Should one always use Repeated Squaring for Modular Exponentiation?

Shmuel T. Klein
Department of Computer Science
Bar Ilan University, Ramat-Gan 52900, Israel
Tel: (972–3) 531 8865 Fax: (972–3) 736 0498
tomi@cs.biu.ac.il

Abstract: Modular exponentiation is a frequent task, in particular for many cryptographic applications. To accelerate modular exponentiation for very large integers one may use repeated squaring, which is based on representing the exponent in the standard binary numeration system. We show here that for certain applications, replacing the standard system by one based on Fibonacci numbers may yield a new line of time / space tradeoffs.

Keywords: Design of algorithms, modular exponentiation, Fibonacci number system, cryptography

1. Introduction

Modular exponentiation is defined as the task of raising a number a to a power m and considering the result modulo some integer N. This is a frequent and time consuming operation, and has many applications, in particular in cryptography. In a typical setting, a, m and N are large integers, say of the order of 2^{1024}, so it is not feasible to calculate a^m mod N by using $m-1$ multiplications, each followed by a modulo operation. The standard solution to this problem is using *repeated squaring* and appears in many handbooks on algorithms, such as [4, 2, 8] to cite just a few.

To improve readability, we shall not always explicitly mention that the multiplications are to be taken modulo N, which is fixed throughout the paper. Note that instead of calculating

$$a^8 = \underbrace{a \times a \times a \times \cdots \times a}_{8 \text{ factors}},$$

the number of multiplications can be reduced by repeatedly squaring the results:

$$a^8 = \Big(\big(a^2\big)^2\Big)^2.$$

If m is not a power of two, it can be expressed as a sum of such powers, giving, for example, $a^{12} = a^8 \times a^4$. For the general case, consider the standard binary representation of m as a sum of powers of 2, that is $m = \sum_{i=0}^{\lfloor \log_2 m \rfloor} b_i 2^i$, where each $b_i \in \{0, 1\}$. Then

$$a^m = a^{b_0} \times a^{2b_1} \times a^{4b_2} \times \cdots \times a^{2^i b_i} \times \cdots.$$

The procedure is thus as follows: prepare a list of basis items a, $a^2, \ldots, a^{2^i}, \ldots$ where each element is obtained by taking its predecessor in the list, squaring it, and reducing the

result modulo N; then take the subset of this list corresponding to the 1-bits in the binary representation of m and multiply the elements of this subset. Denoting the number of 1-bits in the binary representation of m by $h(m)$, the number of multiplications is thus $\lfloor \log_2 m \rfloor + h(m) - 1$.

This is not necessarily the minimum number of required multiplications. For example, for $m = 15$, $\lfloor \log_2 m \rfloor + h(m) - 1 = 6$, but a^{15} can be evaluated in 5 operations, calculating first $d = a^5 = (a^2)^2 \times a$ in 3 multiplications, and then $a^{15} = d^3 = d^2 \times d$ in two more multiplications; see Knuth [10, Section 4.6.3] for an investigation of the function $l(m)$, giving the smallest number of multiplications necessary to calculate a^m. We are, however, not interested in finding the minimum number for each given exponent m, but are rather looking for a general algorithm, giving good average performance when applied with a large number of possible values m. Repeated squaring is one such algorithm, and the point of this work is to show that for certain applications, a different general evaluation procedure might be preferable.

2. Alternatives to repeated squaring

2.1 Standard k-ary number system

As mentioned above, the standard evaluation algorithm is based on representing the exponent m in the standard binary number system as $m = \sum_{i=0}^{\lfloor \log_2 m \rfloor} b_i 2^i$, with $b_i \in \{0, 1\}$, yielding $\lfloor \log_2 m \rfloor + h(m) - 1$ multiplications. The first term can be reduced to $\lfloor \log_k m \rfloor$ for $k > 2$, if one uses the standard k-ary number system in which one can represent m as $m = \sum_{i=0}^{\lfloor \log_k m \rfloor} c_i k^i$, with $c_i \in \{0, 1, \ldots, k-1\}$. However, the basis elements $a^{k^{i+1}}$ cannot be anymore evaluated by a single multiplication from preceding basis elements:

$$a^{k^{i+1}} = a^{k^i \cdot k} = \left(a^{k^i}\right)^k,$$

so for $k = 2$, only one multiplication is needed (this is squaring), but for $k = 3$, one needs two multiplication, for $k = 4$ also two multiplications are sufficient (squaring twice), etc.

