Implementing the Thull-Yap algorithm for computing Euclidean remainder sequences

François Morain

Implementing the Thull-Yap Algorithm for Computing Euclidean Remainder Sequences

François MORAIN
LIX, CNRS, INRIA, École Polytechnique, Institut Polytechnique de Paris
F-91120 Palaiseau, France
morain@lix.polytechnique.fr

ABSTRACT

There are two types of integer gcd algorithms: those which compute the sequence of remainders of Euclid's algorithm and those which build different sequences. The former are more difficult to validate and analyse, whereas the latter are simpler and more efficient. When one wants the euclidean remainders (for instance if one wants to compute continued fractions), only the former can be used. Our main focus is the subquadratic time Thull-Yap GCD algorithm, and in fact on its core computing a half gcd (TYHGCD). This algorithm is tricky due to the difficulty in correcting the remainder sequence that comes back from a recursive call.

The aim of this work is to revise TYHGCD in order to implement it using GMP. We clarify some points of the algorithm, in particular the stopping conditions that are always difficult to set correctly. We add a base case to speed up the whole algorithm, using Jebelean's quadratic algorithm with a stopping condition. We give our own modified version and add the proofs where needed. We insist on the test phase for this algorithm, giving families of hard cases for all branches, some of which are rarely activated. We give some details on our implementation in GMP using low-level functions, adding some remarks on the use of fast multiplications techniques. We pay attention to the data structure needed to store partial quotients, enabling to navigate rapidly back and forth in the sequence of Euclidean remainders. Benchmarks are provided. Some comments are made on Lichtblau's algorithm, which is close in spirit to the Thull-Yap algorithm.

CCS CONCEPTS

• **Computing methodologies**; • **Mathematics of computing** → **Number-theoretic computations**;

KEYWORDS

Integer gcd, subquadratic arithmetic

ACM Reference Format:
François MORAIN. 2022. Implementing the Thull-Yap Algorithm for Computing Euclidean Remainder Sequences. In *Proceedings of the 2022 International Symposium on Symbolic and Algebraic Computation (ISSAC '22), July 4–7, 2022, Villeneuve-d'Ascq, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3476446.3536188

ISSAC '22, July 4–7, 2022, Villeneuve-d'Ascq, France

ACM ISBN 978-1-4503-8688-3/22/07…$15.00
https://doi.org/10.1145/3476446.3536188

1 INTRODUCTION

Computing the gcd of two integers is one of the basic tasks required in any multi-precision integer package or mathematical system. It is also one of the oldest algorithms ever. Optimizations of Euclid's algorithm on computers were developed over the years, improving the quadratic complexity. This includes [11] and Collins (around 1980, reference being lost?), followed by [8, 9]. Subquadratic methods began their (rather chaotic) history with Knuth, followed by [10, 15, 22] (see also [1] for an uncomplete polynomial version). A refined algorithm was designed and very precisely described in [26] (see also [29]), unifying integer and polynomial gcd's, using norms. In [12], another version is given using a more direct approach. It is very important to note that all these methods compute the sequence of euclidean quotients of the classical algorithm, which is interesting for the fast evaluation of continued fractions. These methods are generally believed to be difficult to implement, due to the many sub-cases involved and the fact they are rarely used.

If we do not insist on computing these euclidean remainders but aim at the computation of the gcd only, alternative methods exist, that are simpler and faster. This includes the binary versions: [5, 23, 25, 27] for quadratic ones and [3, 16, 24] for the subquadratic ones. We remark that these algorithms do not compute the Euclidean remainders.

Our work was originally motivated by writing a fast implementation of Cornacchia's algorithm (see Section 4) for primality proving [17]. We rapidly concentrated on the implementation of the Thull-Yap algorithm [26, 29], since the approach is very close to what is needed in a GMP implementation. Note that Lichtblau's algorithm [12] offers an interesting alternative with many common points with the preceding algorithm, but the original article is not as close to an implementation as we thought (see the forthcoming [18]). An algorithm for the dual problem of finding regular matrices (see below) satisfying a given bound is given in [20, Algorithm 5.9], whose application to our problem needs further investigation.

The content of our article is as follows: we recall basic facts about Euclid's algorithm in Section 2, introducing our problem on computing remainders satisfying some bound conditions. Section 3 is concerned with the Thull-Yap algorithm and our revisiting thereof, adding some base-cases and instructions missing in the original work. Also, we take care to data structures used in the algorithm, notably manipulating at the same time a quotient sequence together with its matrix representation. Section 4 explains how this algorithm can be used in Cornacchia's algorithm. In Section 5, we devote our time to building families of numbers establishing covering tests of the Thull-Yap algorithm. Finally, Section 6 gives some details on our implementation in GMP[7] and benchmarks.

Notations: we denote by $\mathrm{M}(n)$ the complexity of multiplication of two n-bit integers. We will operate on integers represented in base $\mathbb{B} = 2^\beta$ for some $\beta \geq 1$, typically the size of the machine words on a processor (32 and 64). If $a > 0$ is a real number, we put $\|a\|_\mathbb{B} = \log_\mathbb{B} |a| = (\log_2 |a|)/\beta$, and by convention, $\|0\|_\mathbb{B} = 0$. We use $\|.\|$ to designate the norm $\|.\|_\mathbb{B}$ to ease notations.

2 EUCLID'S ALGORITHM AND VARIANTS

Since a large part of our work relies on the Thull-Yap algorithm, we use some of the notations in [29], which builds on [26] with (small) differences.

2.1 Basic properties

Let $a > b > 0$ be two integers. Introduce the *Euclidean quotient sequence* (q_i) and *Euclidean remainder sequence* (r_i) defined by using Euclidean division and remainder on successive pairs of numbers:

$$\begin{aligned} a &= bq_1 + r_1, && 0 \leq r_1 < b \\ b &= r_1 q_2 + r_2, && 0 \leq r_2 < r_1 \\ &\cdots \\ r_i &= r_{i+1} q_{i+2} + r_{i+2}, && 0 \leq r_{i+2} < r_{i+1} \end{aligned}$$

We denote by $a \div b$ the quotient of a by b. The sequence (r_i) is decreasing. We denote by k the smallest index for which $r_k = 0$ and remark that $r_{k-1} = \gcd(a, b)$.

Remember the classical Lemma:

Lemma 2.1. *Perform the Euclidean algorithm on $a > b$ with the usual notations. Then $r_2 < b/2$.*

Proof: if $r_1 \leq b/2$, we are done. Otherwise, $b/2 < r_1 < b$ implies $q_2 = 1$, and $r_2 = b - r_1 < b/2$. □

In other words, we are sure to decrease the size of the remainders by 1/2 bit per iteration at worse. As a consequence the length of the sequences is bounded by $2 \log b / \log 2 + 1$.

2.2 Properties of regular matrices

One important formalism is the representation of Euclid's operations using 2×2 matrices. The identity matrix is denoted by E. The Euclidean algorithm produces a sequence of matrices of the form

$$Q = \begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$$

where $q \geq 1$ is an integer; we call such a matrix *elementary*. In order to write compact expressions, we write $Q = [[q, 1], [1, 0]]$ as an array of two row vectors; a column vector is written (a, b). Typically, q will be some Euclidean quotient and the operation $(a, b) = Q(a', b')$ is one step of the Euclidean algorithm: $a = bq + r$ with $0 \leq r < b$, so that $a = a'q + b'$ and $b = a'$.

