



VLSI Research Group

University of Windsor

*An Algorithm for Modular
Exponentiation*

Information Processing Letters

V.S. Dimitrov, G.A. Jullien and W.C. Miller

An Algorithm for Modular Exponentiation

V.S. Dimitrov, G.A. Jullien and W.C. Miller

Abstract

A practical technique for improving the performance of modular exponentiations (ME) is described. The complexity of the ME algorithm is $O\left(\frac{n}{\log n}\right)$ modular multiplications (MMs), where n is the length of the exponent, requiring an $O(n^2)$ precomputed look-up table size with very small constant of proportionality. The algorithm uses a double-based number system which we introduce in this paper.

Keywords: Computer arithmetic; Number theory; Cryptography; Modular Exponentiations

1. Introduction

In many cryptographic systems [1]-[6] the processors must perform one or more exponentiations over a group, such as \mathbf{Z}_p or an elliptic curve group. Considering that the decomposition of the exponentiation involves a large number of MMs, this can either be very time consuming or require large amounts of hardware. The problem can be stated as follows: given positive integers A, E and p (in many typical cases, such as the ElGamal cryptosystem [1], p is a prime), compute $y = A^E \pmod{p}$. Without using precomputations, Braur's theorem [7] tells us that the computation of y requires a value of n with a lower bound of $n = \lfloor \log E \rfloor$. This evaluation is still valid even under unbounded parallelism [14].

Recently, several papers have appeared which investigate the problem of reducing the time needed to perform a modular exponentiation (ME) operation when precomputation of appropriately selected powers of A is allowed. Typical applications are discrete log-based systems with a fixed base [1]-[3]. Some of the previously presented algorithms on this subject are based on the assumption that the number of the precomputed powers is $O(n^r)$ ($r \in \{1, 2\}$) and rely on a proper representation of n ; however, the number of MMs is still cn where c is a constant. In [8] an algorithm based on the representation of n as a sum of the numbers of the form $2^i 3^j$ is presented, and in this case we have on average $0.3381n$ MMs. This algorithm was improved in [9], where the representation of n as a sum of the numbers of the form $3^i 5^j$ leads to an algorithm needing, on average, $0.3246n$ MMs with a lower memory requirement.

In this paper we present a significant improvement to the scheme in [8]. It is based on a new number representation which we will refer to as a double-based number system (DBNS). In this system, numbers are represented as a sum of integers of the form $2^i 3^j$, with near minimal non-zero digits. This representation yields an algorithm that is faster for the commonly used 512-bit exponent. All logarithms used in the paper are in base 2.

2. Mathematical background

Following from de-Weger [10] we use the following definitions:

Definition 1: An integer x is called s -integer if all its prime divisors are among the first s primes.

Definition 2: Let $\mathbf{G}_{2,3}(x)$ be the set of 2-integers, that is, numbers of the form $2^i 3^j$, smaller than or equal to x .

An accurate estimation of the cardinality of $\mathbf{G}_{2,3}(x)$ can be obtained if one uses the following theorem, proved by Hardy [11]:

$$\text{Card}(G_{2,3}(x)) = \frac{(\log x)^2}{\log 9} + c \cdot \log x + o\left(\frac{\log x}{\log \log x}\right) \quad (1)$$

where c is a constant. For our purposes the estimation of $\text{Card}(G_{2,3}(x))$ as $O((\log x)^2)$ will be sufficient.

Definition 3: The representation of a given integer x into the form:

$$x = \sum_{i,j} d_{i,j} 2^i 3^j, \quad d_{i,j} \in \{0, 1\} \quad (2)$$

will be referred to as a *double-based number system* (DBNS).

The algorithms reported in [8] and [9] are based on a special representation of the exponent as a sum of 2-integers and 3-integers, respectively. The representation of a given integer into DBNS is not unique and the different representations lead to different algorithms for ME. The representation (also not necessarily unique) of a given integer as a sum of minimal number 2-integers will be referred to as the *canonic double-based number representation* (CDBNR). The main feature of the CDBNR is the unusual sparsity of the representation. It is easy to check, for example, that 23 is the smallest integer requiring three 2-

integers. The smallest integer, requiring four 2-integers is 431, five 2-integers are needed to present 18431 and six 2-integers are needed to present 3 448 733. Up to this limit we can present every integer as a sum of at most five 2-integers. The procedure to find a CDBNR of a given very large integer seems to be a very complex task; however, in the next section we propose a recursive algorithm which allows us to find a representation which is close to minimal; moreover this algorithm is very easily implemented.

3. An Algorithm for Near-Minimal CDBNR

We start by proposing the algorithm and then proceed to prove a theorem about its time complexity.

