Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 105 (2008) 131–137
Information Processing Letters
www.elsevier.com/locate/ipl

Efficient implementation of algorithms for approximate exponentiation

A. Kooshesh, B. Ravikumar *

Department of Computer Science, Sonoma State University, Rohnert Park, CA 94128, USA

Received 3 November 2005
Available online 19 August 2007
Communicated by F. Dehne

Abstract

We present efficient implementations of algorithms for the following fundamental problem: Given as input three positive integers x, y and j, compute the leading j digits of x^y. A special case of this problem ($k = 2$ and $j = 1$) was recently studied by Hirvensalo and Karhumaki [M.Hirvensalo, J.Karhumaki, Computing partial information out of uncomputable one—The first digit of 2^n to base 3 as an example, in: Mathematical Foundations of Computer Science, in: Lecture Notes in Computer Science, Springer-Verlag, Inc., 2002] for which they presented a polynomial time algorithm. Specifically an algorithm of bit complexity $O(n^2 \log^3 n \log\log n)$ where $n = |y|$ is the number of digits in y. But their algorithm is not efficient in practice. For example, finding the leading digit of 2^y where y is a 500 digit positive integer takes several hours. Hirvensalo and Karhumaki's algorithm is based on computing a rational approximation to $\ln 2$ (and a few other constants) to a high-degree of precision. Our approach is fundamentally different from theirs: we use a modified addition chain algorithm in which the multiplication is truncated to varying number of digits at various steps, followed by a self-tester that validates the truncation. Our algorithm runs several orders of magnitude faster. For example, on an input in which x and y have a few thousand digits, our program computes the leading 1000 digits in under 3 minutes. Since the approximate exponentiation has many application [B.Ravikumar, A Las Vegas randomized approximation algorithm for approximate exponentiation and its applications, in preparation] we hope that our algorithm will stimulate further research on this problem.

Keywords: Algorithms; Software design and implementation

1. Introduction

Exponentiation (computing x^y for given x and y) is perhaps the most important operation after the four basic arithmetic operations. Knuth [1] devotes a large section for computing x^y efficiently. Unlike the basic arithmetic operation, the size of the output x^y is exponential in the size of the input (which $|x| + |y|$, the total number of digits in x and y) and hence the time complexity of computing x^y is inherently exponential.[1] Modular exponentiation (computing $x^y \bmod n$) is fundamental to the RSA algorithm and has been a subject of numerous papers. The (full) exponentiation is inherently in-

* Corresponding author.
E-mail address: ravi93@gmail.com (B. Ravikumar).

[1] Throughout the paper, we will assume that the inputs as well as the output are in decimal. The actual base does not significantly change the nature of the problem so long as it is a positive integer.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2007.08.010

tractable since the output size is exponential in the input size. In an elegant work, Hirvensalo and Karhumaki [6] introduced a natural variation of the problem, namely to compute the first (leading) digit of 2^y. They presented a simple polynomial time algorithm to solve this problem. But its correctness proof is quite non-trivial since is based on a deep result in transcendental number theory due to Baker [2]. We begin by presenting an outline of the algorithm of Hirvensalo and Karhumaki [6].

For an integer $y \geqslant 0$, let d be the leading digit of 2^y in decimal. Their algorithm takes y as input and produces d as output. (We have modified their presentation slightly; they work with ternary number system.) First they show how to compute the number of decimal digits in 2^y. It is easy to see that this number is $1 + \lfloor y * \log_{10} 2 \rfloor$. If r is a rational approximation to $\log_{10} 2$ so that $|\log_{10} 2 - r| < c(1/|y|^T)$ for some (known) constants c and T, then $1 + \lfloor yr \rfloor$ is the exact number of digits in 2^y. This assertion is a consequence of Baker's theorem. Hirvensalo and Karhumaki propose to use the Taylor series $-\log(1-x) = x + \frac{x^2}{2} + \frac{x^3}{3} + \cdots$ truncated to m terms (for an $m = \mathrm{O}(|y|)$) to get such an r. They use rational arithmetic with unlimited precision integers to represent the numerator and the denominator of this expression. The final step is to use the algorithm for length computation to find the leading digit. This is done by determining the unique d, $1 \leqslant d \leqslant 9$, such that $d10^{l-1} \leqslant x^y < (d+1)10^{l-1}$ where l is the number of digits in x^y. Their algorithm, although elegant and theoretically appealing, is far from practical. In contrast, one of the algorithms presented in this paper can find the leading 1000 digits of 2^y where y is a 2000 digit number is less than 1 minute. In view of several interesting applications for this problem, it is important to develop such a practical algorithm for this problem. This is the main contribution of our paper.

