ON THE MEMORY CONSUMPTION OF THE CONTROLLED EUCLIDEAN DESCEND

Gábor Román (Budapest, Hungary)

Communicated by Antal Járai

(Received October 16, 2018; accepted April 10, 2019)

Abstract. In this article we give an algorithm for the controlled euclidean descend which is tailored to the memory architecture used in the computers of today. We supply memory allocation strategy for this algorithm.

1. Introduction

During the classical euclidean method utilising division for computing the greatest common divisor of two integers, the euclidean steps which one performs are altogether called the euclidean descend. In the process, the distance of the input numbers diminishes step-by-step. Let's presume that we want to proceed with the euclidean descend until the difference of the input numbers goes below a certain threshold. So let an $s > 0$ integer be our threshold, and say that we want to perform euclidean steps on the input numbers a and b until $|a - b| < s$ becomes true. This is the task for example during the Cornacchia algorithm, see algorithms 1.5.2 and 1.5.3 in [2], which is used in the Atkin–Morain primality test, see [1]. When such threshold is applied, one calls the euclidean descend as controlled euclidean descend.

The first subquadratic algorithm for computing the greatest common divisors of two integers is by Knuth, see [3]. Schönhage improved this result

Key words and phrases: Controlled euclidean descend.
2010 Mathematics Subject Classification: 11A05, 11Y16.
https://doi.org/10.71352/ac.49.369

using the idea of the controlled euclidean descend combined with a divide-and-conquer scheme in [5]. Further development was done by Schönhage, but the resulting algorithm wasn't published. Later, the same idea is applied for the reduction of binary quadratic forms in [6] also by Schönhage. Based on this paper Möller reconstructed a method called SGCD in [4] which is probably equivalent to Schönhage's unpublished algorithm.

Define $l(z) := \lceil \log_2(1+|z|) \rceil$ to denote the bit size of a given z integer. For any $z \neq 0$ integer, if $l(z) = s$, then $2^{s-1} \leq |z| < 2^s$. It is easier to check the $l(a-b) < s$ condition instead of $|a-b| < s$, so we will make decisions based on bit length instead of the actual value.

We are going to utilise a subroutine called "sdiv," a controlled division. This method has three positive integer inputs $a > b$ and s, where s denotes bit length and $\min\{l(a), l(b)\} > s$ should hold. This routine calculates the greatest such q integer, for which $qb < a$ and $l(a-qb) > s$ holds. This can be implemented as follows. Let $q' \leftarrow \lfloor a/b \rfloor$, and if $l(a-q'b) > s$ then let $q \leftarrow q'$, otherwise let $q \leftarrow q'-1$. The return values are q and $r = a - qb$.

Now we give the reconstructed SGCD algorithm as it is stated in Möller's article.

Algorithm 1. $(SGCD(s,a,b))$ Given a positive integer s threshold and positive integers $a > b$ for which $\min\{l(a), l(b)\} > s$ holds, this algorithm computes positive integers α, β and a 2×2 matrix M having non-negative integer entries, such that $\det M = 1$,

$$\begin{pmatrix} a \\ b \end{pmatrix} = M \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

furthermore $\min\{l(\alpha), l(\beta)\} > s$ and $l(\alpha - \beta) \leq s$ hold.

1. (Initialize) If $\min\{l(a), l(b)\} \leq s+2$ holds, then let $\alpha \leftarrow a$, $\beta \leftarrow b$ and $M \leftarrow I$, then go to step 9.

2. (Split) Let $\sigma \leftarrow \max\{l(a), l(b)\} - s$. If $s \leq \sigma$ holds, then let $s' \leftarrow s$, $p \leftarrow 0$, $\alpha \leftarrow a$ and $\beta \leftarrow b$. Otherwise let $s' \leftarrow \sigma$, $p \leftarrow s - \sigma + 1$ and split a and b such that the $a = 2^p\alpha + a_0$ and the $b = 2^p\beta + b_0$ equalities hold.