k	# mult per basis elmt $l(k)$	# basis elements (digits)	# mult for all basis elmts	avg # mult per digit	avg # mult for all digits	total # mult	total # mult Horner
2	1	*1.000*	1.000	1/2	**0.500**	1.500	1.500
3	2	*0.631*	1.262	1	**0.631**	1.893	1.682
4	2	*0.500*	1.000	6/4	**0.750**	1.750	1.375
5	3	*0.431*	1.292	9/5	**0.775**	2.067	1.637
6	3	*0.387*	1.161	13/6	**0.838**	1.999	1.483
7	4	*0.356*	1.425	17/7	**0.865**	2.299	1.730
8	3	*0.333*	1.000	22/8	**0.917**	1.917	1.292
9	4	0.*316*	1.262	26/9	**0.911**	2.173	1.542

TABLE 1: *Number of multiplications for the standard k-ary system*

Table 1 presents the relevant data for $k = 2, \ldots, 9$. The second column gives the number $l(k)$ of required operations for the calculation of each basis element. The number of necessary basis elements is the number of k-ary digits, which is $\log_k N$, where N is the modulus mentioned above, and this is normalized in the third column, which gives the number of basis elements in units of $\log_2 N$. The fourth column is the total number of multiplications needed for all the basis elements, again in units of $\log_2 N$.

After having evaluated the basis elements, the members of a selected subset of them have to be multiplied. If $m = \sum_{i=0}^{\lfloor \log_k m \rfloor} c_i k^i$, the basis element a^{k^i} is raised to power c_i in $l(c_i)$ multiplications if $c_i > 0$, and one more multiplication is needed per basis element with $c_i > 0$ to get the final product. Assuming that the exponent m is chosen at random, each of the digits $0, 1, \ldots, k-1$ appears in each position with probability $1/k$. The expected number of multiplications for a given digit is thus $\frac{1}{k} \sum_{i=1}^{k-1} (l(i) + 1)$, and these values appear in the fifth column of Table 1. The sixth column is then the average number of multiplications taking all the digits into account, and the seventh column, headed total # mult, is the total sum of operations for both the basis elements and the product of the elements of the subset, again in units of $\log_2 N$. One sees that though interestingly, the total number of multiplications is not monotonically increasing with the order k of the number system, the minimum is still reached for the binary case $k = 2$, so among these alternatives, binary squaring still seems to be the best choice.

There are nevertheless options to reduce the number of required multiplication [9]. The so-called *k-ary method* uses precomputed values $a^2, a^3, \ldots, a^{k-1}$ and Horner's rule as follows. Setting $r(k) = \lfloor \log_k m \rfloor$, the equation for the representation of m in basis k can be rewritten as

$$m = \sum_{i=0}^{r(k)} c_i k^i = c_0 + k(c_1 + k(c_2 + \cdots + k(c_{r(k)-1} + k c_{r(k)}) \cdots)).$$

This suggests that a^m can be evaluated iteratively as follows, working from the innermost parentheses outwards:

```
precompute A[j] ⟵ a^j for j = 1, 2, ..., k − 1
R ⟵ 1
for i ⟵ r(k) to 0 by -1
        R ⟵ R^k
        if c_i > 0 then R ⟵ R × A[c_i]
```

The expected number of multiplications in each iteration for a randomly chosen m is $l(k) + (k-1)/k$, as $l(k)$ operations are necessary to raise R to the kth power and one more to multiply with $A[c_i]$, but c_i will be zero with probability $1/k$. The last column of Table 1 displays the total number of required multiplications in units of $\log_2 N$. As can be seen, it is possible to get better values than the 1.5 needed for $k = 2$, in particular for values of k that are powers of 2, for which raising to the power k is done by $\log_2 k$ repeated squarings.

In fact, the k-ary method for $k = 2^s$ can be interpreted as processing the standard binary representation of m by consecutive windows of s bits in each iteration, and the savings relative to the original method with $k = 2$ are due to the fact that the set of possible values in each s-bit window, $\{0, 1, \ldots, 2^s - 1\}$, is fixed relative to the size $\log_2 N$

of m. This can be generalized to what is known as the *window method*, which achieves even greater savings for certain exponents, when there are long runs of zeros in the standard binary representation of m. The idea is to evaluate the exponentiation as above by windows of limited size, but not to restrict the windows to be adjacent, so that stretches of zeros may be skipped. For example, if $m = \underline{1010}0000000\underline{1101}000$ in binary, where the chosen windows are underlined, then one could evaluate $a^m = \left((a^5)^{2^7} \times a^{13}\right)^{2^3}$. However, for our assumption of a *random* exponent m, the expected length of a run of zeros is just 1, so that the windows method does not yield any additional savings over the k-ary method.