A *regular* matrix is $M = Q_1 \cdots Q_k$ with $k \geq 0$ (when $k = 0$, $M = E$ and E is considered regular too); an elementary matrix is therefore regular. A regular matrix enjoys the following size property.

Proposition 2.2. *If $M \neq E$ is regular*

$$M = \begin{pmatrix} p & q \\ r & s \end{pmatrix}$$

then M satisfies the ordering property

$$\|p\| \geq \max\{\|q\|, \|r\|\} \geq \min\{\|q\|, \|r\|\} \geq \|s\|, \quad \|p\| > \|s\|. \quad (1)$$

The following Proposition is crucial in the design of fast Euclidean remainders computation.

Proposition 2.3. *Given a, b, a', b' s.t. $\|a\| > \|b\| \geq 0$. The following are equivalent.*
(i) a', b' are consecutive elements in an Euclidean remainder sequence of a, b.
(ii) There is a regular matrix M s.t. $(a, b) = M(a', b')$ and $a' > b' > 0$.

Other gcd algorithms yield a transition matrix M that may not be regular.

For practical reasons, we decided to introduce a *regular pair* consisting of a list of elementary matrices and their product: $\mathcal{M} = (Q, M)$. We write $Q = \langle q_1, q_2, \ldots, q_k \rangle$ for the list and the corresponding matrix is $M = \prod_{i=1}^{k} [[q_i, 1], [1, 0]]$. By reference to the basic Euclid algorithm, we obtain

$$(a, b) = Q_1 \cdots Q_k (r_{k-1}, 0) = \langle q_1, \ldots, q_k \rangle (r_{k-1}, 0) = M(r_{k-1}, 0).$$

Adding a new matrix $[[q', 1], [1, 0]]$ to a sequence Q will be denoted $Q \oplus \langle q' \rangle$ for short. We designate by BuildMatrixFromQ the operation of computing M from Q, which we need. A fast implementation of this operation is described in Section 6.4.

We define some operations on regular pairs, that consist of adding or removing entries at the end of the sequence in a LIFO manner. These operations are used in the Thull-Yap algorithm to correct the approximations of the quotients found during a recursive call. Suppose that $(a, b) = M(a', b')$ for some regular pair $\mathcal{M} = (\langle q_1, q_2, \ldots, q_k \rangle, M)$.

(I) Advancing: we add the quotient $q' = a' \div b'$ at the end of $\mathcal{M}$, an operation we note $\mathcal{M} \otimes \langle q' \rangle$ producing $(Q \otimes \langle q' \rangle, M \cdot [[q', 1], [1, 0]])$. This yields a regular matrix when $q' \geq 1$ (i.e., in the case $\|a'\| \geq \|b'\|$). We may advance by more steps.

(II) Backing up: we call the matrix $\langle q_1, \ldots, q_{k-i} \rangle$ the *backing up* of M by i steps ($0 \leq i \leq k$); using a', b' and M, we can back up to the desired remainders. We denote this by $\mathcal{M} \oslash \langle q_{k-i+1}, \ldots, q_k \rangle$.

(III) Toggling: put $T = [[1, 1], [0, -1]]$ and remark that $MTT = M$. This yields $(a, b) = MT(a' + b', -b')$. If $q_k > 1$, then

$$MT = \langle q_1, q_2, \ldots, q_{k-1}, q_k - 1, 1 \rangle.$$

If $q_k = 1$ and $k > 1$, we get $MT = \langle q_1, q_2, \ldots, q_{k-2}, q_{k-1} + 1 \rangle$; in both cases, MT is regular. When $q_k = 1$ and $k = 1$, MT is not regular.

Also, if $\mathcal{M} = (Q, M)$ and $\mathcal{M}' = (Q', M')$ are two regular pairs, we join them as $\mathcal{M} \otimes \mathcal{M}' = (Q \otimes Q', M \cdot M')$.

2.3 Partial remainders

We can formulate a partial Euclid algorithm taking as input a pair of integers (a, b) and a real number $0 < \rho \leq \|a\|$. Algorithm PartialRemainder(a, b, ρ) outputs the unique pair of Euclidean remainders satisfying $\|r_i\| \geq \rho > \|r_{i+1}\|$. It is described in Algorithm 1.

We can make a similar modification of Jebelean's algorithm [9, §4.2] for the same goal (PartialJebelean). In the latter case, we use Jebelean's approach to decrease to the target ρ and we finish

Function PartialEuclid(a, b, ρ)

Input : $a > b \geq 0$ two integers, ρ a real number such that $\|a\| \geq \rho > 0$

Output : a', b', and a regular pair $\mathcal{M} = (Q, M)$ with $\|a'\| \geq \rho > \|b'\|$

1. $Q \leftarrow \emptyset$;
2. **while** $\|b\| \geq \rho$ **do**
   $(q, r) \leftarrow (a \div b, a \bmod b)$;
   $a, b \leftarrow b, r$;
   $Q \leftarrow Q \otimes \langle q \rangle$;
   end
3. **return** $a, b, (Q, BuildMatrixFromQ(Q))$;

Algorithm 1: The partial Euclid algorithm.

with PartialEuclid. Our aim is to show how to use the Thull-Yap algorithm for the same problem.

2.4 The road to sub-quadratic algorithms

The common idea about fast gcd algorithms is the following. Use a divide-and-conquer approach to the problem, using a subroutine that computes "half the gcd" and generally called HGCD. Typically, two recursive calls are performed in such a function, and some correction may have to be done to recover the correct answer. For two n-bit integers, the cost of HGCD is $O(\log n \mathrm{M}(n))$ complexity, and the complexity of the whole GCD is the same.

3 THE THULL-YAP ALGORITHM

We assume the reader is familiar with the algorithm given in [26] and [29, Chapter 2, Appendix A]. We use the same notations except on some minor points. A line by line inspection of the original algorithm given in the reference reveals that we can make it work on β-bit numbers for any $\beta \geq 1$. There are only a few points in the proofs where this really matters and we concentrate on them.

3.1 Norms and bounds

The following quantity (the *magic threshold*) is crucial in the Thull-Yap algorithm:

$$\mathbb{T}_{\mathbb{B}}(a) = 1 + \left\lceil \frac{\|a\|_{\mathbb{B}}}{2} \right\rceil,$$

since one call to the procedure returns a pair of integers (a', b') with $\|a'\|_{\mathbb{B}} \geq \mathbb{T}_{\mathbb{B}}(a) > \|b'\|_{\mathbb{B}}$, ready for the next recursive call and giving the sub-quadratic running time.

Analyzing the algorithm further, we see that it uses the quantities $a_0 = 1 + (a \div \mathbb{B}^m)$ with $m = \mathbb{T}_{\mathbb{B}}(a)$ and also $t = \mathbb{T}_{\mathbb{B}}(a_0)$. It will be required to have $\|a\|_{\mathbb{B}} \geq \mathbb{T}_{\mathbb{B}}(a) + \mathbb{T}_{\mathbb{B}}(a_0)$ to use in the Fixup procedure (see below again).

Proposition 3.1. *(i) if $\|a\|_{\mathbb{B}} \geq 3$, then $\|a\|_{\mathbb{B}} \geq \mathbb{T}_{\mathbb{B}}(a)$;*
(ii) for each $\mathbb{B}$, there exists a constant $\mathcal{A}_{\mathbb{B}}$ such that if $\|a\|_{\mathbb{B}} \geq \mathcal{A}_{\mathbb{B}}$, then $\|a\|_{\mathbb{B}} \geq \mathbb{T}_{\mathbb{B}}(a) + \mathbb{T}_{\mathbb{B}}(a_0)$.

