Fast Exponentation in Cryptography

Irina. E. Bocharova and Boris. D. Kudryashov

St.-Petersburg Academy of Airspace Instrumentation,
Bolshaia Morskaia str.,67, St.-Petersburg,190000, Russia,
e-mail:liap@sovam.com

Abstract. We consider the problem of minimizing the number of multiplications in computing $f(x) = x^n$, where n is an integer and x is an element of any ring. We present new methods which reduce the average number of multiplications comparing with well-known methods, such as the binary method and the q-ary method. We do not compare our approach with algorithms based on addition chains since our approach is intended for cryptosystems with large exponent n and the complexity of constructing the optimal addition chain for such n is too high. We consider the binary representation for the number n and simplify exponentiation by applying ideas close to ideas exploited in data compression. Asymptotical efficiency of the new algorithms is estimated and numerical results are given.

1 Introduction

Exponentiation is one of the main operations that is performed in public key cryptosystems [1], [2]. For example, to encrypt message m in RSA system [2] one should compute $m^e \pmod N$, where pair $\{e, N\}$ defines a public key. The cryptographic key distribution system by Diffie and Hellman [1] is based on calculating y^x, where y denotes a published number associated with one user and x is the secret number of another user.

The total complexity of exponentiation $\kappa(n)$ may be written as follows

$$\kappa(n) = S(n) + M(n),$$

where $S(n)$ denotes the number of squarings and $M(n)$ denotes the number of multiplications. Squarings have less computational complexity than integer multiplications especially if these operations are performed over $GF(2^m)$. Moreover all methods below have approximately the same number of squarings $S(n) = \ell(n)$,where $\ell(n) = \lfloor \log n \rfloor + 1$ denotes the length of the binary representation for n, $\lfloor a \rfloor$ is the largest integer less or equal to a, $\log a$ means $\log_2 a$. So, we discuss methods for speeding up exponentiation in terms of minimizing the number of multiplications.

A detailed review devoted to the methods of fast exponentiation is given in [3]. Since we are interested in computing x^n for cryptosystems, we consider rather large pseudorandom values n. Let $M_{max}(\ell)$ and $M_{av}(\ell)$ be the maximal and the average number of multiplications over all n with binary representation

of length $\ell(n) = \ell$, respectively. For the most commonly used binary method $M(n)$ is proportional to $\ell(n)$. For more efficient methods the number of multiplications can be reduced to a value proportional to $\ell(n)/\log \ell(n)$ by keeping some intermediate results. Let s be the maximal number of intermediate results kept. For algorithms described below the number of multiplications $M(n)$ depends on n and s in accordance with the following formula

$$M(n) = \frac{\alpha_1 \ell(n)}{\log s + \alpha_2} + \alpha_3 s + \alpha_4, \tag{1}$$

where $\alpha_i, i = 1, \ldots, 4$ are parameters depending on the computation algorithm. An optimal value s may be found by minimizing the right side of (1). For large n the result takes the form

$$M(n) = \frac{\alpha \ell(n)}{\log \ell(n)}(1 + o(n)), \tag{2}$$

where $o(n) \to 0$ when $n \to \infty$.

So, the asymptotic performance of the algorithms may be characterized by coefficient α in (2). For finite lengths we will compute the average number of multiplications as a function of $\ell(n) = \ell$ for the optimal memory size s.

Since reducing the number of multiplications is obtained by keeping some intermediate results it is necessary to compare algorithms of exponentiation provided that the memory requirements are the same. It is clear from the description of the algorithms presented below that the required number of precomputations (memory size) does not exceed $M(n)/2$. Increasing s does not immediately leads to increasing the efficiency of exponentiating. For each algorithm there exists an optimal number of precomputations. Asymptotically both s and $M(n)$ are of the same order and thus we shall not calculate s separately. For finite n the required memory size may be the important parameter of the algorithm, especially for the hardware implementation. So, in subsection 3.4 containing some examples we give both $M(n)$ and s values for the known and for the new algorithms. We shall show that the new algorithms provide better time-space tradeoff than known ones.

