean, "Systolic algorithms for exact division", PARS Workshop (Dresden, April 1993).

[12] D. E. Knuth, *The art of computer programming*, Vol. 2, 2^{nd} edition, Addison-Wesley 1981.

[13] D. H. Lehmer, "Euclid's algorithm for large numbers", *Am. Math. Mon.*, Vol. 45, pp. 227-233, 1938.

[14] R. T. Moenck, "Fast computation of GCDs", Proceedings ACM V^{th} Symp. Theory of Computing, pp. 142-151, 1973.

[15] A. Schönhage, "Schnelle Berechung von Kettenbruchentwicklugen", *Acta Informatica*, Vol. 1, pp. 139-144, 1971.

[16] J. Stein, "Computational problems associated with Racah algebra", *J. Comp. Phys.*, Vol. 1, pp. 397-405, 1967.