Proceedings of the Risa Consortium
1997 pp. 53– pp. 68

Implementation and Experiments of Faster Algorithms for Integer GCD's in Risa/Asir

Hirokazu MURAO*

Computer Centre, The University of Tokyo

(RECEIVED 1997/5/30)

Abstract. Some new algorithms for integer GCD's have been implemented in Risa/Asir to improve the efficiency of the existing function of an Euclidean algorithm. The implementation includes the latest algorithm by Jebelean or Weber, which is recognized as most efficient in practice. In this paper, we shall give a brief review of the existing algorithms and describe the details of our implementation. Also reported are the timings measured using various sample problems, result of which indicates that the newly implemented algorithms are considerably faster than the original implementation, and that the latest algorithm is most efficient in practice especially when very long integers are treated. Based on this fact, we suggest changing the default algorithm in Risa/Asir.

1. Introduction

Algorithms and computing methods for various basic calculations, e.g., polynomial arithmetics, have been being revisited and reconsidered these days, mainly aiming at application of parallel processing to algebraic computation. As one such subject, the problem of GCD calculation of long integers has been being extensively studied in the last several years by Tudor Jebelean [3] and by Kenneth Weber [15]. Stimulated by their work, the author has been working on implementing and testing some of the new algorithms in Risa/Asir [9]. The mathematical background and the structures of the existing algorithms are very

*Has been supported in part by Grant-in-Aid for Scientific Research from The Ministry of Education, Science, Sports and Culture, Japan under Grant (C) 07680337. murao@cc.u-tokyo.ac.jp

simple and similar to each other. However, our new implementation, even of a limited number of algorithms, has revealed much improvement in speed to the original implementation in Risa/Asir. This improvement heavily depends on the implementation techniques enabled by the difference of the algorithmic nature. In this paper, we shall describe our new implementation in some details and report the results obtained so far. In particular, the implementation of the latest algorithms, called the generalized-binary algorithm by Jebelean and the accelerated algorithm by Weber and recognized as most efficient, has been completed since the last report by the author [5], and this report includes also this latest and the most successful result.

This paper is organized as follows. In the next section, we shall extensively review the known algorithms on a publication basis, and give a summary of each algorithm. Section 3. is devoted to describe the details of our new implementation, e.g., the techniques used and the specification of user interface. Section 4. gives timings taken on two different kinds of processors using various sample problems, including the application to the Gröbner basis calculation. Finally, Section 5. concludes this report.

2. Existing algorithms

In this section, we give a brief review of the known algorithms. In the following, let X and Y be two given integers where $X > Y$, and we consider the computation of $G = \gcd(X, Y)$.

2.1. Classical algorithms

In his famous textbook, Knuth describes the following three algorithms, the latter two of which are suited for long integers.

Euclid's algorithm :

Letting $A_0 = X$ and $A_1 = Y$ initially, compute the following remainder sequence

$$Q_i \Leftarrow \lfloor A_{i-1}/A_i \rfloor, \tag{1}$$

$$A_{i+1} \Leftarrow A_{i-1} - Q_i A_i = A_{i-1} \bmod A_i \tag{2}$$

for $i = 1, 2, ..., n$ until we obtain $A_{n+1} = 0$. The required GCD is given by $G = A_n$.

We consider the sequence of cofactors U_i and V_i defined by $U_0 = V_1 = 1$, $U_1 = V_0 = 0$, and

$$\begin{pmatrix} U_{i+1} \\ V_{i+1} \end{pmatrix} = \begin{pmatrix} U_{i-1} \\ V_{i-1} \end{pmatrix} - Q_i \begin{pmatrix} U_i \\ V_i \end{pmatrix}, \quad i = 1, 2, ..., n-1. \tag{3}$$

Then, as is well known, the following equation holds:

$$A_i = U_i A_0 + V_i A_1. \tag{4}$$

Here, we notice

- that the cofactor sequence is determined only from the sequence Q_i, and
- that the exact calculation of the sequences of Q_i and of the cofactors is not essential to the calculation of $\gcd(X, Y)$.

These facts are utilized in Lehmer's improved algorithm [4] designed for long integers (*bignums*) by possible elimination of their exact divisions.

Lehmer's algorithm : Suppose that X and Y are long integers. Then, in Eq.'s (1) and (2), the calculations of Q_i and A_i are supposed to be exact and to be performed with full precisions. However, these exact calculations are not mandatory, as described below.