Proof: Write $a = \mathbb{B}^{\alpha}$, where α is a positive real number. Start from

$$m = 1 + \left\lceil \frac{\alpha}{2} \right\rceil, a_0 = 1 + \left\lfloor \mathbb{B}^{\alpha - m} \right\rfloor, \|a_0\|_{\mathbb{B}} = \alpha_0, t = 1 + \left\lceil \frac{\alpha_0}{2} \right\rceil,$$

When α is large, one has $m \approx \alpha/2$ and $t \approx m/2 \approx \alpha/4$, so that $\alpha \geq m + t$ is trivially true. It remains to see what happens when α is small.

(i) If $\alpha \geq 4$, we have $\alpha \geq 1 + (\alpha/2 + 1) \geq 1 + \lceil \alpha/2 \rceil$. If $2 < \alpha < 4$, we have $\lceil \alpha/2 \rceil = 2$ and we need $\alpha \geq 3$ to meet the bound.

(ii) We want

$$\alpha \geq 2 + \left\lceil \frac{\alpha}{2} \right\rceil + \left\lceil \frac{\alpha_0}{2} \right\rceil.$$

We have

$$\mathbb{B}^{\alpha - m} < \mathbb{B}^{\alpha_0} \leq 1 + \mathbb{B}^{\alpha - m}.$$

Taking logs and using $\log(1 + x) \leq x$, we get

$$\alpha - m < \alpha_0 \leq \alpha - m + \frac{1}{\mathbb{B}^{\alpha - m}}.$$

Since $m \leq 2 + \alpha/2$, one has $\alpha - m \geq \alpha/2 - 2$ and for $\alpha \geq 10$, we get

$$\alpha_0 \leq \alpha - m + \frac{1}{\mathbb{B}^3}.$$

To get our result, it is enough to have

$$\alpha \geq 2 + \left\lceil \frac{\alpha}{2} \right\rceil + \left\lceil \frac{\alpha - m + 1/\mathbb{B}^3}{2} \right\rceil,$$

a relation satisfied as soon as $\alpha \geq 10$. □

Remark. By numerical inspection, $\alpha \geq 9$ is best possible in case ii). We can content ourselves by taking a uniform value for $\mathcal{A}_{\mathbb{B}}$, namely $\mathcal{A} = 10$ and we do that in the remaining part of the work. For small values of $\mathbb{B}$, we may find a better value.

3.2 The Thull-Yap algorithm in a nutshell

3.2.1 Outline. The idea of the ThullYapHGCD algorithm on $a > b > 0$ is to compute a regular matrix M such that $(a, b) = M(a', b')$ where

$$\begin{array}{l} \text{if } \|a\| < 2, M = E; \\ \text{if } \|a\| \geq 2, \|a'\| \geq \mathbb{T}(a) > \|b'\|, \text{with } a' > b' \geq 0. \end{array} \tag{2}$$

The first case covers rare cases or cases where a is small (see below; to simplify, we replace 3/2 by 2). In the second case (which is the most frequent), the size of the new entry (a', b') is roughly half that of (a, b), so that we end up with a divide-and-conquer approach to computing $\gcd(a, b)$. Note that the case $\mathbb{T}(a) < \mathcal{A}$ needs a special treatment.

To compute M, the algorithm considers the top half a_0 (resp. b_0) of a (resp. b) and proceeds to find the Euclidean quotients of (a_0, b_0). To recover the correct sequence of quotients for (a, b), procedure Fixup is applied (see below). The full Thull-Yap algorithm uses the ThullYapHGCD algorithm to get down to $\gcd(a, b)$ recursively.

3.2.2 The Basic Setup. Suppose $a > b > 0$ and $a \geq \mathbb{B}^m$. Define $a_0 = 1 + (a \div \mathbb{B}^m)$, $b_0 = b \div \mathbb{B}^m$, $b_1 = b \bmod \mathbb{B}^m$, and $a_1 = a_0 \mathbb{B}^m - a$ so that

$$(a, b) = [[a_0, -a_1], [b_0, b_1]](\mathbb{B}^m, 1)$$

and $0 < a_1 \leq \mathbb{B}^m$. This definition of a_0 is chosen to be sure to have $a_0 > b_0$ in a recursive call. It makes life much easier when updating the numbers using matrices, see Lemma 3.2. The algorithm computes a regular matrix M for (a_0, b_0) and we compute

$$[[a_0', a_1'], [b_0', b_1']] = M^{-1}[[a_0, -a_1], [b_0, b_1]]$$

from which $(a', b') = [[a_0', a_1'], [b_0', b_1']](\mathbb{B}^m, 1)$.

The following Lemma will prove useful when updating the values of a and b, since it boils down to multiplying integers that are all > 0. Its proof is left to the reader.

Lemma 3.2. *Let $M = [[p, q], [r, s]]$. Write*

$$M^{-1}(a, b) = (a', b') = (a'_0\mathbb{B}^m - a'_1, b'_0\mathbb{B}^m + b'_1)$$

with $\delta = \det(M) = \pm 1$. Then

$$a'_1 = \delta(sa_1 + qb_1), \quad b'_1 = \delta(ra_1 + pb_1).$$

3.2.3 Correcting the quotient sequence when $M \neq E$. Given the quotient sequence Q for (a', b') we want the correct sequence Q^* for (a, b) with new starting point $(a^*, b^*) = M^{*-1}(a, b)$ such that

$$\|a^*\| \geq \mathbb{T}(a) > \|b^*\|, \quad a^* > b^* \geq 0. \tag{3}$$

We may have two problems:

Case $\det(M) = -1$: b' can be negative.

Case $\det(M) = +1$: an inversion $b' \geq a'$ may occur.

We slightly adapt the following Lemma of [26] to our needs.

Lemma 3.3 (Fixing up). *Let t be any number such that*

$$\|a'_0\| \geq t > \max\{\|b'_0\|, \|a_0\| - \|a'_0\| + 1\}. \tag{4}$$

Moreover, put $Q = \langle q_1, \ldots, q_k\rangle$. We explain how to deduce Q^* from Q s.t. a^* and b^* (deduced from a', b' with $(a, b) = M(a', b')$) satisfy

$$\|a^*\| \geq m + t > \|b^*\|, \tag{5}$$

$$b^* \geq 0. \tag{6}$$

(-) Suppose $\det(M) = -1$.

(-A) If $b' \geq 0$ then $Q^* = Q$.

(-B) Else if $\|a' + b'\| \geq m + t$, then Q^* is the toggle of Q.

(-C) Else if $q_k \geq 2$ then $Q^* = \langle q_1, \ldots, q_{k-1}, q_k - 1\rangle$ is the backup of the toggle of Q.

(-D) Else Q^* is the backing up of Q by two steps.

(+) Suppose $\det(M) = +1$.

(+A) If $\|a'\| \leq \|b'\|$ then Q^* is the advancement of $\langle q_1, q_2, \ldots, q_{k-1}\rangle$ by at most 2β steps.

(+B) Else if $\|a'\| < m + t$ then Q^* is the backing up of Q by one or two steps.

(+C) Else Q^* is the advancement of Q by at most 2β steps.

Proof: this is the same as in [26], but for the cases where we need adaptation to the case $\mathbb{B} = 2^\beta > 2$.