3. (Check) Let $h \leftarrow s' + \lfloor \sigma/2 \rfloor$. If $\min\{l(\alpha), l(\beta)\} \leq h$ holds, then let $M \leftarrow I$ and go to step 5.

4. (First recursive call) Let α, β and M be the return values of this algorithm recursively called with h threshold and α, β integers.

5. (Reduce) If $\max\{l(\alpha), l(\beta)\} \leq h$ holds, then go to step 6, otherwise if $l(\alpha - \beta) \leq s'$ holds, then go to step 8. Apply sdiv on α and β using s' as bit length, furthermore update the M matrix accordingly. Repeat step 5.

6. (Second recursive call) Let α, β and M' be the return values of this algorithm recursively called with s' threshold and α, β integers.

7. (Multiplication) Let $M \leftarrow MM'$.

8. (Recombination) If $p > 0$ holds, then let

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \leftarrow 2^p \begin{pmatrix} \alpha \\ \beta \end{pmatrix} + M^{-1} \begin{pmatrix} a_0 \\ b_0 \end{pmatrix}.$$

9. (Final reduction) While $l(\alpha - \beta) > s$ holds, apply sdiv on α and β using s as bit length, furthermore update the M matrix accordingly.

10. (Finished) The results are α, β and M. Terminate the algorithm.

During the algorithm, when one has to apply sdiv on the α, β positive integers using bit length s, if $\alpha > \beta$, then $(q, \alpha) \leftarrow \mathrm{sdiv}(\alpha, \beta, s)$ should be computed and the matrix should be updated as

$$M \leftarrow M \begin{pmatrix} 1 & q \\ 0 & 1 \end{pmatrix}$$

otherwise $(q, \beta) \leftarrow \mathrm{sdiv}(\beta, \alpha, s)$ should be computed and the

$$M \leftarrow M \begin{pmatrix} 1 & 0 \\ q & 1 \end{pmatrix}$$

update should be done. The details about the correctness and the running time of algorithm 1 can be found in Möller's article. We are going to analyse the algorithm from the viewpoint of memory consumption.

2. Algorithm tailored to machine memory

We are going to use the memory scheme applied in RAM machines, but with limited cell capacity, like in the computers of today. So we are going to represent the memory as a one dimensional array, where every element of the array can represent a bit scheme, which – if one interprets it as a natural number n – satisfies $0 \leq n < B$ for some $B > 1$ natural number. If $\mathcal{M}$ denotes such an array, then the ith cell or *word* of this array will be $\mathcal{M}[i]$. Every non-zero natural number n can be represented uniquely in the $n = n_k B^k + \ldots + n_1 B + n_0$ form, where $0 < n_k < B$ and $0 \leq n_i < B$ for $i \neq k$. Such representation can be stored in our memory scheme as $\mathcal{M}[i] = n_0, \mathcal{M}[i+1] = n_1, \ldots, \mathcal{M}[i+k] = n_k$, if

the representation starts at the ith cell in the memory. Zero can be comfortably represented as the only number which has zero length.

Define $L(z) := \lceil \log_B(1 + |z|) \rceil$ to denote the word length of a given z integer. Next to bit lengths, from now on we are going to use word lengths for the representation of integers. The capital letter L will be used to mark word lengths. If the base B representation of a natural number is L words long, and it starts at the ith cell, then the last word of the representation will be at the $(i + L - 1)$th cell.

Computer algebraic algorithms are the most efficient when they are aligned with the memory scheme, so in general they work with word boundaries. Algorithm 1 cannot be modified to work only with word lengths instead of bit lengths, because there is a delicate interconnection between the splitting and the reduction steps. So we keep the bit length operations, but handle the numbers aligned to word boundaries.

Memory is requested from the operating system or *allocated*, which operation counts as an expensive one, so we will give an algorithm, which works on a memory slices requested beforehand the application of the algorithm. Also, we fall back to handle the numbers "in place." From the viewpoint of splitting, this means that when we have to split a number at the pth bit, we won't allocate new memory for the upper part, instead we will do the splitting by shifting the bits after the pth bit to the start of the next higher word as it can be seen in figure 1.