- Assume that the lengths (the numbers of words) of A_{i-1} and A_i are equal, and let a_j denote the most significant word of A_j. Then, the following inequality holds:

$$q_i = \left\lfloor \frac{a_{i-1}}{a_i + 1} \right\rfloor \leq Q_i \leq \bar{q}_i = \left\lfloor \frac{a_{i-1} + 1}{a_i} \right\rfloor.$$

  Therefore, if q_i and $\bar{q}_i$ are equal, then it is equal to Q_i.
- Notice that both q_i and $\bar{q}_i$ can be obtained by calculations with single words.
- Therefore, while q_i and $\bar{q}_i$ are equal, the remainder sequence calculation with a_i corresponding to Eq.'s (1) and (2), which can be done only with operations on single words, exactly reflects the calculation of the Q_i sequence.
- At the time when $q_k \neq \bar{q}_k$, the exact A_k of required precisions can be recovered by (4) using the cofactor sequence (3).

For the formulation of the algorithm, refer to the textbook.

Note that the above value sequences obtained via division are not mandatory yet for the $\gcd(X, Y)$ calculation but its giving a method for the purpose below is the point. The essence of any algorithm for integer GCD's is the generation of a new pair $\{C_1, C_2\}$ from a pair $\{B_1, B_2\}$ of positive integers satisfying the conditions

- $\gcd(B_1, B_2) = \gcd(C_1, C_2)$, and
- $\min(B_1, B_2) > \min(C_1, C_2)$ for termination,

and the continuability of this generation until $\gcd(X, Y)$ is obtained. Assume $B_1 > B_2$ and let $t \Leftarrow B_1 \bmod B_2$. Then, the above Euclidean algorithms yield a new pair by $\{C_1, C_2\} \Leftarrow \{B_2, t\}$. Alternatively, one may set $\{C_1, C_2\} \Leftarrow \{t, B_2 - t\}$. The simplest method satisfying the above conditions will be to let

$$\{C_1, C_2\} \Leftarrow \{\min(B_1, B_2), \mid B_1 - B_2 \mid\}, \tag{5}$$

which gives the binary algorithm developed by Stein [12].

The binary GCD algorithm : Notice that in Eq. (5), if B_1 and B_2 are odd, the difference $B_1 - B_2$ is even. If $\gcd(B_1, B_2)$ is known to be odd, then we can remove 2's factor from the difference without violating the former condition, by simply right-shifting the difference, and the convergence to the GCD will be speeded. Namely, the subtraction and the subsequent shift operations, in place of the costly division operation in the Euclidean algorithm, suffice to yield a new pair with an integer of reduced magnitude. There follows a description of the right-shift binary algorithm.

(1) Remove 2's factors from given two integers X and Y, and let $X \Rightarrow 2^{k_1} B_1$ and $Y \Rightarrow 2^{k_2} B_2$, where B_1 and B_2 are odd.

(2) While $(w \Leftarrow B_1 - B_2) \neq 0$, do the following:

(2.1) remove 2's factor from w: $w \Rightarrow 2^s t$;

(2.2) if $t > 0$, let $B_1 \Leftarrow t$, and otherwise let $B_2 \Leftarrow -t$.

(3) Finally, we obtain $\gcd(X, Y) = 2^{\min(k_1, k_2)} B_2$.

Stein also proposed a left-shift algorithm. Assume $B_1 > B_2$. The algorithm obtains $t = 2^e B_2$ such that $t \leq B_1 < 2t$ by left-shifting, and uses $\{B_2, \min(B_1 - t, 2t - B_1)\}$ as a new pair.

2.2. Recent developments

In the last several years, the algorithms for integer GCD's have been evolved to use general methods to make a new pair. Most of the newly developed algorithms are characterized by the following two features.

- Generalization of the binary algorithm to K-ary ones. Let $K = 2^m$ be the magnitude of a single word composing long integers (K need not be a power of 2 and can be a power of 10 as in some implementations). The intermediate expressions of the sequence are reduced by K-ary right-shift. For example, if we have $a = B_1/B_2 \bmod 2^m$, $B_1 - aB_2$

MODIV algorithm by Jebelean [1]

Input: integers U_0, V, N and a radix β (normally $= 2^m$), where $\gcd(V, \beta) = 1$.

Output: $W = U_0/V \bmod \beta^N$.

(0) $W \Leftarrow 0$; $w \Leftarrow V \bmod \beta$; $a' \Leftarrow w^{-1} \bmod \beta$;

(1) **for** $i = 0$ **to** $N - 1$ **do**

$b_i \Leftarrow a' (U_i \bmod \beta) \bmod \beta$;

$W \Leftarrow W + b_i \beta^i$;

$U_{i+1} \Leftarrow (U_i - b_i V) \bmod \beta^{N-i}/\beta$;

(2) **return** W;

Fig. 1. division $\bmod \beta^N$

is divisible by 2^m and $(B_1 - aB_2)/2^m$ obtained by 2^m-ary right-shift is to be used to replace B_1 to make a new pair.