The case (+B): Write

$$[a_0; b_0] = [[p', q'], [r', s']]\ [[q_k, 1], [1, 0]]\ [a'_0; b'_0]$$

leading to

$$z' = q_k a' + b' = (a'_0 + q_k b'_0)\mathbb{B}^n + s'a_1 + q'b_1 \geq z'_0\mathbb{B}^n.$$

We have $\|z'\| \geq \|z'_0\| + m \geq \|a'_0\| + m \geq m + t$ and $m + t > \|a'\|$, so that one backup is enough, and not two as given in [26]. □

This correction Lemma is used as Algorithm 2.

Function Fixup$(a', b', \mathcal{M} = (Q, M), m, t)$
Input : See notations of Lemma 3.3
Output : a^*, b^*, a regular pair $\mathcal{M}^* = (Q^*, M^*)$ such that $(a^*, b^*) = M^*(a, b)$ with $\|a^*\| \geq m + t > \|b^*\|$.
Algorithm 2: The Fixup function.

3.2.4 Correcting the quotient sequence when $M = E$. We use the same notations as above. This case happens when we start the algorithm with a pair (a, b) for which $\|a_0\| < \mathcal{A}$ where $a_0 = 1 + (a \div \mathbb{B}^m)$. Our goal is to find a quotient sequence leading to (a', b') and $\|a'\| \geq m > \|b'\|$. From $a_0\mathbb{B}^m > a$, we deduce $\mathcal{A} + m > \|a_0\| + m > \|a\|$. We have to compute a quotient sequence leading to (a', b') with $\|a'\| \geq m$ the number of steps for this is bounded as a function of $\mathcal{A}$, that is a constant. Procedure Fixup$_0$ follows easily: use PartialEuclid (or PartialJebelean) to reach the desired value of (a', b').

3.2.5 The algorithm and partial correctness. We give a version of ThullYapHGCD that includes some corrections and is closer to a real implementation (for instance, returning $a', b', \mathcal{M} = (Q, M)$ instead of M alone), compare with [26, Section 5]. We also add the necessary modifications to procedure Fixup. Very classically, we also use a base case using Jebelean's algorithm for inputs having a norm less than a predefined constant $\mathcal{J}$ (in practice, we could remove the test w.r.t. $\mathcal{A}$ which is often smaller than $\mathcal{J}$).

To insist on the symmetry of some of the computations, we introduce the function Reduce presented later as Algorithm 4 and that is used twice. We have been very precise in the handling of matrix products needed during the algorithm (the $\otimes$ operations). Note that the last matrix product at line 8 is not needed if we do not require it in the last call, which saves time, because the two matrices are big.

ThullYapHGCD proceeds from (a, b) of size $2m$; after the first call to Reduce, we get numbers a' and b' of size $\approx 3m/2$, and after the second call, the final result are a' and b' of size $\approx m$.

3.2.6 Comments of our changes with the reference. We elaborate on the differences between Algorithm 3 and the reference algorithm in [26]:

Step 1. There are two reasons to stop immediately: when $\|a\| < 2$, or $\|a\| \geq 2$ and $\|b\| < m$, in which case a and b satisfy the bounds (2) already.

Step 2.2 When $\|a_0\| < 2$, the recursive call to ThullYapHGCD came back with $M = E$, which means that we need to use the correcting procedure Fixup$_0$. Otherwise we can apply the procedure Fixup with auxiliary parameter $t = \mathbb{T}(a_0)$.

Step 4.1 is added since this case can (and does) happen.

Step 4.2 This step corresponds to the fact that (c, d) are close enough to our goal (a', b') and it is enough to use Fixup$_0$ to get there.

4 APPLICATION TO CORNACCHIA'S ALGORITHM

Cornacchia's algorithm [6] is used to solve the Diophantine equation $x^2 + dy^2 = N$ in coprime integers x, y, for given squarefree integers $N, d > 0$. This is an important tool in the building of complex multiplication elliptic curves for primality proving [17] where such a fast version was anticipated.

The algorithm (see [19]) starts from a root $a > N/2$ of $a^2 = -d \bmod N$ and computes the euclidean remainder sequence (r_i) on (a, N) stopping for the index ℓ such that $r_\ell < \sqrt{N} \leq r_{\ell-1}$. For small numbers, we may use the functions PartialJebelean. This version is quadratic but very fast in practice. We see that we need

Function ThullYapHGCD(a, b)
  Input : $a > b \geq 0$ two integers
  Output: a', b', a regular pair $\mathcal{M} = (Q, M)$ such that
    $(a, b) = M(a', b')$ $M = E$ or
    $\|a'\| \geq \mathbb{T}(a) > \|b'\|$
  1. $m \leftarrow \mathbb{T}(a)$;
  if $\|a\| < 2$ *or* $\|b\| < m$ **then**
    // if $\|a\| \geq 2$, $\|a\| \geq m > \|b\|$
    return $a, b, (\emptyset, E)$;
  if $\|a\| < \mathcal{A}$ **then**
    return PartialEuclid(a, b, m);
  if $\|a\| < \mathcal{J}$ **then**
    return PartialJebelean(a, b, m);
  2. $a', b', \mathcal{M}, t \leftarrow$ Reduce(a, b, m);
    // $\|a'\| \geq m + t > \|b'\|$
  3. **if** $\|b'\| < m$ **then**
    return $a', b', \mathcal{M}$; // $\|a'\| \geq m + t \geq m > \|b'\|$
  4. $q \leftarrow a' \div b'$; $(c, d) \leftarrow (b', a' \bmod b')$; $\mathcal{M} \leftarrow \mathcal{M} \otimes \langle q \rangle$;
  4.1 **if** $\|d\| < m$ **then**
    return $c, d, \mathcal{M}$; // $\|c = b'\| \geq m > \|d\|$
  4.2 **if** $\|1 + (c \div \mathbb{B}^m)\| < \mathcal{A}$ **then**
    $a', b', \mathcal{T} \leftarrow$ Fixup$_0(c, d, m)$;
    return $a', b', \mathcal{M} \otimes \mathcal{T}$;
  5. $\ell \leftarrow \lceil \|c\| \rceil$; $k \leftarrow 2m - \ell - 1$; // $k \approx m/2$
  6. $c', d', \mathcal{S}, t' \leftarrow$ Reduce(c, d, k);
    // $\|c'\| \geq k + t' = m + 1 > \|d'\|$
  7. $a', b', \mathcal{T} \leftarrow$ Fixup$_0(c', d', m)$;
  8. $\mathcal{M} \leftarrow \mathcal{M} \otimes (\mathcal{S} \otimes \mathcal{T})$;
  9. **return** $a', b', \mathcal{M}$;

Algorithm 3: Our version of the Thull-Yap HGCD algorithm.