2.2 Fibonacci k-ary number system

To improve processing time, the above methods tried to shorten the representation of the exponent m. A complementing approach may suggest to try alternative representations which might be longer than the $\log_2 N$ bits needed for the standard binary form, but possibly sparser. Indeed, such representations do exist, and they use the Fibonacci sequence as basis elements instead of the sequence of powers of 2 [7].

Fibonacci numbers of order $k \geq 2$, denoted by $F_n^{(k)}$, are defined by the following recurrence relation:

$$F_n^{(k)} = F_{n-1}^{(k)} + F_{n-2}^{(k)} + \cdots + F_{n-k}^{(k)} \qquad \text{for } n > 0,$$

and the boundary conditions

$$F_0^{(k)} = 1 \qquad \text{and} \qquad F_n^{(k)} = 0 \quad \text{for } -k < n < 0. \tag{1}$$

Let us first consider the standard Fibonacci numbers of order 2, and use F_n as shortcut for $F_n^{(2)}$.

Any integer B can be represented by a binary string $c_r c_{r-1} \cdots c_2 c_1$ of length r such that $B = \sum_{i=1}^{r} c_i F_i$. This can be seen from the following procedure producing such a representation: given the integer B, find the largest Fibonacci number F_r smaller or equal to B; then continue recursively with $B - F_r$. For example, $45 = 34 + 8 + 3 = F_8 + F_5 + F_3$, so its binary Fibonacci representation would be 10010100. Moreover, the use of the *largest* possible Fibonacci number in each iteration implies the uniqueness of this representation. Note that as a result of this encoding procedure, there are never consecutive Fibonacci numbers in any of these sums, implying that in the corresponding binary representation, there are no adjacent 1s. Similarly, in the binary representations corresponding to the system based on $\{F_n^{(k)}\}$, there are no k consecutive 1s.

The average number of multiplications being composed of the number of bits plus the average number of 1-bits, we need to evaluate these numbers for k-order Fibonacci codes.

2.2.1 Length of k-ary Fibonacci representation

For fixed order k, $F_n^{(k)}$ can be represented as a linear combination of the nth powers of the k roots of the corresponding polynomial $P(k) = x^k - x^{k-1} - \cdots - x - 1$. $P(k)$ has only one real root that is larger than 1, which we shall denote by $\phi_{(k)}$, the other $k-1$ roots are

complex numbers with norm < 1 (for $k = 2$, the second root is also real and its absolute value is < 1) [1]. Therefore, when representing $F_n^{(k)}$ as such a linear combination, the term with $\phi_{(k)}^n$ will be the dominant one, and the others will rapidly become negligible for increasing n. It follows that the number of bits needed to represent the integer N in the k-ary Fibonacci representation is about $\log_{\phi_{(k)}} N = (1/\log_2(\phi_{(k)}))\ \log_2 N$. The values of $\phi_{(k)}$ and $1/\log_2(\phi_{(k)})$ for $k = 2, \ldots, 6$ are given in the second and third columns of Table 2 below. The last line of the table is the limiting value as $k \longrightarrow \infty$, which is equivalent to the standard binary system based on powers of 2.

2.2.2 Density of 1-bits in the Fibonacci representation

To evaluate the average number of 1-bits, we shall assume that for each given length r of the representation of the exponent m, all the possible values of m appear with equal probability. For the standard binary numeration system based on the powers of 2, the probability of a 1-bit is then $\frac{1}{2}$. For the Fibonacci representations, the appearance of a 1 is more restricted, e.g. for $k = 2$, a 1 in a given bit position implies that the adjacent positions hold zeros.