- Use of the following generalized formula as a sequence, instead of the exact remainder sequence (2),

$$C_{i+1} \Leftarrow | \, b_i \, C_{i-1} - a_i \, C_i \, | \, / \beta^n. \tag{6}$$

Notice that the final result is a multiple of the true GCD and may contain a spurious factor.

In the following, we summarize the recently developed algorithms.

Jebelean (1993) [1]

- A method to determine the quotient of an exact division of long integers from the least significant side is developed, and the similar method is applied to the GCD calculations to make an K-ary algorithm as explained next.
- The algorithm MODIV of Figure 1 presents a general method to perform division $\bmod \beta^N$, and is applied to the computation of the reciprocal $\bmod \beta$ of the least significant word of a divisor to be used in a new K-ary algorithm, called EDGCD. This $\bmod \beta^N$ division algorithm is faster than the well-known Euclidean algorithm in general, and

serves as a fast method to compute $c/a \bmod 2^N$ for one-word integers c and a when used with $\beta = 2$ and realized with bit manipulations.

- It is claimed based on the experimental results that the new EDGCD algorithm is not better than Lehmer-Euclid's algorithm.

Sorensen (1994) [11]

- K-ary left- and right-shift algorithms are proposed, and at the same time, the generalized formula (6) is used.
- A simple method for parallelization is also presented, and the complexity analysis is made.

The algorithms seem not practical, compared with the following two similar algorithms, because of the use of tables for computing a_i and b_i of (6) and because the removal of spurious factor is done by trial divisions by prime factors of a_i's and b_i's accumulated during the intermediate calculations.

Jebelean (1994) [2] presented a generalized binary algorithm using (6). Let d_j denote the bit width of C_j, i.e., $d_j = \lceil \log_2 C_j \rceil$. The algorithm performs the elimination of less significant bits in two ways depending on the difference of the magnitudes of a pair as follows.

- When $d = d_{i-1} - d_i$ is not small, e.g., ≥ 8 for the bit width $m = 32$ of a single word, do the same elimination as in [1]:

$$a_i \Leftarrow (C_{i-1} \bmod 2^d)/(C_i \bmod 2^d) \bmod 2^d, \tag{7}$$

$$C_{i+1} \Leftarrow (C_{i-1} - a_i C_i)/2^d. \tag{8}$$

More concretely,

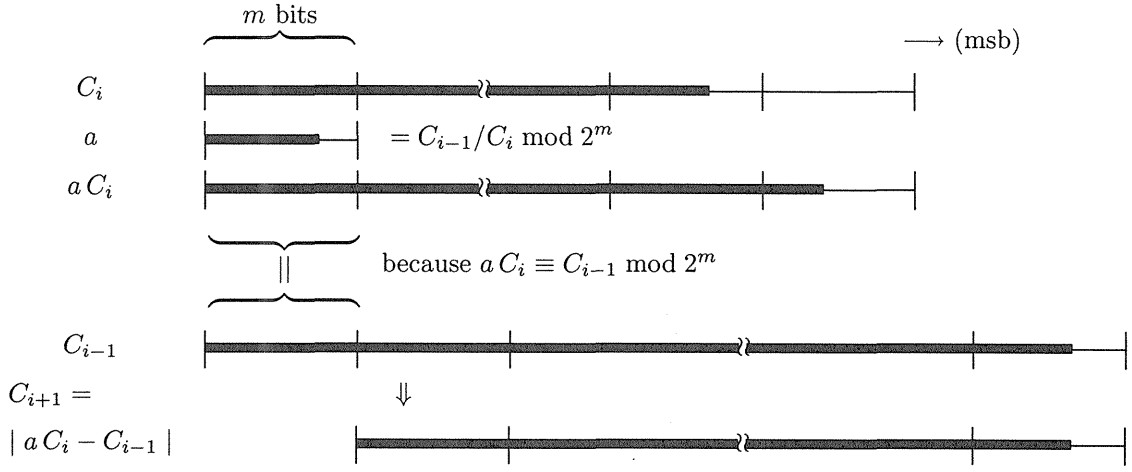
(1) apply the MODIV algorithm with $U_0 = 1$, $V = C_i \bmod 2^m$, $N = m$, $\beta = 2$ to obtain $w = V^{-1} \bmod 2^m$ (equivalent to Weber's method [16, Figure 5.]).