3.1 Near minimal CDBNR algorithm

We propose the following greedy algorithm with the input as a positive integer x ; and an output of 2-integers, a_i , such that $\sum_i a_i = x$. The algorithm finds the largest 2-integer, w , smaller than or equal to x , and recursively applies the same for $x - w$ until reaching zero.

3.2 Theorem 1

The greedy algorithms terminates after $k = O\left(\frac{\log x}{\log \log x}\right)$ steps.

Proof: First of all we mention that $k = O(\log x)$, simply by taking the 2-adic and 3-adic expansion of x . R.Tijdeman [12] has shown that there exists an absolute constant $C > 0$ such that there is always a number of the form $2^a 3^b$ between $x - \frac{x}{(\log x)^C}$ and x . Now we put $n_0 = x$, and the Tijdeman result implies that there exists a sequence:

$$n_0 > n_1 > n_2 > \dots > n_l > n_{l+1} \quad (3)$$

such that $n_i = 2^{a_i} 3^{b_i} + n_{i+1}$ and $n_{i+1} < \frac{n_i}{(\log n_i)^C}$ for $i=0,1,2,\dots,l..$ Obviously the

sequence of integers n_i obtained via the greedy algorithm satisfies these conditions. Here

we choose $l = l(x)$ so that

$$n_{l+1} \leq f(x) < n_l \quad (4)$$

for some function f to be chosen later.

It follows that we now can write a member of the sequence as a sum of k terms of the form

$2^a 3^b$, where $k = l(x) + O(\log f(x))$. We now have to estimate $l(x)$ in terms of $f(x)$,

and to choose an optimal $f(x)$. Note that if $i < l$, then $n_i > n_l > f(x)$, hence

$n_{i+1} < \frac{n_i}{(\log n_i)^C} < \frac{n_i}{(\log f(x))^C}$. This implies the following inequality:

$$f(x) < n_l < \frac{x}{(\log f(x))^{lC}} \quad (5)$$

Thus we find $l(x) < \frac{\log x - \log f(x)}{C \log \log f(x)}$. We now take

$$f(x) = \exp \frac{\log x}{\log \log x} \quad (6)$$

The function in eqn. (6) is the largest possible (apart from constant) to which we can show

that any number on the interval $[1, f(x)]$ can be written as a sum of $O\left(\frac{\log x}{\log \log x}\right)$ terms

of the form $2^a 3^b$. We want to show that with this function f we also have

$l(x) = O\left(\frac{\log x}{\log \log x}\right)$, i.e that there is a constant $D > 0$ such that

$$l(x) < \frac{1}{D} \frac{\log x}{\log \log x} \quad (7)$$

Thus it suffices to show that:

$$\frac{\log x - \frac{\log x}{\log \log x}}{C \log \frac{\log x}{\log \log x}} < \frac{1}{D} \frac{\log x}{\log \log x} \quad (8)$$

This inequality can be rewritten as

$$D \log \log x + C \log \log \log x < C \log \log x + D \quad (9)$$

which is true if $D < C$ and x large enough. Such a D exists, and so the proof is complete. \square

It is not hard to see that the result from Theorem 1 is the best possible, and that it immedi-

ately generalizes to sums of terms of the form $\prod_{i=1}^s p_i^{a_i}$ for any given finite set of primes

$$S = \{p_1, p_2, \dots, p_s\}.$$

We refer to the representation obtained via the greedy algorithm as a *near-canonic double-based number representation* (NCDBNR).

The greedy algorithm allows us to very quickly find a representation of a given integer in the form of eqn. (2). The fact that this representation may not be canonic can be seen by the following example: if $x=41$, then the above procedure returns $41=36+4+1$, whereas 41 can be canonically represented as a sum of two 2-integers, namely 32 and 9. The greedy algorithm, however, is very easily implemented in practice and it guarantees (Theorem 1) the sparsity of the representation.

From Theorem 1 we require, on average, an asymptotically smaller number of ones in representing integers in the DBNS than the binary number system. In the latter case, the expected number of ones to represent an integer, m , is $\lfloor 0.5 \log_2 m \rfloor$. Therefore, we can expect asymptotic improvements in the performance of algorithms whose complexity depends on the number of nonzero terms in the input data. From a practical point of view it is also important to have some information about the implicit constant associated with the complexity analysis. Applying current results from number theory, we arrive at a rather pessimistic picture; i.e. all we can say about the constant C , used in the Tijdeman theorem, is that it is smaller than 10^9 ! Our computer experiments, however, suggest a much smaller value of this constant. For the particular case $p_1 = 2$ and $p_2 = 3$ this constant has been computed as being close to 1. Numerical experiments supporting this estimation are presented in the Section 5.