Our approach is to modify an algorithm that computes the full exponentiation (x^y) fast (i.e., with only $\mathrm{O}(\log y)$ multiplications—we use the addition chain algorithm [1]). We need to modify it since the full exponentiation is not a polynomial time (or even polynomial space) algorithm since the size of the output is exponential in the size of the input. But since we only need the leading (few) digit(s) of the result, we truncate the various intermediate numbers generated to varying degrees in such a way that the final result will have at least the j leading digits.

There are several questions that need to be addressed to make this approach viable: (a) how many digits of various intermediate results should we keep? (b) how do we know that the accumulation of errors will not corrupt the leading j digits of the final result? (c) how efficient will the resulting algorithm be? We will briefly address these questions below: (a) Suppose there are totally k multiplications performed by the addition chain algorithm to compute x^y. Then, it turns out that truncating the ith multiplication at $\mathrm{O}(k + j - i)$ digits is sufficient to give the end result correctly *almost always*. In [8], we provide a theoretical analysis of the algorithm. But the key feature of our algorithm is how we address (b): we have developed a simple self-tester that certifies that at each step, *the truncated product computed has a known number of correct leading digits*. If the self-tester fails, our simple version of the algorithm terminates with a message that the algorithm has failed to compute the result. The probability of error is very low as we demonstrate it experimentally. (In our experiments, it is less than 0.002%.) A more complicated version of the algorithm never fails is presented in [8].

Our goal was to compute the leading (most-significant) thousands of digits of x^y where x and y both have thousands of digits, in a few minutes. This goal has been accomplished by the implementation presented in this paper. We believe that our current implementation is close to the best possible way to find the leading digits of x^y. We have also implemented the algorithm of Hirvensalo and Karhumaki [6] in the most efficient way possible.

The rest of the paper is organized as follows: In Section 2, we describe our implementation of Hirvensalo–Karhumaki algorithm [6]. In Section 3, we describe the implementation details of our algorithm. In Section 4, we present experimental results on the performance of our algorithm. Finally, we conclude with a summary of the paper and directions for future work.

2. Implementation of Hirvensalo–Karhumaki algorithm

We briefly describe the algorithm of Hirvensalo and Karhumaki [6] to compute the leading digit of x^y. We have modified their algorithm to work with decimal system rather than ternary system. We have also generalized the algorithm so that it computes the leading digit of 2^y rather than x^y. The basic idea behind the algorithm of [6] is to find the number of digits in x^y and use this computation to determine the leading digit of x^y. The number of digits in x^y is given by $\lfloor y \log_{10} x \rfloor + 1$. By Baker's theorem, it follows that if q is a rational approximation to $\log_{10} x$ so precise that

$$|q - \log_{10} x| \leqslant \frac{1}{\ln 10} \frac{1}{n^{14}}$$

then $l = \lfloor qy \rfloor + 1$ is the *exact* number of digits in x^y. We can use this length information to obtain the leading

digits of x^y as follows. The leading digit j, $1 \leqslant j \leqslant 9$, is given by the unique j such that $j10^{l-1} \leqslant x^y < (j+1)10^{l-1}$.

The above steps are presented more formally below: The first procedure *approx* $\mathrm{Log}(x, n)$ computes a rational number q such that $|\ln(x) - q| < 1/|y|$[14].

```
rational ApproxLog(a, n)
// assume a, n are positive integers,
//a > 1.
 int m, sum, num, denom, Exp1, Exp2;
 m = 2+ round((14*log(n)+log(a))/
     log(a/(a-1)));
// this call to the \logarithm can be
//very approximate using just
// the standard single precision
//log function in C++
// or with Digits set to 10 in Maple
num = 0;
denom = 1; Exp1 = a; Exp2 = a-1;
for j = 1 to m do {
  denom = denom * j * Exp1;
  Exp1 = Exp1 * a;
  num = num * denom + Exp2;
  Exp2 = Exp2 * (a-1);
 }; //end for loop
 return numerator/denom;
end; //ApproxLog
```

The next procedure computes the number of digits in the decimal representation of x^y:

```
int NumDecimalDigits(x,y)
 // x, y are positive integers, x > 1.
 int q;
 q = ApproxLog(x,y)/ApproxLog(10,y);
 Let qnum be the numerator of q and
   qdenom be the denominator of q.
 r = quotient(qnum*n, qdenom);
 return 1 + r;
end; //NumDecimalDigits
```