Function Reduce(a, b, m)
  Input : $a > b \geq 0$ two integers, m an integer,
    $\|a\| > \|b\| \geq m$
  Output: $a', b', \mathcal{M} = (Q, M)$, t such that
    $(a, b) = M(a', b')$ for regular M with $M = E$
    (and $t = 0$) or $\|a'\| \geq m + t > \|b'\|$ (and
    $t = \mathbb{T}(a_0)$ with a_0 described below).
  1. $a_0 \leftarrow 1 + (a \div \mathbb{B}^m)$; $b_0 \leftarrow b \div \mathbb{B}^m$;
  2. **if** $\|a_0\| < \mathcal{A}$ **then**
    $t \leftarrow 0$;
    $a', b', \mathcal{M} \leftarrow$ Fixup$_0(a, b, m)$;
  else
    2.0. $t \leftarrow \mathbb{T}(a_0)$;
    2.1. $a_0^*, b_0^*, \mathcal{M} \leftarrow$ ThullYapHGCD(a_0, b_0);
    2.2. $(a', b') \leftarrow (a_0^*, b_0^*)\mathbb{B}^m + M^{-1}(-a_1, b_1)$ with
      $\mathcal{M} = (Q, M)$;
    2.3. $a', b', \mathcal{M} \leftarrow$ Fixup$(a', b', \mathcal{M}, m, t)$;
  3. **return** $a', b', \mathcal{M}, t$;

Algorithm 4: Algorithm Reduce.

to perform a *single* call to ThullYapHGCD to get an approximation to our answer, which leads to a fast algorithm.

Function FastSquarerootRemainder(a, b)
  Input : $a > b$ two integers
  Output: The largest euclidean remainder r_ℓ in the
    sequence for (a, b) such that $r_\ell \leq \sqrt{a} < r_{\ell-1}$
  1. $a', b', M \leftarrow$ ThullYapHGCD(a, b);
  2. **return** PartialEuclid$(a', b', \sqrt{a})$;

Algorithm 5: FastSquarerootRemainder.

Proposition 4.1. *Algorithm* FastSquarerootRemainder *is correct and terminates in time* $O(\log n \mathrm{M}(n))$ *algorithm for* n*-bit integer* N.

Proof: since N is not a perfect square, that large inequalities do not matter. From (2), we get that at the beginning of Step 2, we have $b' < \mathbb{B}^2\sqrt{a}$ and $a' \geq \mathbb{B}\sqrt{a}$. If $b' \leq \sqrt{a}$, we are done. Otherwise, using Lemma 2.1, we conclude that with at most 4β steps of the Euclidean algorithm, we find two remainders of the Euclidean sequence for a', b' (and therefore for (a, b)) satisfying $a^* \geq \sqrt{a} > b^*$. The complexity analysis comes from the reference. □

In practice, we can replace PartialEuclid by PartialJebelean in Step 2.

5 TESTING

HGCD programs are somewhat easier to test than gcd programs, since for the latter, random pairs of integers yield a gcd that is in general very small. For HGCD, using random pairs or making exhaustive loops is worthwhile. To go further, we first explain how to compute pairs that have large Euclidean quotients. Since the Thull-Yap algorithm is complex, we need to find test numbers that cover the different branches of the algorithm. This is the best we can do. Proving the correctness of the implementation (and of that of other gcd algorithms) might be done using the techniques of [14] and this would have a definite impact on these tricky programs. In the following, we give *families* of test numbers.

5.1 Special numbers

Let $M = 2^n - 1$ for $n \geq 0$ denote a Mersenne number; F_n designates the n-th Fibonacci number. Remember that $\gcd(M_n, M_k) = M_{\gcd(n,k)}$; also $\gcd(F_n, F_{n-1}) = 1$ (and provide the maximum length sequence with all quotients equal to 1). We hope to create a maximum of problems with all these sequences of 1 in their binary expansion. Note that

$$2^n - 1 = 2(2^{n-1} - 1) + 1, \quad 2^{n-1} - 1 = (2^{n-1} - 1) \times 1 + 0$$

so that the sequences of quotients is $(2, 2^{n-1} - 1)$ giving an example of quotients tending to infinity. This can be generalized as

$$M_n = 2^n - 1 = 2^{n-k}(2^k - 1) + 2^{n-k} - 1 = 2^{n-k}M_k + M_{n-k}.$$

for which, with $k < n/2$, M_{n-k} is the euclidean remainder and the partial quotient 2^{n-k} is large. This is also a way of generating several large quotients in the same Euclidean sequence.

5.2 Special primes as tests for ThullYapHGCD

We take the following from [2]. Suppose we want to write a prime p as $4p = x^2 + dy^2$ in coprime integers x and y (i.e., write p as the norm of an integer in the quadratic field $\mathbb{Q}(\sqrt{-d})$ with $d > 0$). For special

values of positive $d \equiv 1, 2 \bmod 4$, the so-called *idoneal numbers*, congruence conditions are enough for asserting the existence of (x, y). To test our HGCD algorithm, it is enough to compute r_ℓ using FastSquarerootRemainder and check whether $p - r_\ell^2 = dy^2$.

5.3 Covering all cases

Now, we need to craft many more of these to cover all the branches of the code, in particular the Fixup procedure at the heart of the algorithm. We give below families of tests that work for any value of $\mathbb{B}$.

Remember the parameters associated with input numbers (a, b):

$$m = \mathbb{T}_{\mathbb{B}}(a), a_0 = 1 + (a \div \mathbb{B}^m), a_1 = a_0\mathbb{B}^m - a, t = \mathbb{T}_{\mathbb{B}}(a_0);$$

$$(b_0, b_1) = (b \div \mathbb{B}^m, b \bmod \mathbb{B}^m).$$

The following easy result will help us building test instances, starting from auxiliary integers u, A_i and B_i that help us construct two integers a and b as input for the program.

Lemma 5.1. *Let u be an integer and $n \in \{u+2, u+3\}$; put $t = 1 + \lceil u/2 \rceil$. Let A_0, A_1, B_0, B_1 be integers, such that*
(i) $\mathbb{B}^{u-1} < B_0 < A_0 < \mathbb{B}^u$.
(ii) $B_0 < A_0$; $1 \le A_1 \le \mathbb{B}^n$, $0 \le B_1 < \mathbb{B}^n$.
Let $b = B_0\mathbb{B}^n + B_1$ and

$$a = \begin{cases} (A_0 - 1)\mathbb{B}^n + A_1 & \text{if } 0 \le A_1 < \mathbb{B}^n, \\ (A_0 - 2)\mathbb{B}^n + A_1 & \text{if } A_1 = \mathbb{B}^n. \end{cases}$$

Then a and b are such that $\mathbb{T}_{\mathbb{B}}(a) = n$ and $a_i = A_i$, $b_i = B_i$.

Proof: For the first case,

$$\mathbb{B}^{n+u-1} < a < (\mathbb{B}^u - 1)\mathbb{B}^n + (\mathbb{B}^n - 1) = \mathbb{B}^{n+u} - 1,$$

so that $n+u-1 < \|a\| < n+u$ and $\mathbb{T}_{\mathbb{B}}(a) = n$ for the $n \in \{u+2, u+3\}$. The other properties are obvious, as well as the second case. □

Let us come back to our algorithm. By construction, $(a_0', b_0') = M^{-1}(a_0, b_0)$ and

$$\|a_0'\| \ge t > \|b_0'\|.$$

Our matrix $M = [[p, q], [r, s]]$ has positive entries satisfying $p \ge q, r \ge s > 0$. Using Lemma 3.2, we compute

$$\delta(a' + b') = \delta(a_0' + b_0')\mathbb{B}^m + (r - s)a_1 + (p - q)b_1, \tag{7}$$

where a_1, b_1, $r - s$ and $p - q$ are positive by construction. Our idea is to force the fixup case to appear when coming back to the first level of recursion, which is enough for our purpose.