2 Conventional Algorithms for Fast Exponentiation

Let $\mathbf{n} = (n_1, n_2, \ldots, n_{\ell(n)})$ be the binary representation of n.

2.1 Binary Method

To compute x^n we use the following formula

$$x^n = (\ldots((x^{n_1})^2 x^{n_2})^2 x^{n_3} \ldots)^2 x^{n_{\ell(n)}}.$$

This method requires $M(n) = w(n)$ multiplications, where $w(n)$ denotes the Hamming weight of the sequence $(n_1, \ldots, n_{\ell(n)})$. If $\mathbf{n}$ is a random sequence of binary independent identically distributed variables with $Pr(n_i = 1) = p$ then

$$M_{av}(\ell) = p\ell.$$

For the binary method it is necessary to keep only x and a current intermediate result.

2.2 q-ary Method

Let $\mathbf{d} = (d_1, \ldots, d_t)$, $0 \le d_j < q$, $0 \le j < t$, be the q-ary representation of n, i.e. $n = d_0 q^t + d_1 q^{t-1} + \ldots + d_t$. For calculating x^n it is necessary to compute $x^2, x^3, \ldots, x^{q-1}$ and then to use the formula

$$x^n = (\ldots(x^{d_0 q} x^{d_1})^q x^{d_2})^q x^{d_3})^q \ldots x^{d_t}).$$

Let $q = 2^k$, where k is a positive integer. Then the number of multiplications is equal to

$$M(n) = \ell(n)/k + 2^{k-1} - k. \tag{3}$$

This method requires $s = 2^k - 2$ memory cells for keeping x^{d_j}, $1 < j < q$. For large n the right side of (3) is minimal if $k = \log \ell(n) - 2 \log\log \ell(n)$. Substituting the optimal value of k into (3) we obtain

$$M(n) = \frac{\ell(n)}{\log \ell(n)}(1 + o(n)). \tag{4}$$

Note that for large n values $M_{max}(\ell(n))$ and $M_{av}(\ell(n))$ asymptotically coincide and both satisfy (4).

2.3 Method of Factorization

Suppose it is possible to represent n in the form $n = ml$, where m is the least prime divisor of n and $l > 1$ or in the form $n = ml + 1$ otherwise. To compute x^n we first obtain $X = x^m$ and then compute $x = X^l$ or $x = X^l x$.

The method of factorization uses this rule recurrently. On average the method of factorization is better than the binary method but factorization of n has approximately the same complexity as the problem of breaking RSA cryptosystem, using n as a public key. Hence this method cannot be recommended for large n.

2.4 Addition Chains

This method implies that n is represented in the form $n = p_1 + p_2 + \ldots + p_l$, where terms $p_1, p_2, \ldots, p_l$ are found by using a special tree (addition chain). It is evident that x^n can be computed by the following formula

$$x^n = x^{p_1} x^{p_2} \ldots x^{p_l}.$$

However this method is unacceptable for cryptography since the complexity of constructing the addition chains exponentially grows with $\ell(n)$.

3 Exponentiation Based on Data Compression Algorithms

First we explain the idea of this approach. It follows from the above brief review that methods of fast exponentiation require precomputations and the storage of some intermediate results. We can improve the space-time tradeoff of exponentiating by precomputing x^d, for such sequences $\mathbf{d}_i$ which often appear in the sequence $\mathbf{n}$. Indeed, the algorithms presented below represent the different ways for constructing a proper set $D = \{\mathbf{d}\}$.

Some data compression algorithms exploit the same idea. They extract the most probable subsequences from the sequence which should be compressed and encode them by fewer bits than others.

3.1 LZ-Approach for Computing x^n

The application of the Lempel-Ziv data compression algorithm [4] to exponentiation has been considered in [5] and [6].

Remind LZ-algorithm of parsing string of length $\ell(n)$ into different subblocks.