4. A Modular Exponentiation Algorithm based on NCDBNR

The use of the NCDBNR and a look-up table of $O(n^2)$ word size (n : is the length of the exponent E) allows an asymptotically fast algorithm, as described below:

Step 1: Precompute all $A^{2^i 3^j} \pmod{p}$ for all i, j such that $2^i 3^j \leq E$ and store the values obtained in a look-up table.

Step 2: Find the NCDBNR of E via the greedy algorithm, that is $E = \sum_{i=1}^k 2^{a_i} 3^{b_i}$.

From Theorem 1 it follows that $k = O\left(\frac{n}{\log n}\right)$.

Step 3: Multiply (modulo p) the corresponding elements $(a_1, b_1), (a_2, b_2), \dots, (a_k, b_k)$ from the look-up table.

We now analyze the time and memory complexity of the proposed algorithm. *Step 1* is pre-computed, therefore the complexity of the algorithm consists only of *Step 2* and *Step 3*. *Step 2* requires $O(k)$ operations, therefore, the expected number of MMs in *Step 3* is

$k = O\left(\frac{n}{\log n}\right)$. The size of the look-up table is $\text{card}(G_{2,3}(E)) \approx \frac{n^2}{3.17}$ words, and if p is

comparable in magnitude to the exponent the size of the table is $O(n^3)$ bits. We are grateful to one of the reviewers for pointing this out.

5. Two examples

In confirming the above estimation of the expected number of ones in the NCDBNR, we offer an example. The selected numbers are 64 and 65-digit prime numbers (212 and 215-bit respectively) [13].

$p_{64} = 3490\ 5295108476\ 5094914784\ 9619903898\ 1334177646\ 3849338784\ 3990820577$

$p_{65} = 32769\ 1329932667\ 0954996198\ 8190834461\ 4131776429\ 6799294253\ 9798288533$

Theorem 1 predicts that the expected number of ones is 27.67 for p_{65} . The application of the greedy algorithms yields 30 integer pairs for p_{64} and 29 integer pairs for p_{65} , where the corresponding pairs of integers (a_i, b_i) (c_i, d_i) are given in Table 1 and Table 2. Table 3 shows the results of performing experiments with 10000 randomly chosen 512-bit integers to find the number of 2-integers needed to represent each of the numbers in the NCDBNR (Theorem 1 predicts that the expected number of ones in NCDBNR should be about $\frac{512}{\log_2 512} \approx 57$). The computational experiments (including random number generations) are performed with the help of NTL 1.7 - a library for performing number theory, which is available at <http://www.cs.wisc.edu/~shoup/ntl/>.

TABLE 1. The pairs of integers corresponding to the NCDBNR of p_{64}

(51,101)	(63,88)	(55,86)	(21,103)	(22,97)	(79,56)
(46,72)	(125,18)	(27,73)	(101,16)	(40,53)	(76,26)
(17,59)	(54,31)	(69,16)	(50,24)	(45,23)	(40,21)
(11,35)	(21,25)	(7,28)	(38,5)	(12,17)	(18,10)
(14,9)	(5,10)	(10,4)	(8,3)	(6,2)	(0,0)

TABLE 2. The pairs of integers corresponding to the NCDBNR of p_{65}

(78,86)	(77,82)	(75,79)	(42,95)	(11,109)	(35,89)
(128,26)	(35,78)	(132,12)	(56,54)	(81,34)	(76,32)
(76,28)	(25,51)	(48,32)	(52,25)	(41,26)	(22,34)
(8,35)	(1,32)	(27,11)	(4,21)	(13,10)	(7,11)
(15,3)	(1,9)	(9,1)	(0,3)	(1,0)	

TABLE 3. The distribution of the number of 2-integers in representing 10000 randomly chosen 512-bit integers

number of 2-integers	52	53	54	55	56	57	58	59
occurrences	1	0	1	4	31	331	951	2322
number of 2-integers	60	61	62	63	64	65	66	67
occurrences	2860	2403	862	207	23	2	1	1

6. Comparison with Published Algorithms

To make a fair comparison among the existing techniques, let us consider that the exponent E is a 512 bit integer. The application of the algorithm, based on the hybrid binary-ternary number system (*HBTNS*) [8] requires, on average, 173 MMs. The algorithm based on a ternary-quintary number system [9] requires, on average, 166 MMs. Our new algorithm requires, on average, 57 MMs, but it also requires 83370 stored values. If this storage requirement is too severe for a particular application, the algorithm can be easily modified in order to reduce the memory requirements.