The last procedure returns an integer from 1 to 9 that represents the leading digit of x^y:

```
int LeadDigit(x,y)
int length, denomq, numq, q, j,
    logiplus1;
boolean done;
length = NumDecimalDigits(n);
denomq = ApproxLog(10,y);
numq = ApproxLog(x,y);
q = numq/denomq; done = false;
while (!done) do {
  logiplus1 = ApproxLog(j+1,n)/denomq;
  if (y < (length-1) * q + logiplus1)
  then done = true;
  else j = j+1;
 }; // end while loop
 if (done) then return j else return 9;
end;
```

In the above procedures, calls to all the basic arithmetic operations (+, *, quotient, etc.) are unlimited precision arithmetic operations and are supported in the **gmp** package. Multiplication is implemented using Karatsuba's algorithm [1] although when the number of digits in really large (well over 10^6), some simpler variations of Schonage–Strassen's algorithm based on Fast Fourier Transform can be faster than Karatsuba's algorithm. For the range (between 10^3 and 10^5 digits) over which we have tested the algorithm, GMP's native implementation (based on Karatsuba's algorithm) seems to be the most appropriate.

In the procedure *LeadDigit*, we use sequential search for j which in the worst-case can go through the *while* loop 9 times. In the actual implementation, we use binary search that reduces the number of iterations to at most 4. We also considered some ways to speed up the computation of very high precision logarithm computation. As noted by Crandall [4] and others, the alternating series for $\ln(x)$ converges slowly and speeding it up is a challenging computational problem. We have tried the acceleration technique of Cohen et al. [3], but the overhead makes it less efficient that direct computation of $\ln(x)$ for all the examples we tried.

3. Our algorithm

We have implemented our exponentiation algorithm in Maple and in C++. In this paper, the C++ implementation (using the package **gmp**) is described. We start with an algorithm for computing x^y using an addition chain [1]. Recall the basic idea behind an addition chain: An addition chain for an integer b is a sequence of integers $n_0, n_1, \ldots, n_k = n$ where $n_0 = 1$, $n_k = b$, and for $i \geqslant 1$, $n_i = n_j + n_k$ for some $j, k < i$. The length of the chain is k. It is easy to see that the number of multiplications necessary to compute a^b is at most k if b has an addition chain of length $k+1$: basically, each addition $n_i = n_j + n_k$ in the chain addition corresponds to a multiplication in the computation of a^b since $a^{n_i} = a^{n_j} \times a^{n_k}$. As described in Section 1, our algorithm creates an addition chain for the exponent, but instead of doing full multiplication, it truncates the multiplication, and uses self-checking to make sure that the leading digits maintained by the algorithm is correct. Our algorithm takes as input, the base and exponent

(a and b, respectively), the number of desired digits (n), and the size of the addition chain window (w). As output, it either (correctly) displays the first n digits of a^b or it displays "0" as an admission of failure to compute the result. This happens in the following circumstance. Let $n_0, \ldots, n_k = b$ be the addition chain that is used by the algorithm to compute a^b. Suppose that the algorithm has predetermined that the number of leading digits of a^{n_j} that would be computed is L_j. The next step involves computing the leading L_{j+1} digits of $a^{n_{j+1}}$. Suppose $n_{j+1} = n_k + n_s$ in the addition chain. If the leading L_{j+1} digits of a^n_{j+1} can not be determined from the leading L_k digits of a^{n_k} and the leading L_s digits of a^{n_s}, then the algorithm would terminate with 0 as the output.

In what follows, we first briefly describe the addition chain algorithm that we have implemented. Next, we present a high-level description of our implementation and provide a detail analysis of each step.

Addition Chain. Addition Chain is a method for calculating the power exponentiation efficiently. To compute a^b, we calculate

$$a^1, a^{b_1}, a^{b_2}, \ldots, a^{b_m}\ (= a^b),$$

where

$$a^{b_i} = a^{b_j} \times a^{b_k}, \quad j, k < i.$$

Note that the length of the chain—the number of multiplications to compute a^b is m. Thus, to compute a^b optimally, we need to minimize the length of the chain, a problem known to be NP-hard [5].