- *A sequence* $\mathbf{n}$ *is read beginning at the left and* $\mathbf{d}_k = (n_{i(k-1)+1}, \ldots, n_{i(k)})$ *is selected as the* $k-th$ *block if the block* $\mathbf{d} = (n_{i(k-1)+1}, \ldots, n_{i(k)-1})$ *was already seen but* $(n_{i(k-1)+1}, \ldots, n_{i(k)})$ *was not seen yet.*

Let $\mathbf{n} = (\mathbf{d}_1, \ldots, \mathbf{d}_{c(n)})$ be a parsing of $\mathbf{n}$ by LZ-algorithm. Then $c(n)$ is called LZ-complexity of n [4]. Let d_i be numbers with the binary representations $\mathbf{d}_i$, $i = 1, \ldots, c(n)$.

For computing x^{d_k} we use the following formula

$$x^{d_k} = \begin{cases} x, & \text{if } i(k) = 1\ ; \\ (x^d)^2, & \text{if } n_{i(k)} = 0\ ; \\ (x^d)^2 x, & \text{if } n_{i(k)} = 1\ , \end{cases}$$

where d coincides with one of the previous blocks and hence x^d was already computed.

We exponentiate by the formula

$$x^n = (\ldots((x^{d_1})^{2^{\ell(d_2)}} x^{d_2})^{2^{\ell(d_3)}} \ldots)^{2^{\ell(d_c)}} x^{d_c}).$$

Note that the original LZ-algorithm keeps every new subsequence and puts it into the set D even if this subsequence contains only zeros. However zero in the binary representation of n corresponds only to squaring and does not require multiplication.

So we consider the algorithm similar to an LZ-algorithm but having some differences. This algorithm parses $\mathbf{n}$ into strings which begin with 1. Strings containing only zeros are skipped and are not put into the dictionary.

Using the modified algorithm we find the following representation for n:

$$\mathbf{n} = (\mathbf{d}_1, 0^{z_1}, \mathbf{d}_2, 0^{z_2}, \ldots, \mathbf{d}_{C(n)}, 0^{z_{C(n)}}), \tag{5}$$

where 0^z means a sequence of z zeros, $C(n)$ is the number of the found subsequences $\mathbf{d}$.

The algorithm of exponentiating may be described as follows:

- *Let $(n_1, n_2, \ldots, n_t)$ be the processed part of the sequence $\mathbf{n}$, and f be accumulated intermediate result. If $n_{t+1} = 0$ then we put $f = f^2, t = t + 1$ and go to the next step. If $n_{t+1} = 1$ then we form a new sequence $\mathbf{d}$.*

For computing x^n now we use the formula

$$x^n = (\ldots((x^{d_1})^{2^{s_1+\ell(d_2)}} x^{d_2})^{2^{s_2+\ell(d_3)}} \ldots)^{2^{s_C-1+\ell(d_C))}} x^{d_C})^{2^{s_C}}, \tag{6}$$

It is evident that the number of multiplications is equal to

$$M(n) = C(n) + C_1(n)$$

Were $C_1(n)$ is the number of substrings computed using multiplication (strings with last symbol equal to 1). For keeping intermediate results $s = C(n)$ memory cells are required.

Let $\mathbf{n}$ be a random sequence of binary independent identically distributed variables with $\mathbf{Pr}(n_i = 1) = p$. Using a well-known technique (see for example [7]) the following expressions for the maximal and the average complexity for large n may be easy derived:

$$M_{max}(\ell) = \frac{2\ell}{\log \ell}(1 + o(n)), M_{av}(\ell) = \frac{(1+p)\mathcal{H}(p)\ell}{\log \ell}(1 + o(n)), \tag{7}$$

where $\mathcal{H}(p) = -p \log p - (1-p)\log(1-p)$ is the binary entropy function. Note that this method is asymptotically worse than the q-ary method, but it is better on average for sequences with small entropy or small ZL-complexity.

3.2 Exponentiating Based on the Typical Sets

Let k, m be integers, $m = \ell(n)/k$. We split $\mathbf{n}$ into m subblocks of length k, $\mathbf{n} = (\mathbf{n}_1, \ldots, \mathbf{n}_m)$. Let denote $D = \{\mathbf{d}\} \subset \{0,1\}^k$ some set of the binary sequences of length k.

For any D consider the following algorithm of exponentiating.