(2) letting $C \Leftarrow C_{i-1}$, repeat the following calculation until C becomes sufficiently small (until C gets < 0 in the EDGCD algorithm):

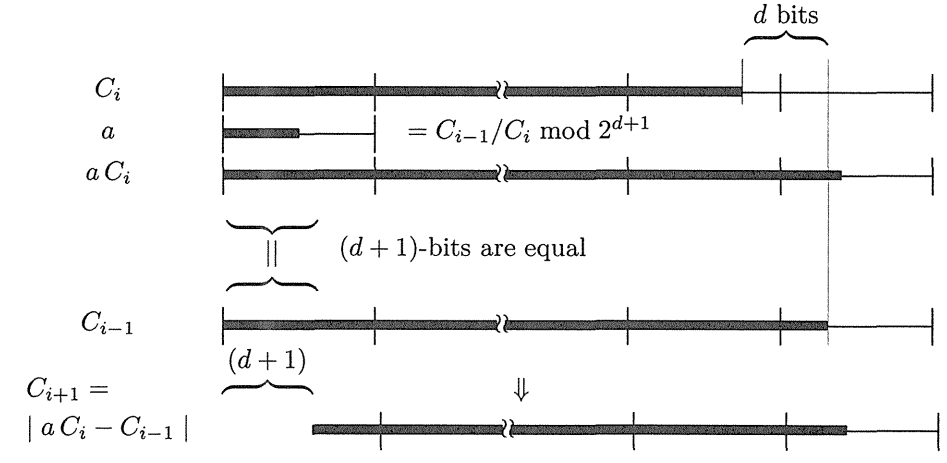
$$C \Leftarrow (C - a C_i)/2^m, \quad \text{where } a = (C \bmod 2^m)\, w.$$

After each iteration step, the bit width of C is reduced by m.

(i) MODIV/bmod operation when $\lceil \log_2 C_{i-1} \rceil - \lceil \log_2 C_i \rceil \geq m$



(ii) MODIV/bmod operation when $d = \lceil \log_2 C_{i-1} \rceil - \lceil \log_2 C_i \rceil < m$



(iii) generalized formula (6)

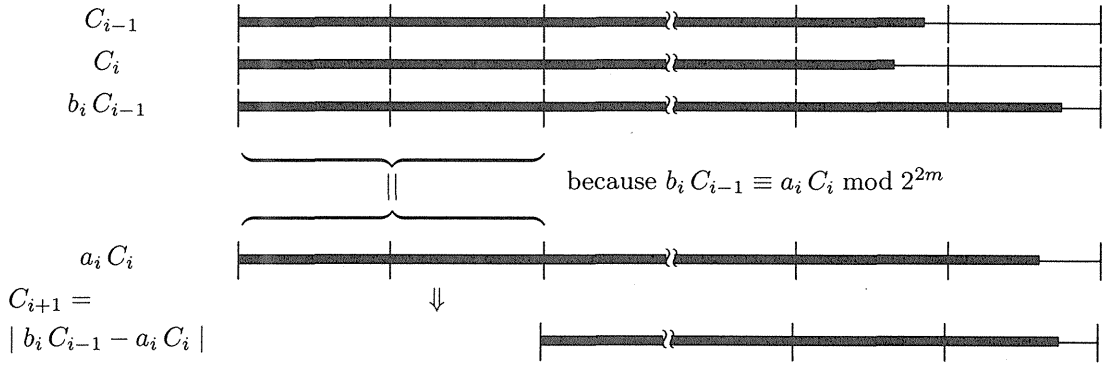


Fig. 2. Elimination of less significant bits

- Otherwise, use the generalized formula (6) with b_i and a_i chosen such that

$$b_i \left(C_{i-1} \bmod 2^{2m}\right) \equiv a_i \left(C_i \bmod 2^{2m}\right) \bmod 2^{2m}, \quad 0 < b_i, \mid a_i \mid < 2^m. \tag{9}$$

- The spurious factor possibly contained in the final C_n is removed by the GCD calculation $G = \gcd(\gcd(X \bmod C_n, C_n), Y \bmod C_n)$.

Weber (1995) [16] independently developed the same algorithm as the previous one and named it the accelerated algorithm.