To be more precise, let us assume that the exponent E is written ('folded') into the form:

$E = 2^{256}E_1 + E_2$ where E_1 and E_2 are 256-bit integers. Then the computation of

$y = A^E \pmod{p}$ can be transformed into

$y = A^{2^{256}E_1 + E_2} \pmod{p} = A_1^{E_1} \pmod{p} A^{E_2} \pmod{p}$, where $A_1 = A^{2^{256}} \pmod{p}$. A_1 can

be precomputed in advance, so we are in position to precompute two look-up tables containing the values of $A^D \pmod{p}$ and $A_1^D \pmod{p}$, for every D of the form $2^a 3^b$, smaller than or equal to 2^{256} . Now the total size of the look-up tables is reduced to

$2 \cdot \frac{256^2}{3.17} \approx 41347$ values. The average number of modular multiplications is increased to

$2 \cdot \frac{256}{\log_2 256} + 1 = 65$; therefore, for the price of eight MMs we can reduce the size of the

look-up table by a factor of two. Separating the exponent into more parts (for convenience their number should be selected as a power of two) we can obtain further reductions of the total look-up table size. Table 4 gives the expected number of MMs and the required number of precomputed values for our new approach and for two other published techniques [3][9].

TABLE 4. Comparison among existing algorithms for ME using precomputations

Algorithms	No.MMs	Storage
Chen-Chang-Yang [9]	166	36027
Brickell-Gordon-McCurley-Wilson [3]	96	10880
New (unfolded exponent)	57	83374
New (2-folded exponent)	65	41347
New (4-folded exponent)	76	20673
New (8-folded exponent)	92	10336
New (16-folded exponent)	117	5168
New (32-folded exponent)	159	2584

7. Conclusions

A new algorithm for modular exponentiation with fixed base has been proposed. It uses a double-based number system representation, with pre-stored computations, and requires a smaller number of modular multiplications than many previously published algorithms. We introduce a folding procedure which allows a flexible trade-off between table storage size and the number of modular multiplications.

8. Acknowledgments

The authors wish to acknowledge the financial support from Natural Sciences and Engineering Research Council of Canada and the Micronet Network of Centres of Excellence.

We are also indebted to Dr. de Weger (Leiden University) for his many valuable comments and discussions regarding the presentation, especially for his help in proving Theorem 1. We would also like to thank Dr.N.J.A.Sloane (AT&T) for his encouraging comments. Finally, we are indebted to the reviewers for their suggestions and critical remarks.

9. References:

- [1] T.ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. on Information Theory*, IT-31, (1985), no.4,pp.469-472
- [2] C.P.Shnorr, Efficient signature generation by smart cards, *Journal of Cryptology*, vol.4,(1991),no.3,pp.161-174
- [3] E.F.Brickell and K.S.McCurley, An interactive identification scheme based on discrete logarithms and factoring, *Journal of Cryptology*, vol.5, (1992),no.1, pp.29-39
- [4] E.F.Brickell, D.M.Gordon, K.S.McCurley and D.B.Wilson, Fast exponentiation with precomputation, *Advances in Cryptology-Proceedings of Eurocrypt'92* (R.A.Rueppel,ed.), *Lecture Notes in Computer Science*, vol. 658, Springer-Verlag, 1993, pp.200-207
- [5] P.de Rooij, Efficient exponentiation using precomputations and vector addition chains, *Proceedings Eurocrypt'94*, pp.403-415
- [6] J.von zur Gathen, Processor-efficient exponentiation in finite fields, *Information Processing Letters*, vol.41, 1992, pp.81-86
- [7] D.E.Knuth, *Seminumerical Algorithms*, second ed., *The Art of Computer Programming*, vol.2, Addison-Wesley, Reading, Massachusetts, 1981
- [8] V.Dimitrov and T.Cooklev, Two algorithms for modular exponentiation using non-standard arithmetics, *IEICE Trans.Fundam.*, 1995, E78-A,(1), pp.82-87
- [9] C.-Y.Chen, C.-C.Chang and W.-P.Yang, Hybrid method for modular exponentiation with precomputations, *Electronics Letters*, vol. 32, 1996,(6), pp.540-541
- [10] B.M.M.de-Weger, *Algorithms for Diophantine equations*, CWI Tracts-Amsterdam, vol. 65, 1989
- [11] G.Hardy, *Ramanujan*, Cambridge University Press, 1940
- [12] R.Tijdeman, On the maximal distance between integers composed of small primes, *Compositio Mathematica*, vol.28, 1974, pp.159-162
- [13] P.Montgomery, A survey on modern integer factorizing algorithms, *CWI Quarterly*, vol.7, 1994,(4), pp.337-366
- [14] A.Borodin and I.Munro, *The computational complexity of algebraic and numeric problems*, *Theory of Computation Series*, vol.1, 1974