Say $k \ge 2$ and

$$\begin{aligned} a_0 &= q_1 b_0 + r_2, 0 \le r_2 < b_0, \\ b_0 &= q_2 r_2 + r_3, 0 \le r_3 < r_2, \\ r_2 &= q_3 r_3 + r_4, 0 \le r_4 < r_3, \\ \cdots &\quad \cdots \quad \cdots \\ r_{k-2} &= q_k r_k + r_{k+1}, 0 \le r_{k+1} < r_k \end{aligned}$$

and $\|r_k\| \ge t > \|r_{k+1}\|$.

Our strategy is to select A_1 and B_1 and the $q_i > 1$'s and r_i's so as to meet the conditions of Lemma 5.1, with $B_0 = q_2 r_2 + r_3$, $A_0 = q_1 B_0 + r_2$ (this makes $A_0 > B_0$ trivially).

5.3.1 *The case* $\det(M) = -1$.

The case (-A) is the most standard case and does not require specially crafted test cases. In the three other cases, one has $b' < 0$. Remember that

$$b' = b_0'\mathbb{B}^n - (ra_1 + pb_1)$$

with $p \ge r$. Large values of p and r will help render b' negative.

All our examples will have $k = 3$. We compute

$$b' = r_4\mathbb{B}^n - A_1 - (B_1 q_1 q_2 + A_1 q_2 + B_1)q_3 - B_1 q_1, \tag{8}$$

$$a' + b' = (r_3 + r_4)\mathbb{B}^n + (-q_1 q_2 q_3 + q_1 q_2 - q_1 - q_3 + 1)B_1 - A_1 q_2 q_3 + A_1 q_2 - A_1 \tag{9}$$

At this point, we note that taking $A_1 = \mathbb{B}^n$ and $r_4 = 1$ makes $b' < 0$ for all values of the remaining parameters.

The case (-B):. we want to build examples with $\|a' + b'\| \ge m + t$. The toggle matrix will give us the solution via $T = [[1, 1], [0, -1]]$ and

$$(a, b) = MT(a' + b', -b').$$

If $M = \langle q_1, \ldots, q_k \rangle$, then $MT = \langle q_1, \ldots, q_k - 1, 1 \rangle$ if $q_k > 1$ and $MT = \langle q_1, \ldots, q_{k-1} + 1 \rangle$ if $q_k = 1$ and $k > 1$; MT is not regular when $k = 1$ and $q_k = 1$, but this is impossible by case (-A) being treated first.

Let us build an example with $q_3 = 1$ in details to illustrate our strategy and we will be shorter in the other cases. Take $r_3 = \mathbb{B}^t$, $q_2 = \mathbb{B}^{u-t-1} - 2$, $q_1 = 1$, $B_1 = 0$ so that $b' = -\mathbb{B}^n q_2$, $a' + b' = \mathbb{B}^{n+t}$. Now

$$\mathbb{B}^{u-1} < B_0 = \mathbb{B}^{u-1} + \mathbb{B}^{u-t-1} - \mathbb{B}^t - 2 < A_0 = \mathbb{B}^{u-1} + \mathbb{B}^{u-t-1} - 1 < \mathbb{B}^u$$

and we apply Lemma 5.1.

Now turn towards $q_3 > 1$. Select first $q_3 = 2$, $r_3 = \mathbb{B}^t + q_2$, $B_1 = 0$, q_1 to obtain

$$a' + b' = \mathbb{B}^{n+t}.$$

When $\mathbb{B} = 2$ (resp. $\mathbb{B} > 2$), we take $q_2 = \mathbb{B}^{u-t-2}$ (resp. $\mathbb{B}^{u-t-1}$) and the corresponding values of A_0 and B_0 meet conditions of Lemma 5.1.

The cases (-C) and (-D):. we want $\|a' + b'\| < m + t$; for (-C), we also want $q_3 \ge 2$ and for (-D), we need $q_3 = 1$.

For (-C), we take $q_3 = 2$, $B_1 = 1$ and the same values as in the case (-B) with $q_3 = 2$.

For (-D), we use the same formulas with $q_3 = 1$ to get

$$a' + b' = r_3\mathbb{B}^n - B_1 q_1.$$

Taking $r_3 = \mathbb{B}^t$ yields $a' + b' = \mathbb{B}^{n+t} - B_1 q_1$, so that $B_1 = 1$ makes $a' + b'$ of small norm. Now we take $q_2 = \mathbb{B}^{u-t-1}$ and $q_1 = 1$ to meet the conditions of Lemma 5.1.

5.3.2 *The case* $\det(M) = +1$.

In this case

$$a' = a_0'\mathbb{B}^n - (qB_1 + sA_1), \quad b' = b_0'\mathbb{B}^n + (rA_1 + pB_1),$$

and remember that $p \ge q, r \ge s$. We need also even $k \ge 2$. All our examples have $k = 2$, this could be extended to larger k's without much trouble. When $k = 2$, we write

$$\begin{aligned} A_0 &= q_1 B_0 + r_2, 0 \le r_2 < B_0, \\ B_0 &= q_2 r_2 + r_3, 0 \le r_3 < r_2 \end{aligned}$$

With these values:

$$p = q_1 q_2 + 1, q = q_1, r = q_2, s = 1.$$

Also, we need

$$\|r_2\| \geq t > \|r_3\|, \tag{10}$$

in order to have exactly two iterations reaching r_3. Moreover:

$$a' = r_2\mathbb{B}^n - (q_1B_1 + A_1), \quad b' = r_3\mathbb{B}^n + (q_2A_1 + (q_1q_2+1)B_1). \tag{11}$$

The case (+A):. our target is $\|a'\| \leq \|b'\|$ or simply $a' \leq b'$. Start from (11) and take $A_1 = \mathbb{B}^n$ making

$$a' = (r_2 - 1)\mathbb{B}^n - q_1B_1, \quad b' = (r_3 + q_2)\mathbb{B}^n + (q_1q_2 + 1)B_1.$$

For instance, take $r_3 = \mathbb{B}^t - 1$, $r_2 = \mathbb{B}^t$, $q_2 = \mathbb{B}^{u-t-2}$, $q_1 = \mathbb{B}$, $B_1 = 1$.

The case (+B):. to build a case with $m + t > \|a'\| > \|b'\|$, we use the formulas (11) again. We may take $r_3 = 1$, $r_2 = \mathbb{B}^t$, $q_2 = \mathbb{B}^{u-t-1}$, $q_1 = 1, A_1 = 1, B_1 = 0$. This makes $\|b'\| < \|a'\| < n + t$ with $n = m$.

The case (+C):. $\|a'\| > \|b'\|$ and $\|a'\| \geq m + t$. This is a very frequent case, no specific construction is needed.

6 IMPLEMENTATION IN GMP

6.1 Norms and lengths

In an implementation context, we may favor use of the number of digits (*length* denoted by L) of an integer a in base $\mathbb{B}$. By convention, $L(0) = 1$ (contrary to the convention in GMP); for $a > 0$:

$$L(a) = n \text{ if } \mathbb{B}^{n-1} \leq a < \mathbb{B}^n, n \geq 1. \tag{12}$$

The Thull-Yap algorithm is described in terms of norms, that is real numbers. It is not conceivable to use these in an actual implementation. Hopefully, we only need *comparisons* of norms. We need several operations:

$$(i)\ \lfloor \|a\| \rfloor = L(a) - 1$$

$$(ii)\ \lceil \|a\| \rceil = \begin{cases} r & \text{if } a = \mathbb{B}^r \\ L(a) & \text{otherwise} \end{cases}$$

$$(iii) \|a\| < m \Leftrightarrow \begin{cases} r < m & \text{if } a = \mathbb{B}^r \\ L(a) \leq m & \text{otherwise} \end{cases}$$

$$(iv) \|a\| \leq m \Leftrightarrow \begin{cases} r \leq m & \text{if } a = \mathbb{B}^r \\ L(a) \leq m & \text{otherwise} \end{cases}$$

$$(v) \|a\| \geq m \Leftrightarrow \begin{cases} r & \text{if } a = \mathbb{B}^r \\ L(a) & \text{otherwise} \end{cases}$$

Testing that $a = \mathbb{B}^r$ is easy when we have the digits of a in base $\mathbb{B}$.