- Detailed descriptions of the methods for required calculations are given, including the one for a_i and b_i which uses Wang's method [13] for a rational number reconstruction from two modular numbers.
- The dmod operation, which is equivalent to the calculation of C in the previous algorithm, is defined as

$$\mathrm{dmod}(u, v, \beta) = \frac{\mid u - (u/v \bmod \beta^{\lg_\beta(u)-\lg_\beta(v)+1})\, v \mid}{\beta^{\lg_\beta(u)-\lg_\beta(v)+1}},$$

  where $\lg_\beta(U)$ denotes the required number of words of magnitude β to represent a long integer U. In particular, the dmod operation when $\beta = 2$ is called **bmod** (bit-modulus), and is equivalent to the calculation of (7) and (8).
- There is also a reference to the bmod algorithm ($C_{i+1} \Leftarrow \mathrm{dmod}(C_{i-1}, C_i, 2)$), which is equivalent to Jebelean's EDGCD algorithm, and it is used for the GCD calculation of spurious factor removal.
- In [14], Weber parallelized his algorithm using a shared-memory multiprocessor system.

Figure 2 depicts how less significant bits are eliminated by the above operations.

Jebelean (1995) [3] suggested a few methods to improve Lehmer's algorithm.

- In Lehmer's algorithm, having pointed out that magnitudes of cofactors are of half-word size in most cases and that the calculation of A_k with multiple precisions via (4) is costly, Jebelean developed a method to perform the calculation in each step with double words. The empirical results indicate that this improvement is not very effective.
- The required condition to proceed the cosequence calculation is relaxed.
- A new algorithm adopting these two improvements is proposed.

3. Implementation in Risa/Asir

The original implementation in Risa/Asir for integer GCD calculation (`igcd`) uses Euclid's algorithm. In addition to the original, the following three algorithms are implemented in C:

- right-shift binary algorithm,
- bmod algorithm,
- accelerated (generalized binary) algorithm.

As can be seen later, Euclid's algorithm is sometimes most efficient when the lengths of two integer arguments are very different. This is because the longer argument can be reduced and shortened very much by the initial division. To treat such cases, we added a simple functionality to perform division onto initial arguments depending on the difference of their lengths to the above three algorithms. Note that in any of the above, 2's factors are treated separately and removed from the initial arguments.

The bmod algorithm is used to remove a spurious factor in the accelerated algorithm by computing

$$\gcd(\gcd(X, C_n), Y)$$

for the final C_n. If the initial remainder calculation $Z \Leftarrow X \bmod Y$ is performed, where $X > Y$, this Z is used in the above GCD calculation instead of Y. The initial remainder calculation may be performed also in these GCD calculations.

3.1. Technicalities for faster execution

In order to make the implemented code execute as fast as possible, various techniques are investigated and actually used. They are as follows.

- Memory space for working area to be used for storing intermediate expressions is allocated at the invocation time of each algorithm and is recycled. Localized memory access can expect the effect of memory caching.
- Following Weber's implementation [16], the sequence of multiplication by single-word integers, addition or subtraction and shifting of long integers is realized by one-time scan of the words of long integers in order to reduce the number of slow memory accesses, except when the result of the subtraction turned out to be negative, in which case once more scan of the result is required for negation.

- In every algorithm, the right-shift binary algorithm is used after a pair of integers are both reduced to within the magnitude of double-words.
- In the binary algorithm, the difference calculation of two long integers begins with the equality check starting from the most significant words, and at the time when unequal words are found and the sign of the result is fixed, changes into the true subtraction operation starting from the least significant words. Therefore, in contrast to the notice made by Jebelean at the beginning of [2], this calculation can be done only with one time scan.
- As mentioned earlier, $u/v \bmod 2^m$ can be calculated in two ways; application of the MODIV algorithm with $\beta = 2$ and the calculation of the remainder sequence of 2^m and v. Both algorithms are tested on processors with (PowerPC) and without (SuperSparc) instructions for integer multiplication and division. Our test result indicates that the former is slightly faster than the latter, which agrees with Jebelean's result [1], on both processors. However, this difference is negligible when very long integers are treated.

3.2. User interface

The following `igcd`*xxx* functions are implemented, in addition to the original `igcd`.

function name	algorithm used
`igcdeuc`	Euclid's algorithm, Risa/Asir's original implementation
`igcdbin`	right-shift binary algorithm
`igcdbmod`	bmod algorithm
`igcdacc`	accelerated (generalized-binary) algorithm

Furthermore, in order to give a facility to switch the choice of an algorithm when called via the original `igcd` function or internally from Risa, three internal variables are prepared and a control function `igcdcntl([`n`])`, which performs assignment to those variables, is implemented. Its specification follows[1)].