1. *We precompute x^d for all $\mathbf{d} \in D$ (here d means a number with the binary representation $\mathbf{d}$), and put $f = 1$;*
2. *For $i = 1$ to m do*
   begin
   - *if $\mathbf{d} = \mathbf{n}_i \in D$ then $y =$ precomputed x^d*
     else we compute $y = x^d$ by the binary method.
   - *Put $f = fy$*

   end

The performance of exponentiating algorithms depends on the proper choice of D. Suppose that $\mathbf{n}$ is a random sequence of independent identically distributed binary variables with probability $\mathbf{Pr}(n_i = 1) = p$. Let D be the typical set [7], i.e.

$$|D| = 2^{k(\mathcal{H}(p)+\delta(k))},$$

$$\sum_{\mathbf{d}\in D} \mathbf{Pr}(\mathbf{d}) = 1 - \epsilon(k).$$

where $\epsilon(k), \delta(k) \to 0$ if $k \to \infty$.

For large $\ell(n)$ and k we have

$$M_{av}(\ell(n)) \le \ell(n)/k + \ell(n)\epsilon(k) + k|D|. \tag{8}$$

Minimizing right side of (8) over k after simple transformations we get

$$M_{av}(\ell(n)) \le \frac{l\mathcal{H}(p)}{\log \ell}(1 + o(n)) \tag{9}$$

Comparing (4), (7) and (9) it is easy to see that the proposed method is better than the method based on the LZ-algorithm and better than the q-ary method if the entropy of $\mathbf{n}$ is less than 1.

It is not surprising that this method is advantageous compared to the LZ algorithm, since the LZ data compression algorithm is not optimal for sources with known probability distribution. On the other hand, the method based on the typical sets cannot be recommended for practical use, since the above estimates are valid only for very large $\ell(n)$.

In the next subsection we construct an algorithm with the same asymptotic behavior having good performances for finite lengths $\ell(n)$.

3.3 Exponentiation by Using Variable-to-fixed Length Coding

It is intuitively clear that the problem of minimizing the number of multiplications reduces to the problem of constructing a proper set D. In this subsection we consider the algorithm based on parsing of $\mathbf{n}$ using variable-to-fixed length codes (VF-codes) for source coding [8]. This algorithm may be regarded as the 2^k-ary method with variable k, depending on the specific sequence.

Suppose that the set D contains binary sequences of different lengths and the first symbol of any $\mathbf{d} \in \mathbf{D}$ is 1.

The algorithm uses the representation of $\mathbf{n}$ in the form (5) where $\mathbf{d}_1$ denotes a binary vector from D of maximal length beginning with the first 1 in the sequence $\mathbf{n}$, $\mathbf{d}_2$ denotes a binary vector from D with maximal length beginning from the next 1 after $\mathbf{d}_1$, etc.

Assume that all x^d, $\mathbf{d} \in D$ are precomputed. Then to calculate x^n it is necessary to perform the following sequence of operations:

1. *Read the next symbol of* $\mathbf{n}$.
2. *When the next symbol is equal to zero square the intermediate result and return to the step 1. Otherwise read sequence* $\mathbf{n}$ *symbol by symbol and square an intermediate result while a read sequence coincides with some* $\mathbf{d}_i$ *from* D. *Multiply the obtained result by* x^{d_i} *and return to step 1.*

In other words, computations are performed by formula (6).

Example 1. Let $D = \{1, 11, 111, 101\}$ then for $n = 669, (\mathbf{n} = (1010011101))$ we obtain $x^{669} = ((x^5)^{2^5}x^7)^{2^2}x$. □

Let $\mathbf{n}$ be a random sequence of independent binary variables and p be the probability of 1. Without loss of the generality we assume that $p \leq 1/2$. Let construct for this random sequence the optimal Tunstall VF-code [8] and denote the obtained set of codewords by A. We obtain D from A by adding one symbol 1 to the beginning of each sequence. It follows from [8] that there exists a set $A = \{\mathbf{a}\}$ for which the following inequality holds:

$$\mathbf{E}[\ell(a)]) \geq \frac{\log|A| + \log p}{\mathcal{H}(p)}.$$

Therefore,

$$\mathbf{E}[\ell(d)] \geq \frac{\log|D| + \log p}{\mathcal{H}(p)} + 1. \tag{10}$$