A variation of the addition chain algorithm that asymptotically outperforms many of heuristics for computing a short chain is called the *window method* [1,7]. Window method modifies the above rules as follows.

$$b_i = 2 \times b_{i-1},$$

doubling rule, $a^{b_i} = (a^{b_{i-1}})^2$,

$$b_i = b_{i-1} + b_k, \quad b_k \in D,$$

star chain rule, $a^{b_i} = a^{b_{i-1}} \times a^{b_k}$.

For a window size, w, D is a the set of odd integers in $2^r - 1$, where $1 \leqslant r \leqslant w$.

Given a, b, and w, we used the following algorithm to compute the a chain for b using the window method.

Step 1: Compute $D = \langle 1, 2, 3, \ldots, 2^w - 1\rangle$. Note that 2, even though it is not technically a member of D, is required for the computation of members of D.

Step 2: Repeatedly apply the following two steps to $B = 1c_1c_2 \ldots c_{l-1}$, the binary representation of b, starting with its most significant digit.

Phase 1: Let the most significant w binary digits in the remaining sequence be $1s_1s_2 \cdots s_{w-1}$ with s_{w-l} the last binary digit with value 1. That is, the digits $s_{w-l+1}, s_{w-l+2}, \ldots, s_{w-1}$ are all zero digits. First apply the doubling rule ($b_i = 2 \times b_{i-1}$), s_{w-l} times. Next apply the star chain rule. Finally, we apply the doubling rule l times. Remove the most significant w bits from the sequence.

Phase 2: Let the first l_0 binary digits of the sequence be zero digits. Apply doubling rule l_0 times. Remove the l_0 bits from the sequence. Note that at this point either the sequence is empty or its most significant digit is a 1.

Observe that Phase 1 processes exactly w bits of the sequence whereas the number of bits consumed by Phase 2 depends on the sequence.

Consider the following simple example.

1. Let $b = 1578$ and $w = 3$ so that $D = \langle 1, 2, 3, 5, 7\rangle$.
2. $B = 11000101010$.
3. According to Phases 1 and 2 of the above algorithm, B will be parsed as follows:

$$\underbrace{11}_{3}0/00//\underbrace{101}_{5}/0//\underbrace{1}_{1}0///$$

Note that "/" means the parsing in Phase 1 and "//" means the parsing in Phase 2; the end of one cycle. Then, the chain for $b = 1578$ becomes

$$\langle 3, 6/, 12, 24//, 48, 96, 192, (+5)197/,$$
$$394//, 788, (+1)789, 1578///\rangle.$$

This means that the number of multiplications (the length of the addition chain) for this example is 12. To compute a^{1578}, we do:

Compute a^3 and a^5
Let $AC = [a^3, 0, 0, 0, 0, 0, 0, a^5, 0, 0, a, 0]$
Let $p = 1$
For each i = 1, 12 **do**
  if $AC[i]$ is zero (a doubling rule)
    $p \leftarrow p \times p$
  else
    $p \leftarrow p \times AC[i]$
end-for
return p

The Power Algorithm. We now present a high-level description of the implementation of our power algorithm.

Input: a, b, n, and w.
Output: The first n digits of a^b using window size w.
Compute $AC[1,\ldots,m]$, the addition chain for b.
  m is the number of multiplications.
$p \leftarrow 1$
curDigits $\leftarrow$ *expectedDigits*
for each AC[i], $1 \leqslant i \leqslant m$, **do**
  # Perform one multiplication
  expectedDigits $\leftarrow n + (m-i) \times C$
  # Require C digits per multiplication.
  case 1: p_{i-1} has not been truncated yet.
    if AC[i] is a star-chain rule **then**
      $p_i \leftarrow p_{i-1} \times AC[i]$
    else # doubling rule
      $p_i \leftarrow p_{i-1} \times p_{i-1}$
    end-if
  end-case 1
  case 2: p_{i-1} has been truncated.
    if AC[i] is a star-chain rule **then**
      $p' \leftarrow (p_{i-1}+1) \times AC[i]$
      $p_i \leftarrow p_{i-1} \times AC[i]$
    else # doubling rule
      $p' \leftarrow (p_{i-1}+1) \times (p_{i-1}+1)$
      $p_i \leftarrow p_{i-1} \times p_{i-1}$
    end-if
    if p' and p_i do not share *expectedDigits* digits,
    exit and return 0
    curDigits $\leftarrow \min\{$*expectedDigits* $+ C'$, *curDigits*$\}$
  end-case 2
  truncate p_i to contain *curDigits* digits
end-for
return the first n digits of p

Consider the sequence of numbers, $p_1, p_2, \ldots, p_m$, that are generated through the iterations of the loop as we compute the first n digits of a^b. Note that p_m must have at least n correct leading digits of a^b.