Let us first restrict ourselves to the standard Fibonacci case $k = 2$, and refer to the binary strings representing numbers in the 2-ary Fibonacci number system as F-strings. Thus 100 is an F-string, but 110 is not. Define T_r as the number of different F-strings of length r, and Q_r as the total number of 1's in all the F-strings of length r, $r \geq 0$. We thus have $T_0 = Q_0 = 0$, $T_1 = 2$, $Q_1 = 1$, $T_2 = 3$ and $Q_2 = 2$, and the set of F-strings of length 3 being $\{000, 001, 010, 100, 101\}$, we get $T_3 = 5$ and $Q_3 = 5$. In general, one gets

$$T_r = T_{r-1} + T_{r-2},$$

since the F-strings of length r can be generated by either prefixing an F-string of length $r-1$ by 0, or by prefixing an F-string of length $r-2$ by 10. The T_r are thus Fibonacci numbers, and according to the boundary conditions defined in eq. (1), $T_r = F_{r+1}$ for $r \geq 1$. Similarly, the general recurrence for Q_r is

$$Q_r = Q_{r-1} + Q_{r-2} + T_{r-2} :$$

we again split the set of F-strings of length r into those with leading 0 and those with leading 10; removing these leading bits, we are left in the former set with F-strings of length $r-1$, which contribute Q_{r-1} 1-bits, and in the latter set with T_{r-2} F-strings of length $r-2$, which contribute Q_{r-2} 1-bits, to which the T_{r-2} 1's in the removed prefixes of the form 10 have to be added. By repeatedly applying the resulting recurrence for Q_r, one gets

$$\begin{aligned} Q_r &= Q_{r-1} + Q_{r-2} + F_{r-1} \\ &= 2\,Q_{r-2} + Q_{r-3} + F_{r-1} + F_{r-2} \\ &= 3\,Q_{r-3} + 2\,Q_{r-4} + F_{r-1} + F_{r-2} + 2\,F_{r-3} \\ &= 5\,Q_{r-4} + 3\,Q_{r-5} + F_{r-1} + F_{r-2} + 2\,F_{r-3} + 3\,F_{r-4}. \end{aligned}$$

The last equation can be rewritten as

$$Q_r = F_4\,Q_{r-4} + F_3\,Q_{r-5} + F_0\,F_{r-1} + F_1\,F_{r-2} + F_2\,F_{r-3} + F_3\,F_{r-4},$$

which can be generalized to

$$Q_r = F_\ell\, Q_{r-\ell} + F_{\ell-1}\, Q_{r-\ell-1} + \sum_{j=1}^{\ell} F_{j-1}\, F_{r-j}.$$

Substituting $\ell = r-1$, this yields

$$\begin{aligned} Q_r &= F_{r-1}\, Q_1 + F_{r-2}\, Q_0 + \sum_{j=1}^{r-1} F_{j-1}\, F_{r-j} \\ &= F_{r-1}\, Q_1 + F_{r-2}\, Q_0 + \sum_{j=1}^{r} F_{j-1}\, F_{r-j} - F_{r-1}F_0 \\ &= \frac{1}{5}\left[(r+1)L_{r+1} - F_r\right], \end{aligned}$$

where L_r is the r-th Lucas number, and we have used a well known formula for the convolution of Fibonacci numbers, see, e.g., [5, Formula A3.55].

To evaluate the density of 1-bits in the Fibonacci representation, we divide the number of 1-bits by the total number of bits, both limited to F-strings of length r, to get

$$\frac{Q_r}{r\,T_r} = \frac{\frac{1}{5}\left[(r+1)L_{r+1} - F_r\right]}{r\,F_{r+1}}.$$

We are interested in the limiting value as $r \to \infty$, so one can use $F_r \approx \frac{1}{\sqrt{5}}\phi^{r+1}$ and $L_r \approx \phi^r$ as approximations to Fibonacci and Lucas numbers (see, e.g., [5, Formulas A3.71–72]), where $\phi = \phi_{(2)} = \frac{1+\sqrt{5}}{2}$ is the golden ratio. This gives

$$\begin{aligned} \frac{Q_r}{r\,T_r} &= \frac{1}{5}\left(1+\frac{1}{r}\right)\frac{\phi^{r+1}}{\frac{1}{\sqrt{5}}\phi^{r+2}} - \frac{1}{5r}\frac{1}{\phi} \\ &\longrightarrow \frac{1}{5}\frac{\sqrt{5}}{\phi} = \frac{1}{\sqrt{5}}\left(\frac{2}{1+\sqrt{5}}\right) = \frac{1}{2}\left(1-\frac{1}{\sqrt{5}}\right) = 0.276393 \end{aligned}$$

as r tends to infinity.