The average length of all-zero series is equal to

$$\mathbf{E}[z] = (1-p)/p. \tag{11}$$

For any parsing we have

$$\ell(n) \geq \sum_{i=1}^{C-1}(z_i + \ell(d_i)).$$

Let $\mathcal{P}(C)$ be the probability distribution for the random variable C. Averaging the last inequality over all $\mathbf{n}$ of the fixed length $\ell(n)$ we obtain

$$\ell(n) \geq \sum_C \mathcal{P}(C)\mathbf{E}\left[\sum_{i=1}^{C-1}(z_i + \ell(d_i))\middle| C\right] = \sum_C \mathcal{P}(C)\sum_{i=1}^{C-1}(\mathbf{E}[z_i|C] + \mathbf{E}[\ell(d_i)|C]).$$

For $i \leq C-1$ values $E[z_i|C]$ and $E[d_i|C]$ do not depend on i. Therefore

$$\ell(n) \geq (\mathbf{E}[C] - 1)(\mathbf{E}[z] + \mathbf{E}[\ell(d)])$$

and

$$\mathbf{E}[C] \leq \frac{\ell(n)}{\mathbf{E}[z] + \mathbf{E}[\ell(d)]} + 1. \tag{12}$$

To evaluate the complexity of precomputations, let consider the binary Tunstall code. Using a tree structure of the code we can compute all $x^{\mathbf{d}}, \mathbf{d} \in D$ spending one multiplication per subsequence.

The total average computational complexity is determined as follows

$$M_{av}(n) = \mathbf{E}[C] + |D|.$$

Using (11),(10) and (12) we obtain

$$M_{av}(\ell(n)) = 1 + \frac{\ell(n)}{\frac{1}{p} + \frac{\log|D| + \log p}{\mathcal{H}(p)}} + |D|. \tag{13}$$

Minimizing right side over $|D|$ we get the result asymptotically coinciding with (9) when $\ell(n) \to \infty$.

3.4 Examples. Numerical Results.

To analyze the above algorithm for the finite lengths $\ell(n)$ we introduce a finite state machine. This machine describes random walks of the algorithm along a binary tree used for parsing $\mathbf{n}$ into subsequences $\mathbf{d} \in D$.

The ordered graph of this machine contains states corresponding to the set of distinct sequences from D, zero state and states corresponding to intermediate nodes of the binary tree. We denote this extended set by D^*.

Example 1 (continuation). If $D = \{1, 11, 111, 101\}$ then machine states belong to $D^* = \{0, 1, 10, 11, 111, 101\}$. We have included the state 10 to provide the path from state 1 to state 101. □

Every directed branch connecting two adjacent states is labeled by Z^m, where $m \in \{0, 1\}$, Z is the formal variable, m denotes the number of multiplications corresponding to the given transition. Each symbol of the sequence $\mathbf{n}$ corresponds to one transition in the state diagram. The initial state is zero state.

Let $\mathcal{P} = \{p_{ij}\}, i, j = 1, \ldots, |D^*|$ be the transition probability matrix for the finite-state machine and $\mathbf{p}$ be the stationary state distribution. Introduce the matrix $G(Z) = \{p_{ij}Z^{m_{ij}}\}, i, j = 1, \ldots, |D^*|$, where m_{ij} denotes the number of multiplications corresponding to the transition from the state i to the state j. Then the average number of multiplications per symbol of sequence $\mathbf{n}$ may be calculated as

$$M_0 = \mathbf{p}G'(Z)\mathbf{1}|_{Z=1},$$

where G' is the matrix of derivatives for elements $G(Z)$ and $\mathbf{1}$ denotes the vector of $|D^*|$ ones. The total average number of multiplications is equal to