During the ith iteration of the loop, p_{i-1} has at least *expectedDigits* $= n + (m - i - 1) \times C = d_{i-1}$, correct leading digits. C is a constant value. The number of correct leading digits of p_i, denoted by d'_i, is the number of correct leading digits that

Star-Chain Rule: $p_{i-1} \times AC[i]$ and $(p_{i-1}+1) \times AC[i]$ have in common.
Doubling Rule: p_{i-1}^2 and $(p_{i-1}+1)^2$ have in common.

If $d'_i < d_i$, then we have to either stop the algorithm and declare failure, or we have to increase C and restart the algorithm so that each iteration provides more correct digits to the next iteration. Thus the choice of *expectedDigits* directly impact the running-time and the correctness of the result. Setting it too low (a smaller C) makes the algorithm more efficient as the numbers will have fewer digits. However, this could result in insufficient number of correct digits in the final computation. On the other hand, setting *expectedDigits* too high will make the computation too slow as we will have to multiply larger numbers. The challenge is to find a small value that succeeds with a high probability.

During our experimentation with different values of C, we learned that many times at a given iteration we had more than sufficient number of correct digits before we truncated the number prior to the next iteration. We had to come up with a way to preserve those additional correct digits for situations when we did not have sufficient number of correct digits in future iterations. To do so, at each iteration, we still required to have d_i correct digits. However, we allow p_i to retain as many as $d_i + C'$ (*curDigits* in the notation of the algorithm), for some constant $C' > 0$. That choice allowed us to set $C = 1$, one digit per multiplication. We set $C' = 20$.

To see why this simple technique allowed for such great efficiency, consider the computation of $p_i = p_{i-1}^2$ using the doubling rule. The number of digits in p_i is $2 \times p_{i-1} - e$, where $e \in \{0, 1\}$. Given $(p_{i-1}+1)^2 = p_{i-1}^2 + 2 \times p_{i-1} + 1$, the number of most significant digits that p_{i-1}^2 and $(p_{i-1}+1)^2$ share is at most $d_{i-1} - e$. This suggests that if we were to set $C = 1$ so that the number of required correct digits of p_i be $d_i = d_{i-1} - 1$ (d_{i-1} is the number of digits in p_{i-1}), many times we will fail to meet this requirement. However, if $e = 0$ and the most significant digit of p_{i-1} is less than 5, then $C = 1$ would work fine. Allowing $d_{i-1} \leqslant |p_{i-1}| \leqslant d_{i-1} + C'$ helps to accumulate correct digits beyond the minimum requirement, and up to a fixed limit, whenever they are available. Digits saved this way are spent in cases when the most significant digit of p_{i-1} is larger than 4, or the carry of $2 \times p_{i-1} + 1$ cuts into the first d_{i-1} digits of p_{i-1}^2. This proved to be an effective strategy.

The correctness of the above algorithm and its time complexity are stated below without proof. The proofs can be found in [8].

Theorem 1. *On input a, b, n and w, if the above algorithm returns $d \neq 0$, then the first n-digits of a^b is d.*

In the following, we assume that the multiplications in the above algorithm are performed using Karatsuba's algorithm [1]. This takes $O(l^\alpha)$ digit operations

to compute the product of two l digit integers where $\alpha = \log_2 3$.

Theorem 2. *The number of bit operations performed by the above algorithm on input a, b and n is upper-bounded by $|b|(|a| + |b| + n)^{\alpha}$. This bound is $O(l^{\alpha+1})$ when a is a constant and $|b| = l$.*

This bound is $O(l^{\alpha+1})$ when a and n are constants (as in [6]) and $|b| = l$ and is only slightly better than the time complexity $O(l^{\alpha+1} \log^2 l)$ of the algorithm of [6]. So it is surprising that our algorithm is significantly faster than theirs in practice, as shown in the next section.

4. Experimental results on the performance of the algorithms

In this section, we present experimental results that describe the performance of our algorithm. In order to demonstrate the performance of our exponentiation algorithm, we ran our implementation with a fixed number of digits in the base (500) and required 100 digits for each computation. We then varied the number of digits in the exponent. Each experiment was repeated 10 times to get a representative value. These parameters were also used to compute the actual number of digits in the exponentiation. The results are given in the upper figure.