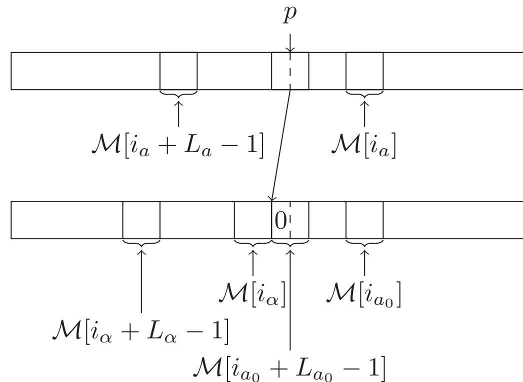

Figure 1. Splitting a at the pth bit. After the split, $a = 2^p\alpha + a_0$ will hold. During the split we shift the bits above the pth bit to the next higher word boundary. We set the emptied bits to zero. This procedure requires an additional available word after the representation of a if the pth bit is not exactly before a word boundary.

This way we can split a number using only shifting, and the resulting numbers can be used readily for further computations. Albeit there should be one more memory cell available after the number, if the split happens at a bit which is not before a word boundary. The memory slice which we will use to represent the a and b input numbers will be $\mathcal{A}$ and $\mathcal{B}$ respectively.

We have to store the resulting matrices between the recursive calls. For this, we are going to use the memory slices $\mathcal{M}_{11}$, $\mathcal{M}_{12}$, $\mathcal{M}_{21}$ and $\mathcal{M}_{22}$ to store the elements of the matrices in a stack-like scheme, see figure 2.

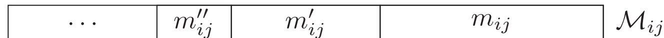

Figure 2. Elements of matrices stored in a stack-like scheme.

The elements of the matrices at the same position will be represented in a common memory slice next to each other. The resulting matrix elements of the first recursive call at every level will be stored in the (currently) bottom part of $\mathcal{M}_{ij}$, and the resulting matrix elements of the second recursive call will be stored right above them.

The memory slices $\mathcal{A}$ and $\mathcal{B}$ should have more place than which is just enough for the representation of a and b, because during the recursive calls one may continuously split the input numbers. Also, despite the fact that we know an upper bound for the resulting matrices on the topmost level, we need more place in $\mathcal{M}_{ij}$ slices to store the matrices at the lower levels of the recursion.

First we look at the required space in $\mathcal{A}$ and $\mathcal{B}$. During algorithm 1, the deepest level of recursion is at most $\log_2 m$, where m is the maximal bit length of the input numbers at the topmost level. This means that $\log_2 m$ number of splits can occur at most. Every split could introduce the need for an additional word in the memory, so $\lceil \log_B(m+1) \rceil + \lceil \log_2 m \rceil$ words will be enough space in $\mathcal{A}$ and in $\mathcal{B}$ separately at the beginning of algorithm 2.

Now we turn to the required storage space in an $\mathcal{M}_{ij}$ slice before the application of algorithm 2. We need the following lemma from the article of Möller.

Lemma 1 (Size of the matrix indices)**.** *Let a, b, α and β be positive integers, for which* $\max\{l(a), l(b)\} \le n$ *and* $\min\{l(a), l(b)\} > s$*. Let furthermore* $M := (m_{11}, m_{12}; m_{21}, m_{22})$ *given row-wise be a 2×2 matrix with non-negative integer entries such that*

$$\begin{pmatrix} a \\ b \end{pmatrix} = M \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

and $\det M = 1$*. Then* $\max\{l(m_{11}), l(m_{12}), l(m_{21}), l(m_{22})\} \le n - s$*. In fact we also have* $\max\{l(m_{11} + m_{12}), l(m_{21} + m_{22})\} \le n - s$*.*

Based on this lemma, the size of the resulting matrix (and the matrices on the lower levels of the resursion) will be the largest if we invoke algorithm 1 with threshold set to zero. For the partial call tree in this case, see figure 3.