6.2 Matrix-vector products

Let us start with an easy (but important) remark. With the usual notations of Section 3.2.2, write $a = (a_0 - 1)\mathbb{B}^m - a_1$, $b = b_0\mathbb{B}^m + b_1$, with all a_i's and b_i's $\approx \mathbb{B}^m$. The algorithm is applied to (a_0, b_0) and we recover a regular matrix $M = [[p, q], [r, s]]$, together with a_0^*, b_0^* such that $(a_0^*, b_0^*) = M^{-1}(a_0, b_0)$. By Lemma 3.2, we need to compute

$$(a_1', b_1') = \delta[[s, q], [r, p]](a_1, b_1),$$

where all entries are positive (trick!). In Step 2, $a_1, b_1 \sim \mathbb{B}^m$, and r, s, p, q are $\sim \mathbb{B}^{m/2}$ (*unbalanced case*). In Step 6, the corresponding quantities are all $\sim \mathbb{B}^{m/2}$ (*balanced case*). In GMP, some work was put to take care to unbalanced multiplications and we cannot do better.

Were it not the case for GMP, in the unbalanced case and using classical evaluation, we need 8 unbalanced $m/2 \times m$ integer multiplications. Either the underlying arithmetic package knows how to handle this, or we write $a_1 = a_{11}\mathbb{B}^{m/2} + a_{10}$, $b_1 = b_{11}\mathbb{B}^{m/2} + b_{10}$ and evaluate

$$\begin{pmatrix} s & q \\ r & p \end{pmatrix} \begin{pmatrix} a_{10} & a_{11} \\ b_{10} & b_{11} \end{pmatrix} \begin{pmatrix} 1 \\ \mathbb{B}^{m/2} \end{pmatrix}.$$

This amounts to multiplying two 2×2 matrices with entries close to $\mathbb{B}^{m/2}$. We may also use Strassen's (or Bodrato's) algorithm to use (essentially) 7 multiplications of $m/2 \times m/2$ integers. This is done using the function `mpn_matrix22_mul` (implementing Bodrato's version [4]) from GMP, and as of version `6.2.1`, activated at size 14.

In the balance case, we can share (modular) FFTs between entries if they are large enough: computing $sa_1 + qb_1$ and $ra_1 + pb_1$ would cost 6 FFTs (sharing that for a_1, b_1), 4 iFFTs. A similar effort could be put in the Strassen case.

6.3 Handling partial quotients

It is well known that partial quotients in an Euclidean quotient sequence tend to be small. As demonstrated in Section 5, we can build sequences with large quotients, but not too many of them. This is why we propose the following. Our data structure consists in two arrays: the first one T stores integers smaller than some bound *maxint* and the second one P stores pointers to `mpz_t`. We use the classical approach for extending the sizes of T or P, namely doubling the size when required. At index $i \geq 0$ of T, we store $q_i > 0$ if $q_i < maxint$; if $q_i \geq maxint$, we store q_i as an `mpz_t` in P at index $j \geq 0$ and store $-j$ in t. Adding or deleting a partial quotient is easy done and the amortized cost is $O(n)$ if we need to store n elements (using indices for the last elements of T and P). The first idea is to use 64 bits for *maxint*, but 32 will be enough for very large numbers (if needed, we could have several small integer arrays for larger values of *maxint*);

6.4 Computing products of regular matrices

Function BuildMatrixFromQ computes

$$M = \prod_{i=1}^{k} Q_i, \text{where } Q_i = [[q_i, 1], [1, 0]]$$

for generally small q_i's (very frequently fitting in a single machine word). Remember that $k \leq 2 \log b / \log 2$ by Lemma 2.1. Remark that

$$[[u, v], [w, x]][[q, 1], [1, 0]] = [[qu + v, u], [qw + x, w]]$$

and this operation costs essentially two multiplications by q, where q is a single digit (very frequently $q = 1$), so it is rather inexpensive. For large values of k (this is not our case, though), we can use a product tree whose leaves are the Q_i's or subproducts of small matrices.

Matrix multiplications are needed to update the final matrix in our algorithm. The Strassen/Bodrato algorithm is used there too.

6.5 Program and benchmarks

We have implemented Jebelean's algorithm as well as the Thull-Yap algorithm in GMP (also in Magma to clarify it), using low

level functions `mpn_*`. We used the famous `mpn_random2` to check our implementation and we are grateful to the GMP people to have programmed such a lovely and performing bug catcher for complicated integer arithmetic algorithms such as Thull-Yap's. We give timings (in ms) for our implementation of HGCD in Table 1 on our `Intel(R) Xeon(R) CPU E7-4850 v2 @ 2.30GHz`, compiler `gcc 9.3.0` with the `-O2` option. We used 100 random numbers of n 64-bit words in our functions: Euclid, Jebelean and Thull-Yap. Needless to say, Euclid's algorithm is less efficient than Jebelean's

n	Euclid	Jebelean	Thull-Yap
500	5.76	1.00	2.24
1000	19.02	2.65	4.57
1500	41.57	5.46	7.46
2000	73.77	9.21	10.28
2500	116.91	13.96	13.45
3000	170.02	19.67	17.00
3500	232.28	26.27	20.49
4000	303.88	33.82	23.65
4500	384.08	42.37	27.04
5000	474.45	51.86	31.01
5500	571.34	62.58	35.37
6000	679.12	74.41	39.20
6500	794.60	87.91	43.37
7000	940.11	102.83	47.34
7500	1084.10	117.68	51.54
8000	1209.63	137.25	55.07
8500	1388.21	151.99	58.80
9000	1550.27	171.17	62.96
9500	1695.10	191.02	67.60
10000	1928.16	210.71	72.57

Table 1: Average timings for PartialRemainder algorithms on n-word numbers.

quite rapidly. The Thull-Yap algorithm is more efficient around 2500 64-bit words, sizes larger than currently used in ECPP's records. More work is needed to gain time on this implementation. Directly comparing with GMP's implementation of `mpn_hgcd` is difficult. For the sake of comparison, we compared function `mpn_gcd` with the full gcd algorithm associated to our implementation of the Thull-Yap algorithm; results are given in Table 2. GMP is superior to our implementation (reasons for this include the handling of all the 2×2 matrices appearing, and also some required copies of large integers).

7 CONCLUSION

We have implemented and discussed some potential improvements to the Thull-Yap algorithm. A large part of our implementation can be used to implement the algorithm of Lichtblau. This is work in progress [18]. Magma code corresponding to this article is available on the author's web page [1].