In the next series of experiments, we fixed the number of digits in the base to 500 and of the exponent to 1500 and varied the number of digits in the output. The result of these experiments are shown in the lower figure.

Finally, we fixed the number of digits in the base to 50 and of the exponent to 100 and ran the algorithm 100,000 times. There were only 17 failures. That is, for 17 cases our algorithm stopped and reported that it could not correctly provide the answer. At this point, we could have increased the constant C in order to get a correct answer or used one of the other methods to recover from the failure. These alternatives will be explained in the full version of the paper.

The efficiency of our algorithm is demonstrated in the figure on the left as it shows that it takes about $2\frac{1}{2}$ minutes to compute the lead 100 digits of a^b when a and b have 500 and 4000 digits, respectively. This number changes slightly when we compute the (exact) number of digits in a^b along with the leading j digits of it.

In comparison, Hirvensalo and Karhumaki's algorithm required over 6 minutes to compute the leading digit of 2^b where b is a 100 digit number and more than

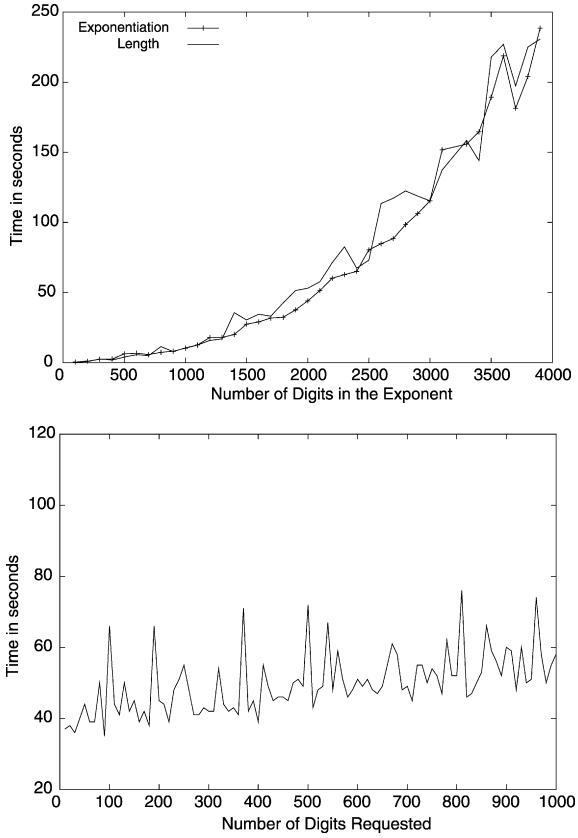

Fig. 1.

1 hour to compute the leading digit of 2^b when b is a 200 digit number.

5. Conclusions

The problem of approximate exponentiation is a fundamental one and is likely to find many applications. A breakthrough in identifying this problem and presenting the first polynomial time algorithm was made by Hirvensalo and Karhumaki [6]. In this paper, we have made a major advance in providing a practical algorithm for a more general version of their paper. In a companion paper, several interesting applications of this problem are presented. We expect the list of applications of this problem to grow since it provides an alternative to floating point approximation in several contexts. We believe that our implementation is very efficient and that it would be very hard to significantly improve it. However, much theoretical work remains to be done on the precise performance of the algorithm.

References

[1] Semi-Numerical Algorithms, The Art of Computer Programming, vol. 2, third ed., Addison-Wesley, Inc., Reading, MA, 1998.

[2] A. Baker, Transcendental Number Theory, Cambridge University Press, 1975.

[3] H. Cohen, F.R. Villegas, D. Zagier, Convergence acceleration of alternating series, Experimental Mathematics 9 (1) (2000) 1–12.

[4] R. Crandall, Projects in Scientific Computation, TELOS/Springer-Verlag, 1994.

[5] P. Downey, B. Leong, R. Sethi, Computing sequences with addition chain, SIAM J. Comput 10 (3) (1981) 638–646.

[6] M. Hirvensalo, J. Karhumaki, Computing partial information out of uncomputable one—The first digit of 2^n to base 3 as an example, in: Mathematical Foundations of Computer Science, in: Lecture Notes in Computer Science, Springer-Verlag, Inc., 2002.

[7] N. Kunihoro, H. Yamamoto, Window and extended window methods for addition chain and addition–subtraction chain, IECE Trans. Fundamentals E81-A (1) (Jan. 1998) 72–81.

[8] B. Ravikumar, A Las Vegas randomized approximation algorithm for approximate exponentiation and its applications, 2004.