[1)] The name of the function is changed from `igcdhow` and more functionalities are added since the report [5].

argument	functionality
`"euc"` or 0	specifies the use of Euclid's algorithm.
`"bin"` or 1	specifies the use of the right-shift binary algorithm.
`"bmod"` or 2	specifies the use of the bmod algorithm.
`"acc"` or 3, ...	specifies the use of the accelerated algorithm.
none	returns the current choice of an algorithm as one of the above integer values.
$(-div$ $-10000 * acc)$	given negative integer assigns each of the following thresholds when $\neq 0$: *div*: threshold for the initial remainder calculation; if the ratio of the lengths (the number of words) of two arguments is $\geq div$, the initial remainder calculation is to be performed. *acc*: threshold used in the accelerated algorithm to switch the elimination method; if the difference of the bit lengths of two integers is $\geq acc$, the bmod operation is used instead of (6). The default values of *div* and *acc* are 50 and 10, respectively.
`[]`	returns an integer $-div-10000*acc$ for the current values of *div* and *acc*.

4. Empirical studies

Our new implementation is tested on the following two platforms.

- SparcStation 20 model 61 (SuperSparc@60MHz. SPECint92: 98.2) + SunOS 4.1.4 + gcc-2.7.2 (`-O2`)
- IBM RS6000 43P/133 (PowerPC 604@133MHz. SPECint92: 176.4 and SPECint95: 4.72) + AIX 4.1 + cc (`-O3`)

While the former processor does not implement instructions for integer multiplication and division, the latter does. To see the effect of this difference and to compare the algorithms, we have tested various sample problems. The integers used are the products of random numbers generated by the `random()` function, each of which fits in 27-bit width, the word size in Risa/Asir. Tables below summarizes the average timings (in milliseconds, except

L_{in}	5	10	20	50	100	200	500	1000	2000
$\bar{l}_{in}$	1.006	1.563	3.196	9.295	21.34	48.86	143.7	323	713
N	10000	10000	10000	10000	1000	1000	100	50	10
SparcStation 20/61									
Euclid	1.089	2.974	9.212	47.755	178.55	680.61	4,177.6	16,378.0	62,518
binary	0.196	0.418	1.117	4.962	17.96	66.82	397.0	1,536.8	6,072
bmod	0.263	0.652	1.659	6.437	21.04	74.60	426.0	1,644.0	6,424
$acc = 6$	0.265	0.517	1.123	3.475	10.29	34.26	190.9	731.8	2,862
$acc = 10$	0.313	0.524	1.113	3.542	10.39	34.30	189.3	722.8	2,849
$acc = 20$	0.297	0.552	1.157	3.670	10.59	35.27	193.2	736.8	2,885
IBM RS6000 43P/133									
Euclid	0.515	1.386	4.183	20.943	75.71	286.64	1,720.3	6,720.4	26,814
binary	0.125	0.251	0.593	2.326	7.65	26.49	150.9	571.0	2,462
bmod	0.158	0.317	0.788	2.908	8.80	30.36	170.8	649.0	2,583
$acc = 6$	0.166	0.257	0.566	1.635	4.36	14.04	73.8	275.8	1,096
$acc = 10$	0.158	0.337	0.595	1.693	4.33	13.79	71.6	273.4	1,089
$acc = 20$	0.192	0.337	0.600	1.675	4.58	13.82	74.1	276.6	1,102

Table 1. Timings when two arguments are of equivalent lengths

when noted otherwise) of N-times calls to `igcd`xxx, and the following notations are used in the tables.

L_{in}: the number of random numbers used to make one argument
$\approx$ the number of words of an argument

$\bar{l}_{in}$: the average number of words of argument(s) after 2's factors are removed, almost 90% of L_{in}

$\bar{l}_G$: the average number of words of the GCD

N: the number of samples tried

4.1. Dependence on the lengths of bignum arguments

Cases of arguments with equivalent lengths

The first examples are with two arguments of equivalent lengths. Table 1 summarizes the timings. With the accelerated algorithm, timings are taken for three different values of acc to see the dependence of computing times on acc, but no particular difference can be observed. Also, the table indicates that the accelerated algorithm is most efficient.

Cases of arguments with different lengths

Table 2 shows the timings when the lengths of two arguments are different. We have mea-

sured the cases with and without initial remainder calculation for every sample problem, and the table contains the timings of both cases in each row of an algorithm; the upper is without division while the lower is with division. The accelerated algorithm is used with $acc = 10$ (default). Our observation follows.