2.2.3 Usefulness of the Fibonacci representation for modular exponentiation

The density of 1-bits in the Fibonacci representation with $k = 2$ is thus surprisingly low, only slightly more than a quarter of the bits are 1's, and even taking into account that the representation itself is about 44% longer than for the standard binary system, the expected number of 1-bits for representing an integer N will be only $1.4404 \times 0.2764\ \log_2 N = 0.398\ \log_2 N$, which is more than 20% smaller than for the standard system. The fourth and fifth columns of Table 2 give, for $k \geq 2$, the probability for a given bit-position to hold a 1, and the coefficient of $\log_2 N$ of the expected number of 1-bits in the k-ary Fibonacci representation of the integer N. The probabilities for $k > 2$ have been evaluated numerically.

Recall that the operations needed to evaluate the modular exponentiation $a^m \bmod N$ could be split into two classes: the generation of basis elements, the number of which

k	$\phi_{(k)}$	# bits ($\times \log_2 N$)	prob. of a 1-bit	avg # of 1-bits ($\times \log_2 N$)	total # of mult ($\times \log_2 N$)
2	1.6180	*1.4404*	0.276	**0.398**	1.839
3	1.8393	*1.1375*	0.382	**0.434**	2.709
4	1.9276	*1.0562*	0.434	**0.458**	3.627
5	1.9660	*1.0254*	0.462	**0.474**	4.575
6	1.9836	*1.0120*	0.478	**0.484**	5.544
∞	2	*1*	0.5	**0.5**	1.5

TABLE 2: *Number of multiplications for the k-ary Fibonacci number systems*

equals the number of bits in the representation of N, and the multiplications of a selected subset of these basis elements, the size of this subset being the number of 1-bits in the representation of m. For each basis element, a single operation is needed for $k = 2$, as

$$a^{F_{i+1}} = a^{F_i} \times a^{F_{i-1}},$$

so the element is not obtained by squaring the preceding one, but rather by multiplying the two preceding ones. But similarly to the case of the standard k-ary numeration systems, for $k > 2$, a single operation is not sufficient to generate a basis element, and in fact, $k - 1$ multiplications are needed. The last column of Table 2 gives the total sum for both calculating the basis elements and then multiplying some of them, again in units of $\log_2 N$.

Because of the high price to be paid for evaluating the basis elements for $k > 2$, the total number of required operations is strictly increasing with k, and the minimum is thus reached for $k = 2$ with $1.84 \log_2 N$, which is about 23% more than for the standard binary system based on powers of two. It thus seems that repeated squaring can not be beaten by changing the representation of the integers.

The surprising part is, however, the fact that once the basis elements are given, the average number of such elements to be multiplied is smaller for the Fibonacci representations than for the standard binary system, with a minimum at $k = 2$, as given in the fifth column (in boldface). This corresponds to a scenario in which modular exponentiation $a^m \bmod N$ is repeatedly requested with constant a and N, but for many different exponents m. In that case, the basis elements $a^{F_1}, a^{F_2}, \ldots, a^{F_{\lfloor \log_\phi N \rfloor}}$, are evaluated only once and stored in an array A, and the operations required for calculating $a^m \bmod N$, for each m, are:

1. represent m by the F-string $c_r \cdots c_2 c_1$ in the Fibonacci system with $k = 2$;
2. multiply the elements of A whose indices correspond to the 1-bits in $c_r \cdots c_2 c_1$.

The time needed for generating the basis elements is therefore not accounted for, as it can be amortized over a potentially unlimited number of evaluations of $a^m \bmod N$. The time required for step 1. involves the conversion of a number m from the standard to the Fibonacci representation. This can be done by the above mentioned iterative procedure using $\log_{\phi_{(2)}} m$ subtractions, but since subtractions and multiplications require times of different orders of magnitudes, the dominant part of the time complexity is that of step 2.

Other variants of storing precomputed values have been suggested to accelerate exponentiation [3, 11], and the Fibonacci approach gives another line of time / storage tradeoffs. Figure 1 plots the number of multiplications against the corresponding number of elements that are stored, both in units of $\log_2 N$, for the Fibonacci approach and two k-ary methods using precomputed values. In the *Partial* precomputation method, the elements stored are $a^k, a^{k^2}, \ldots, a^{k^{\log_k N}}$, and each element is raised to a power i, $0 \le i < k$ in $l(i)$ multiplications. The corresponding values appear in Table 1, where the number of elements appears in the third column (in *italics*), and the number of multiplications in the sixth column (in **boldface**); the corresponding points on the plot are labeled P2, P3, etc. The *Full* precomputation method stores, in addition to the above basis elements, also all their required powers, that is, a^{ik^j}, for $1 \le i < k$ and $1 \le j \le \log_k N$. The space is thus $(k-1)\log_k N$, but only one multiplication is needed for each of the $\log_k N$ terms. The values are listed in Table 3 and the corresponding points on the plot are F2, F3, etc.