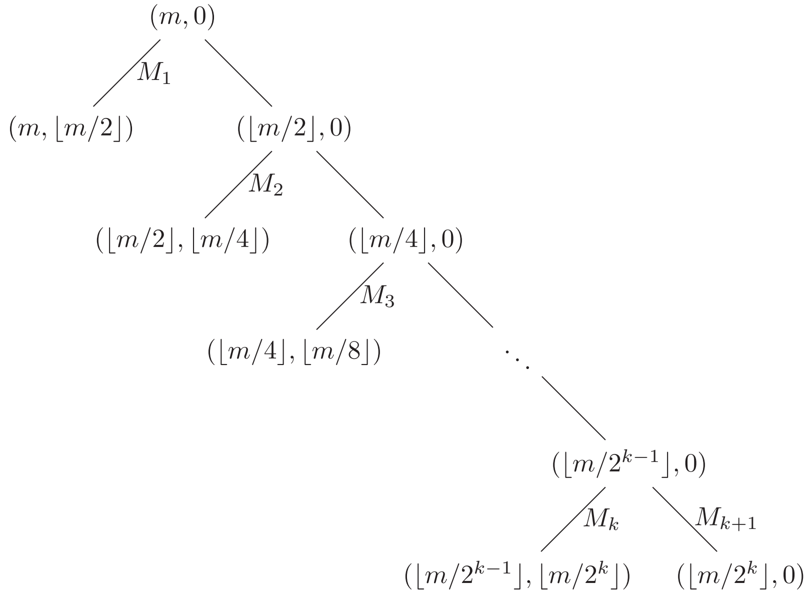

Figure 3. Part of the call tree when one invokes algorithm 1 with threshold set to zero. Going left in the tree symbolises the first recursive call, going right symbolises the second recursive call. The first number in the pairs is the maximal bit length of the input numbers during the given call and the second number is the threshold. Take note that k is such that $2^k \leq m < 2^{k+1}$ holds. The resulting matrices are denoted as $M_1, M_2, \ldots, M_{k+1}$.

If $\log_2 m$ is the deepest recursion level, then the greatest number of matrices we have to store simultaneously is $k+1$ if $2^k \leq m < 2^{k+1}$, see figure 3. The resulting matrices have non-negative elements, so as we progress left in the tree, the size of the matrices' elements get smaller and smaller. Based on this we have to give an upper bound for the space required to store the corresponding elements from the $M_1, M_2, \ldots, M_{k+1}$ matrices, to get the required size of an $\mathcal{M}_{ij}$ slice. According lemma 1, this will be

$$1 + \sum_{i=0}^{k-1} \left\lceil \log_B \left(\left\lfloor \frac{m}{2^i} \right\rfloor + 1 \right) \right\rceil$$

because an element of M_{k+1} requires one word, furthermore after the first

recursive call, the size of the elements of the resulting M_i matrix can be at most $\lfloor \frac{m}{2^i} \rfloor - \lfloor \frac{m}{2^{i+1}} \rfloor$, but during the reductions, their size can get bigger.

This size increment may cause problems, but there is a simple solution. On any recursive level, let the matrix after step (5) be M', the return matrix after step (6) be M'', and their product be M. We obtain that

$$m_{11} = m'_{11}m''_{11} + m'_{12}m''_{21},$$
$$m_{22} = m'_{22}m''_{22} + m'_{21}m''_{12},$$

which means that above m'_{11} there is enough space except one word for m''_{11}, and above m'_{22} there is enough space except one word for m''_{22}. Moreover above m'_{12} there is space for m''_{21} and above m'_{21} there is space for m''_{12}. (Again, except one word in both cases of course.) The cases when either $m'_{12} = 0$ or $m'_{21} = 0$ are trivial, one only has to check the calculation of m_{12} and m_{21}. (The cases when either $m''_{12} = 0$ or $m''_{21} = 0$ are also trivial because of this.) Hence it is enough to swap the pointers of the non-diagonal elements.