- When the lengths of two arguments are different, the use of the binary algorithm should be preceded by the initial remainder calculation between them.
- The timings of the bmod algorithm with and without initial remainder calculation indicates that, as is concerned with the implementation in Risa/Asir, the bmod operation is slightly faster than the remainder calculation when the lengths of arguments are not very different (ratio $\lesssim$10, despite their similarity.
- We may observe a similar tendency with the accelerated algorithm also using the bmod operation without the initial remainder calculation. Recall that the remainder calculation affects on the speed of the post-GCD calculation for spurious factor removal, which explains why the bmod algorithm is faster than the accelerated one if the remainder calculation is not performed initially.

4.2. Application to more practical problems

As pointed out by Neun and Melenk [6], it is very often with Gröbner basis calculation for solving a system of algebraic equations that coefficients of the intermediate expressions grow enormously and most of the computing times are spent by the calculations of integer coefficients. Although the recent development of the modular plus trace-lifting algorithm has improved the Gröbner basis calculation very much, the computation of long integers, especially in the reductions to normal forms, can still be a neck of computation, as was reported by Noro and Yokoyama [10]. To see how our implementation affects on or improves the efficiency of this type of calculation, we have tested the following problem of more practical significance.

```
dp_gr_flags(["Multiple",2]);
dp_gr_main( cyclic(n),[c0,c1,...,c_{n-1}],1,1,0);
```

(DRL order with homogenization and calculation with the modular images)

Because in the last report [5], there can be observed no particular distinction in timings among the different algorithms when $n = 5$ and $n = 6$, we have tried only the case when $n =$

(i) Cases when one of the arguments is the product of 10 random numbers

L_{in}	20	50	100	200	500	1000	2000
$\bar{l}_{in}$	9.682	9.682	9.681	9.681	9.67	9.68	9.7
	18.725	45.976	91.456	182.438	455.19	909.99	1819.3
$\bar{l}_G$	2.082	2.615	2.988	3.349	3.84	4.22	4.7
N	20000	20000	20000	10000	2000	1000	200
SparcStation 20/61							
Euclid	3.165	3.622	4.528	6.531	12.07	21.32	40.2
binary	0.821	2.682	8.195	27.555	154.37	594.54	2,316.6
	0.653	1.167	2.635	3.770	8.92	17.28	33.9
bmod	0.722	0.998	1.394	2.866	11.60	39.68	145.2
	0.909	1.387	2.236	3.955	8.85	17.20	34.4
acc	0.655	0.997	1.923	4.625	21.49	76.67	288.8
	0.913	1.566	2.381	4.059	9.08	17.05	33.9
IBM RS6000 43P/133							
Euclid	0.530	0.616	0.707	1.061	1.73	3.11	5.6
binary	0.389	1.191	3.700	12.526	69.15	264.33	1,034.3
	0.315	0.417	0.641	1.067	2.23	4.01	6.8
bmod	0.189	0.304	0.469	1.037	4.41	15.23	58.1
	0.415	0.499	0.665	0.969	1.82	3.45	6.6
acc	0.232	0.360	0.644	1.589	7.83	28.49	109.1
	0.376	0.483	0.680	1.105	2.12	3.67	6.9

(ii) Cases when one of the arguments is the product of 100, (iii) 200 and (iv) 1000 random numbers

L_{in}	100				200			1000
	200	500	1000	2000	500	1000	2000	2000
$\bar{l}_{in}$	91.44	91.42	91.50	91.4	182.3	182.4	182.6	910.5
	182.41	455.16	910.07	1819.5	455.3	909.6	1819.3	1819.3
$\bar{l}_G$	27.79	33.46	37.39	41.1	65.1	73.9	81.4	394
N	1000	1000	1000	100	200	100	100	20
SparcStation 20/61								
Euclid	175.45	185.15	206.59	257.5	676.8	697.4	771.6	14,914
binary	47.22	206.02	695.57	2,540.8	260.0	807.3	2,762.3	4,393
	22.75	37.18	62.31	113.0	90.6	133.1	220.5	1,814
bmod	24.20	40.44	80.78	217.7	96.5	150.2	311.2	1,937
	25.22	39.70	64.91	115.9	97.4	139.0	225.9	1,907
acc	16.41	43.66	117.44	372.0	73.0	167.2	461.8	1,336
	16.66	31.94	57.09	107.5	62.4	107.4	195.3	1,083
IBM RS6000 43P/133								
Euclid	74.62	77.40	84.62	100.3	282.5	292.3	318.3	6,395
binary	19.60	86.46	299.91	1,108.5	103.7	335.0	1,179.8	1,766
	9.31	14.31	23.08	40.4	35.9	51.8	85.7	703
bmod	10.11	16.57	32.47	85.2	38.7	59.5	122.9	774
	10.28	15.70	24.09	41.0	39.3	55.9	88.8	776
acc	7.03	17.75	46.95	146.9	28.9	65.7	181.9	518
	6.77	11.77	20.50	38.6	24.3	40.9	75.1	433