$$M_{av}(\ell(n)) = \ell(n)\mathbf{p}G'(Z)\mathbf{1}|_{Z=1} + |D|, \tag{14}$$

Example 1 (continuation). If $D^* = \{0, 1, 10, 11, 111, 101\}$ then

$$\mathcal{P} = \begin{pmatrix} 1-p & p & 0 & 0 & 0 & 0 \\ 0 & 0 & 1-p & p & 0 & 0 \\ 1-p & 0 & 0 & 0 & p & 0 \\ 1-p & p & 0 & 0 & 0 & 0 \\ 1-p & p & 0 & 0 & 0 & 0 \end{pmatrix}, \tag{15}$$

$$G(z) = \begin{pmatrix} 1-p & p & 0 & 0 & 0 & 0 \\ 0 & 0 & 1-p & p & 0 & 0 \\ (1-p)z & 0 & 0 & 0 & p & 0 \\ (1-p)z & 0 & 0 & 0 & 0 & p \\ (1-p)z & pz & 0 & 0 & 0 & 0 \\ (1-p)z & pz & 0 & 0 & 0 & 0 \end{pmatrix} \tag{16}$$

$$\mathbf{p} = \left(\frac{1-p^2}{1+2p}, \frac{p}{1+2p}, \frac{(1-p)p}{1+2p}, \frac{p^2}{1+2p}, \frac{(1-p)p^2}{1+2p}, \frac{p^3}{1+2p}\right),$$

After simple transformations we obtain

$$M_{av}(\ell(n)) = \frac{\ell(n)p}{1+2p} + 3.$$

It follows from the obtained result that the average number of multiplications is $(1+2p)$ times less as the number of multiplications for the binary method. □

Example 2. In this example we propose a set D that is proper in the case of $p = 1/2$, but we estimate the average number of multiplications for any p. Let

$$D = \{1, 3, \ldots, 2^{h-1} - 1, 2^{h-1} + 1, 2^{h-1} + 3, \ldots, 2^h - 1\}, s = |D| = 2^{h-1}.$$

Then

$$D^* = \{0, 1, 2, \ldots, 2^{h-1} - 1, 2^{h-1} + 1, 2^{h-1} + 3, \ldots, 2^h - 1\}.$$

Note that the sets D and D^* from Example 1 represent the particular case of these sets when $h = 3$. Matrix $G(z)$ has the form

$$G(z) = \left(\begin{array}{cccccccc|cccc} 1-p & p & 0 & 0 & \ldots & 0 & 0 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 1-p & p & \ldots & 0 & 0 & 0 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & 0 & \ldots & 1-p & p & 0 & 0 & \ldots & 0 \\ \hline (1-p)z & 0 & 0 & 0 & \ldots & 0 & 0 & pz & 0 & \ldots & 0 \\ (1-p)z & 0 & 0 & 0 & \ldots & 0 & 0 & 0 & pz & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ (1-p)z & 0 & 0 & 0 & \ldots & 0 & 0 & 0 & 0 & \ldots & pz \\ \hline (1-p)z & pz & 0 & 0 & \ldots & 0 & 0 & 0 & 0 & \ldots & 0 \\ (1-p)z & pz & 0 & 0 & \ldots & 0 & 0 & 0 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ (1-p)z & pz & 0 & 0 & \ldots & 0 & 0 & 0 & 0 & \ldots & 0 \end{array}\right). \tag{17}$$

Each band of this matrix contains 2^{h-2} rows.

The average number of multiplications computed by (14) is determined as follows

$$M_{av} = \min_h \frac{\ell(n)}{h - 1 + 1/p} + 2^{h-1}.$$

□

Example 3. Here we consider the special set D that is proper for rather small p. Let $D = \{1, 11, 101, 1001, \ldots, 10^{s-2}1\}$. Using (14), we obtain

$$M_{av}(\ell(n)) = \min_s \frac{p\ell(n)}{2 - (1-p)^{s-1}} + s.$$

For large s this method is approximately twice better than the binary method. It can be proved that this set D is optimal if s and p satisfy the inequality

$$s \leq \log p / \log(1-p) + 1.$$

□

Numerical results for $\ell(n) = 512$ are given in the Table.

Note that the performance of the q-ary method does not depend on p and for $p = 1/2$ it is superior (in the sense of minimizing the number of multiplications) to the binary method that does not require additional memory for storing data. When p tends to zero the binary method becomes better than the q-ary method.