Algorithm 2. $\big(SGCD(s, i_a, L_a, i_b, L_b, i_{11}, i_{12}, i_{21}, i_{22}, L_{11}, L_{12}, L_{21}, L_{22})\big)$ Let s be a non-negative integer threshold, furthermore $a > b$ positive integers where a is represented in $\mathcal{A}$ starting at i_a having length of L_a, and b is represented in $\mathcal{B}$ starting at i_b having length of L_b. Given that $\min\{l(a), l(b)\} > s$ holds, this algorithm computes positive integers α in $\mathcal{A}$ starting at i_a with length returned in L_a, and β in $\mathcal{B}$ starting at i_b with length returned in L_b, furthermore a 2×2 matrix M with its m_{11}, m_{12}, m_{21} and m_{22} elements stored in the corresponding $\mathcal{M}_{11}$, $\mathcal{M}_{12}$, $\mathcal{M}_{21}$ and $\mathcal{M}_{22}$ slices, starting at i_{11}, i_{12}, i_{21} and i_{22} having length returned in L_{11}, L_{12}, L_{21} and L_{22} respectively, such that $\det M = 1$,

$$\begin{pmatrix} a \\ b \end{pmatrix} = M \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

where $\min\{l(\alpha), l(\beta)\} > s$ and $l(\alpha - \beta) \leq s$ hold.

1. (Initialize) If $\min\{l(a), l(b)\} \leq s + 2$ holds, then let $M \leftarrow I$ and go to step 9.

2. (Split) Let $\sigma \leftarrow \max\{l(a), l(b)\} - s$. If $s \leq \sigma$ holds, then let $s' \leftarrow s$, $p \leftarrow 0$, $i_\alpha \leftarrow i_a$ and $i_\beta \leftarrow i_b$. Otherwise let $s' \leftarrow \sigma$, $p \leftarrow s - \sigma + 1$ and shift the bits of a and b from the pth bit as describe before the algorithm. Let $i_{a_0} \leftarrow i_a$, $L_{a_0} \leftarrow \lceil \log_B p \rceil$, $i_\alpha \leftarrow i_a + L_{a_0}$, $L_\alpha \leftarrow L_a - L_{a_0}$, furthermore $i_{b_0} \leftarrow i_b$, $L_{b_0} \leftarrow \lceil \log_B p \rceil$, $i_\beta \leftarrow i_b + L_{b_0}$ and $L_\beta \leftarrow L_b - L_{b_0}$.

3. (Check) Let $h \leftarrow s' + \lfloor \sigma/2 \rfloor$. If $\min\{l(\alpha), l(\beta)\} \leq h$ holds, then let $M \leftarrow I$ and go to step 5.

4. (First recursive call) Call this algorithm recursively with h threshold, α represented in $\mathcal{A}$ at i_α with length L_α, β represented in $\mathcal{B}$ at i_β with length L_β and M represented in $\mathcal{M}_{11}$, $\mathcal{M}_{12}$, $\mathcal{M}_{21}$, and $\mathcal{M}_{22}$ starting at i_{11}, i_{12}, i_{21} and i_{22} updating the lengths L_{11}, L_{12}, L_{21} and L_{22} respectively.

5. (Reduce) If $\max\{l(\alpha), l(\beta)\} \leq h$ holds then go to step 6, otherwise if $l(\alpha - \beta) \leq s'$ holds then go to step 8. Apply sdiv on α and β using s' as bit length, furthermore update the M matrix accordingly. Repeat step 5.