Table 2. Timings when the lengths of two arguments are different

7. The following table shows the timings (in seconds) measured on IBM RS6000 43P/133.

algorithm used	*div*	CPU time	+	GC
Euclid	-	8046	+	980.2
binary	50	14940	+	988.5
	1	7698	+	960.5
bmod	50	7474	+	993.7
acc	50	7525	+	1051.0
	2	7527	+	1003.0

We can observe a little bit improvement. More successful result is reported by Noro and McKay [8]. They applied our new implementation to the integer content removal in the calculation of normal forms for a large-scale Gröbner basis, and our implementation has had a big effect on the improvement of the computing time in cooperation with their improved method for integer contents [7].

5. Conclusion

With our new implementation in Risa/Asir, we have tested some of the algorithms newer than the Euclidean. When integer arguments are lengthy and of equivalent lengths, the latest algorithm, called accelerated or generalized binary, using a generalized sequence (6) is most efficient in practice, which agrees with the previous empirical results by the developers of the algorithm. Even in the cases that the lengths are very different, we can make the arguments have equivalent lengths by initially performing the remainder calculation between them, and then apply any of the newer algorithms if the reduced arguments are still lengthy, for speed. For more general cases, it is difficult to determine the most efficient way, e.g., how large or how different the lengths of arguments should be to perform the initial remainder calculation, although computing times will fluctuate very little in most cases, as practical applications are concerned.

Anyway, the improvement due to our new implementation of the newer algorithms is clear, and the aim of our attempt has been successfully achieved. Finally, the author would like to stress the need for the use of the latest algorithm, e.g., bmod or accelerated, as default.

References

[1] Jebelean, T.: An algorithm for exact division, *Journal of Symbolic Computation*, **15**(2), 1993, 169–180.

[2] Jebelean, T.: A generalization of the binary GCD algorithm, *Proceedings of ISSAC '94* (von zur Gathen, J. and Giesbrecht, M., eds.), Oxford, England, 1994, 111–116.

[3] Jebelean, T.: A double-digit Lehmer-Euclid algorithm for finding the GCD of long integers, *Journal of Symbolic Computation*, **19**(1–3), 1995, 145–157.

[4] Lehmer, D. H.: Euclid's algorithm for large numbers, *American Mathematical Monthly*, **45**, 1938, 227–233.

[5] Murao, H.: Improving the integer GCD calculation in Risa/Asir, (in Japanese). Workshop on Computer Algebra system and its application (2nd Risa Consortium held at Ehime U. on Mar. 14∼16, 1996).

[6] Neun, W. and Melenk, H.: Very large Gröbner basis calculations, *CAP '90: Computer Algebra and Parallelism, Second International Workshop Proceedings* (Zippel, R. E., ed.), *LNCS*, 584, Ithaca, USA, Springer-Verlag, 1990, 89–99.

[7] Noro, M.: Efficient removal of the content of an integral vector and its application, (in these proceedings).

[8] Noro, M. and McKay, J.: Computation of replicable functions on Risa/Asir, (to appear in PASCO'97), 1997.

[9] Noro, M. and Takeshima, T.: Risa/Asir — a computer algebra system, *Proceedings of ISSAC '92* (Wang, P. S., ed.), Berkeley, CA, 1992, 387–396.

[10] Noro, M. and Yokoyama, K.: New methods for the change-of-ordering in Gröbner basis computation, Technical report, Institute for Social Information Science, Fujitsu Laboratories Ltd., 1995.

[11] Sorenson, J.: Two fast GCD algorithms, *Journal of Algorithms*, **16**, 1994, 110–144.

[12] Stein, J.: Computational problems associated with Racah algebra, *Journal of Computational Physics*, **1**, 1967, 397–405.

[13] Wang, P. S.: A p-adic algorithm for univariate partial fractions, *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation* (Wang, P. S., ed.), Snowbird, Utah, 1981, 212–217.

[14] Weber, K.: Parallel implementation of the accelerated integer GCD algorithm, *Proceedings of PASCO '94* (Hong, H., ed.), Linz, Austria, 1994, 405–411.

[15] Weber, K.: *Parallel integer GCD algorithms and their application to polynomial GCD*, PhD thesis, Dept. of Mathematics and Computer Science, Kent State Univ., Ohio, 1994.

[16] Weber, K.: The accelerated integer GCD algorithm, *ACM Transactions on Mathematical Software*, **21**(1), 1995, 111–122.