[1] http://www.lix.polytechnique.fr/Labo/Francois.Morain

n	Thull-Yap	GMP
500	4.69	1.32
1000	10.28	3.55
1500	16.69	6.23
2000	22.60	9.26
2500	28.44	12.57
3000	36.33	15.27
3500	42.89	18.62
4000	48.71	22.38
4500	55.47	26.31
5000	62.04	31.67
5500	70.93	36.66
6000	81.12	40.31
6500	89.17	45.35
7000	97.16	49.80
7500	105.69	54.01
8000	112.91	60.72
8500	118.82	65.34
9000	127.97	70.13
9500	136.60	75.19
10000	146.63	82.12

Table 2: Average timings for gcd algorithms on n-word numbers.

We applied it to Cornacchia's algorithm. This algorithm is also used in $\mathbb{Z}[i]$ for cryptographic applications [13]. A fast gcd algorithm exists for $\mathbb{Z}[i]$ (see [28]), and it would be interesting to try to adapt the Thull-Yap algorithm for this goal.

ACKNOWLEDGMENTS

We thank A. Bostan, F. Chyzak and M. Mezzarobba for discussions on the optimal matrix multiplication algorithms; A. Rosowski for answering queries on his work [21] and sending formulas that did not go through in our work; D. Stehlé for discussion around the Thull-Yap algorithm; N. Möller for answering questions on his algorithm and its implementation in GMP; E. Bach and Collins's daughter for trying to locate the work referred to in the introduction (work that seems to be lost); J. van der Hoeven, G. Lecerf for helpful discussions. Finally, the referees for this work did a very precise job enabling to improve the exposition of the results.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. 1974. *The design and analysis of computer algorithms.* Addison–Wesley.
[2] A. O. L. Atkin and F. Morain. 1993. Elliptic curves and primality proving. *Math. Comp.* 61, 203 (July 1993), 29–68.
[3] D. J. Bernstein and B.-Y. Yang. 2019. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 3 (2019), 340–398. https://doi.org/10.13154/tches.v2019.i3.340-398
[4] M. Bodrato. 2010. A Strassen-like matrix multiplication suited for squaring and higher power computation. In *Symbolic and Algebraic Computation, International Symposium, ISSAC 2010, Munich, Germany, July 25-28, 2010, Proceedings*, Wolfram Koepf (Ed.). ACM, 273–280. https://doi.org/10.1145/1837934.1837987
[5] R. P. Brent. 1976. Analysis of the binary Euclidean algorithm. *SIGSAM Bull.* 10, 2 (1976), 6–7. https://doi.org/10.1145/1093397.1093399
[6] G. Cornacchia. 1908. Su di un metodo per la risoluzione in numeri interi dell' equazione $\sum_{h=0}^{n} C_h x^{n-h} y^h = P$. *Giornale di Matematiche di Battaglini* 46 (1908), 33–90.

[7] GMP. [n.d.]. https://gmplib.org.

[8] T. Jebelean. 1993. Improving the multiprecision Euclidean algorithm. In *Design and implementation of symbolic computation systems (Lecture Notes in Comput. Sci.)*, A. Miola (Ed.), Vol. 722. Springer-Verlag, 45–58. Proceedings DISCO'93, Gmunden, Austria, September 15-17, 1993.

[9] T. Jebelean. 1995. A Double-Digit Lehmer-Euclid Algorithm for Finding the GCD of Long Integers. *J. Symb. Comput.* 19, 1-3 (1995), 145–157. https://doi.org/10.1006/jsco.1995.1009

[10] D. E. Knuth. 1970. The analysis of algorithms. In *Actes du Congrès International des Mathématiciens*. Gauthier-Villars, 269–274.

[11] D. H. Lehmer. 1938. Euclid's algorithm for large numbers. *Amer. Math. Monthly* 45 (1938), 227–233.

[12] D. Lichtblau. 2005. Half-GCD and fast rational recovery. In *Symbolic and Algebraic Computation, International Symposium ISSAC 2005, Beijing, China, July 24-27, 2005, Proceedings*, Manuel Kauers (Ed.). ACM, 231–236. https://doi.org/10.1145/1073884.1073917

[13] P. Longa and F. Sica. 2014. Four-Dimensional Gallant-Lambert-Vanstone Scalar Multiplication. *J. Cryptol.* 27, 2 (2014), 248–283. https://doi.org/10.1007/s00145-012-9144-3

[14] G. Melquiond and R. Rieu-Helft. 2020. WhyMP, a Formally Verified Arbitrary-Precision Integer Library. In *ISSAC 2020 - 45th International Symposium on Symbolic and Algebraic Computation*. Kalamata, Greece, 352–359. https://doi.org/10.1145/3373207.3404029

[15] R. T. Moenck. 1973. Fast Computation of GCDs. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong (Eds.). ACM, 142–151. https://doi.org/10.1145/800125.804045

[16] N. Möller. 2008. On Schönhage's algorithm and subquadratic integer GCD computation. *Math. Comp.* 77, 261 (2008), 589–607. https://doi.org/10.1090/S0025-5718-07-02017-0

[17] F. Morain. 2007. Implementing the asymptotically fast version of the elliptic curve primality proving algorithm. *Math. Comp.* 76 (2007), 493–505. http://www.lix.polytechnique.fr/Labo/Francois.Morain/Articles/fastecpp-final.pdf

[18] F. Morain. 2022. Remarks on Lichtblau's HGCD algorithm. (Feb. 2022). Preprint.

[19] A. Nitaj. 1995. L'algorithme de Cornacchia. *Exposition. Math.* 13 (1995), 358–365.

[20] V. Y. Pan and X. Wang. 2002. Acceleration of Euclidean algorithm and extensions. In *Symbolic and Algebraic Computation, International Symposium ISSAC 2002, Lille, France, July 7-10, 2002, Proceedings*, Teo Mora (Ed.). ACM, 207–213. https://doi.org/10.1145/780506.780533

[21] A. Rosowski. 2019. Fast Commutative Matrix Algorithm. *CoRR* abs/1904.07683 (2019). arXiv:1904.07683 http://arxiv.org/abs/1904.07683

[22] A. Schönhage. 1971. Schenelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica* 1 (1971), 139–144.

[23] J. Sorenson. 1994. Two fast GCD algorithms. *J. Algorithms* 16 (1994), 110–144.

[24] D. Stehlé and P. Zimmermann. 2004. A Binary Recursive Gcd Algorithm. In *Algorithmic number theory (Lecture Notes in Comput. Sci.)*, D. Buell (Ed.), Vol. 3076. Springer, 411–425. 6th International Symposium, ANTS-VI, Burlington, VT, USA, june.

[25] J. Stein. 1967. Computational problems associated with Racah algebra. *J. Comp. Phys.* 1 (1967), 397–405.

[26] K. Thull and C.-K. Yap. 1990. A Unified Approach to HGCD Algorithms for polynomials and integers. Available at https://cs.nyu.edu/yap/papers/index.html.

[27] K. Weber. 1995. The accelerated integer GCD algorithm. *ACM Trans. Math. Software* 21, 1 (March 1995), 111–122.

[28] A. Weilert. 2000. Asymptotically Fast GCD Computation in Z[i]. In *Algorithmic Number Theory, 4th International Symposium, ANTS-IV, Leiden, The Netherlands, July 2-7, 2000, Proceedings (Lecture Notes in Computer Science)*, Wieb Bosma (Ed.), Vol. 1838. Springer, 595–613. https://doi.org/10.1007/10722028_40

[29] C.-K. Yap. 2000. *Fundamental problems of algorithmic algebra*. Oxford University Press.