The proposed method based on VF coding has better performances than the q-ary method for $p = 1/2$ and requires less multiplications than the binary method when p decreases. The memory requirements are significantly less than for the q-ary method.

Table. Numerical results

Method	Probability p	Average number of multiplications	Memory s
2^k-ary		111	62
Binary	1/2	256	0
LZ-algorithm (simulation results)	1/2	128	40
Example 2	1/2	102	16
Binary	1/4	128	0
Example 2	1/4	80	16
Example 3	1/4	77	8
Binary	1/8	64	
Example 2	1/8	55	8
Example 3	1/8	48	8

3.5 Application to Cryptology

Consider the above algorithms from the point of view of exploiting them in cryptography. For some applications, such as the decrypting procedure in RSA system or the key generating procedure by Diffie-Hellman, n is not changed for a long time. In RSA encrypting n is variable because it represents the public key of a subscriber with whom it is necessary to communicate.

At first sight it seems very important to know whether or not n varies every time we exponentiate.

Note that for the above algorithms the total computational expenditures consist of the number of operations that one needs to analyze n and the number of multiplications $M(n)$. If n is invariable it is analyzed only once when it is used for the first time or this analysis may be performed in advance and then its results are taken into account in developing the corresponding software or hardware.

However the complexity of the preliminary analysis is proportional to the length of the sequence $\mathbf{n}$. At the same time to multiply just two numbers with binary representation of length $\ell = \ell(n)$ a number of calculations at least proportional to $\ell \log \ell$ is required.

Thus, both asymptotically and for finite lengths the computational expenditures for analysis of n are negligible compared to the complexity of multiplying and cases of constant and variable n do not differ in principle. Hence, the above algorithms may be exploited in all the enumerated applications.

3.6 Conclusion

We have considered the problem of computing x^n in connection with cryptography problems. Our main goal has been to minimize the average number of multiplications. The comparisons of the obtained results with the binary and the q-ary methods have been made because only these methods are acceptable for large n. The problem of exponentiating may be solved by the well-known methods of source coding. However it have been clarified that the method of exponentiation based on the LZ-algorithm presented in [5] provides poor asymptotic performance and can be used only for the sequences with small LZ-complexity.

To improve the asymptotical efficiency of exponentiating we proposed to use the concept of typical sets. It makes it possible to obtain average number of multiplications $H(p)$ times less as for the q-ary method but for extremely large n. Asymptotically efficient constructible algorithm have been got using so-called VF-codes [8]. Some examples of specific codes and formulae for average number of multiplications are given. Presented numerical results show the significant benefit of the new method compared to the q-ary method.

References

[1] W. Diffie and M.E. Hellman, *New directions in cryptography.* IEEE Trans. Inform. Theory, vol. IT-22, pp.644-654, Nov.,1976.

[2] R.L. Rivest, A. Shamir and L. Adelman, *A method of obtaining digital signatures and public-key cryptosystems*, Commun. ACM, vol.21,pp.120-126, Feb.,1978

[3] D.E. Knuth , *Seminumerical algorithms* The Art of Computer Programming, vol.2, Addison-Wesley, Reading , Mass., 1969.

[4] J.Ziv and A.Lempel, *Compression of individual sequences via variable rate coding.* IEEE Trans. Inform. Theory, V.24, No 5, Sep., 1978.

[5] Y.Yacobi, *Exponentiating faster with addition chains*, Proceedings of Eurocrypt'90.

[6] I.Bocharova and B.Kudryashov, *Fast exponentiation based on Lempel-Ziv algorithm*, In Proceedings on the 6th joint Swedish-Russian International Workshop on Information Theory, August, 1992, pp.259-263

[7] T.M.Cover and J.A.Thomas, *Elements of information theory.* New York: Wiley, 1991.

[8] F. Jelinek and K.S.Schneider, *On variable-length-to-block coding*, The structural and distance properties of punctured convolutional codes, IEEE Trans. Inform. Theory, V.18, No 6, Nov., 1982.