k	# stored elements	avg # mult
2	*1.000*	**0.500**
3	*1.262*	**0.421**
4	*1.500*	**0.375**
5	*1.723*	**0.345**
6	*1.934*	**0.322**
7	*2.137*	**0.305**
8	*2.333*	**0.292**
9	*2.524*	**0.280**

TABLE 3: *Full storage method*

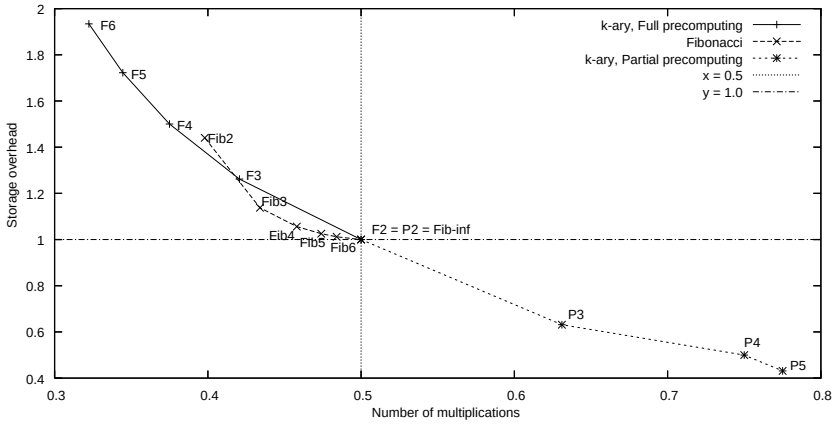

FIGURE 1: *Plot of time / space tradeoffs*

The space and time values for the Fibonacci variants of order k appear in Table 2, in the third (italics) and fifth (boldface) columns, respectively, and the points on the plot are labeled Fib$k$. The curves for both standard k-ary methods have their origin at the same point (0.5,1) and progress in opposite directions with increasing k. The Fibonacci variants yield another kind of tradeoffs, originating for $k = 2$ at the point (0.398,1.44) and asymptotically approaching, for $k \to \infty$, the point (0.5,1) which was the origin for the other methods.

3. Conclusion

While modular exponentiation is usually performed using repeated squaring or its generalizations to standard k-ary methods, this work suggests that non-standard representations, such as those based on Fibonacci numbers, may yield new time / space tradeoffs, which can be advantageous in certain applications.

Examples of possible applications where a and N are fixed, and $a^m \bmod N$ is to be evaluated for many different m, include cryptographic protocols based on the discrete logarithm and El-Gamal [6] encryption and signature schemes.

References

[1] Apostolico A., Fraenkel A.S., Robust transmission of unbounded strings using Fibonacci representations, *IEEE Trans. Inform. Theory* **33** (1987) 238–245.

[2] Brassard G., Bratley P., *Fundamentals of Algorithmics*, Prentice Hall, Englewood Cliffs, NJ (1996).

[3] Brickell E.F., Gordon D.M., McCurley K.S., Wilson D., Fast exponentiation with precomputation, *Proc. Eurocrypt'92*, LNCS **658**, Springer Verlag (1992) 200–207.

[4] Cormen T.H., Leiserson C.E., Rivest R.L., *Introduction to Algorithms*, MIT Press, Cambridge (1990).

[5] Dunlap R.A., *The Golden Ratio and Fibonacci Numbers,* World Scientific Publishing, Singapore (1997).

[6] El Gamal T., A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. on Inf. Th.* **IT–31** (1985) 469–472.

[7] Fraenkel A.S., Klein S.T., Robust Universal Complete Codes for Transmission and Compression, *Discrete Applied Mathematics* **64** (1996) 31–55.

[8] Goodrich M.T., Tamassia R., *Algorithm Design: Foundations, Analysis, and Internet Examples*, John Wiley & Sons, Inc., New York, NY (2002).

[9] Gordon D.M., A survey of fast exponentiation methods, *Journal of Algorithms* **27**(1) (1998) 129–146.

[10] Knuth D.E., *The Art of Computer Programming, Vol.* **II**, *Semi-Numerical Algorithms,* Addison-Wesley, Reading, MA (1973).

[11] Lim C.H., Lee P.J., More flexible exponentiation with precomputation, *Proc. Crypto' 94*, LNCS **839**, Springer Verlag (1994) 95–107.