6. (Second recursive call) Call this algorithm recursively with s' threshold, α represented in $\mathcal{A}$ at i_α with length L_α, β represented in $\mathcal{B}$ at i_β with length L_β and M' represented in $\mathcal{M}_{11}$, $\mathcal{M}_{12}$, $\mathcal{M}_{21}$, and $\mathcal{M}_{22}$ starting at $i_{11} + L_{11}$, $i_{12} + L_{12}$, $i_{21} + L_{21}$ and $i_{22} + L_{22}$ storing the lengths of the resulting elements in L'_{11}, L'_{12}, L'_{21} and L'_{22}.

7. (Multiplication) Let $M \leftarrow MM'$.

8. (Recombination) If $p > 0$ holds, then let

$$\begin{pmatrix} a \\ b \end{pmatrix} \leftarrow 2^p \begin{pmatrix} \alpha \\ \beta \end{pmatrix} + M^{-1} \begin{pmatrix} a_0 \\ b_0 \end{pmatrix},$$

otherwise shift α and β to i_a and i_b respectively, furthermore let $L_a \leftarrow L_\alpha$ and $L_b \leftarrow L_\beta$.

9. (Final reduction) While $l(a - b) > s$ holds, apply sdiv on a and b using s as bit length, furthermore update the M matrix accordingly.

10. (Finished) The results are in $\mathcal{A}$ at i_a with length stored in L_a, $\mathcal{B}$ at i_b with length stored in L_b, furthermore M represented in $\mathcal{M}_{11}$, $\mathcal{M}_{12}$, $\mathcal{M}_{21}$, and $\mathcal{M}_{22}$ starting at i_{11}, i_{12}, i_{21} and i_{22} with element lengths stored in L_{11}, L_{12}, L_{21} and L_{22} respectively. Terminate the algorithm.

A few comments should be made about the algorithm. At the topmost call, i_{11}, i_{12}, i_{21} and i_{22} should be zero, or the places where we want to put the resulting elements of our matrix into the $\mathcal{M}_{11}$, $\mathcal{M}_{12}$, $\mathcal{M}_{21}$ and $\mathcal{M}_{22}$ slices. The only criterion is that we should have enough storage space in a given slice after the initial cell to store the resulting matrices during the recursive stages. In step 8 we don't have to actually calculate the inverse of the matrix M. This is based on

$$M^{-1} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}^{-1} = \frac{1}{\det M} \begin{pmatrix} m_{22} & -m_{12} \\ -m_{21} & m_{11} \end{pmatrix} = \begin{pmatrix} m_{22} & -m_{12} \\ -m_{21} & m_{11} \end{pmatrix}$$

because $\det M = 1$, and the representation of negative numbers can be avoided by intervening the signs into the computation of the recombination in the form

of branching. Throughout the recursive calls, three temporary variables with lengths taken to be the maximum of the word lengths of the input numbers at the topmost level will suffice to store the intermediate results of the comparisons, sdiv operations, matrix multiplication and the recombination.

References

[1] **Atkin, A.O.L. and F. Morain,** Elliptic curves and primality proving, *Math. Comp.*, **61(203)** (1993), 29–68.

[2] **Cohen, H.,** *A Course In Computational Algebraic Number Theory*, Springer–Verlag, third, corrected printing (1996).

[3] **Knuth, D.E.,** The analysis of algorithms, *Actes du Congrés International des Mathématiciens*, (1970), 269–274.

[4] **Möller, N.,** On Schönhage's algorithm and Subquadratic Integer GCD Computation, *Math. Comp.*, **77(261)** (2008), 589–607. Article electronically published on September 12, 2007.

[5] **Schönhage, A.,** Schnelle Berechnung von Kettenbruchentwicklungen, *Acta Informatica*, **1(2)** (1971), 139–144.

[6] **Schönhage, A.,** Fast reduction and composition of binary quadratic forms, *ISSAC '91 Proceedings of the 1991 international symposium on Symbolic and algebraic computation* (1991), 128–133.

G. Román
Department of Computer Algebra
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest
Pázmány Péter sétány 1/C
Hungary
romangabor@caesar.elte.hu