Please cite this article as:
Artur Jakubski, Robert Perliński, Review of general exponentiation algorithms, Scientific Research of the Institute of Mathematics and Computer Science, 2011, Volume 10, Issue 2, pages 87-98.
The website: http://www.amcm.pcz.pl/

REVIEW OF GENERAL EXPONENTIATION ALGORITHMS

Artur Jakubski, Robert Perliński

Institute of Computer and Information Sciences
Czestochowa University of Technology, Poland
artur.jakubski@icis.pcz.pl, robert.perlinski@icis.pcz.pl

Abstract. Arithmetic on large integers is often necessary in cryptography. Many cryptosystems depend on the possibility of computing powers g^e (exponentiation). In this paper we describe well-known exponentiation algorithms with their complexity analysis. Our algorithm introduces a new idea of solving the exponentiation problem. We show cases in which the complexity of our algorithm is better than the complexity of any other algorithms.

Introduction

The problem of efficient exponentiation is extremely important for the development of modern cryptography. Many cryptographic systems use this operation in the information encryption process (eg. RSA or ElGamal).

This work concerns the problem of effectively calculating g^e, where g is an element of group G, and e is a positive integer number [1+2]. Generally, this problem is considered when g is a real number. In this paper we analyse general exponentiation algorithms and compare their effectiveness. As the criterion of the effectiveness of the algorithm we adopted the number of group operations which the algorithm performs. When this number is smaller for a set of values adopted by e, such an algorithm is considered to be more efficient. We are aware of the fact that analysis of the number of algorithm bit operations would be more conclusive. Aware of this fact, we abandoned it to simplify the analysis and thus to make the paper clearer.

In this paper we present our own contribution, a new general exponentiation algorithm. We called it the algorithm of zero-one sequences and it is contained in Section 2 of this work. We compare its effectiveness to other algorithms presented in the work.

1. Review of general exponentiation algorithms

The problem of exponentiation can be considered due to the changing values of g or e. We can consider the problem for fixed base g and changing exponent e or

inversely, for fixed *e* and arbitrarily chosen *g*. The third option is when both the values are subject to change.

There are therefore three types of exponentiation algorithms:

- general exponentiation algorithms,
  exponentiation for arbitrarily chosen *g* and arbitrarily chosen *e*, so-called general purpose exponentiation algorithms:
  - simple binary and *k*-ary exponentiation
  - sliding-window exponentiation
- fixed-exponent exponentiation
- fixed-base exponentiation.

In this paper we consider the general exponentiation problem, where neither *g* nor *e* are predetermined. In our study, we will analyse the values of *e* in a certain range in order to compare the effectiveness of the algorithms.

1.1. Exponentiation - naive method

It seems that the easiest way to calculate g^e is to perform $e-1$ operations in group *G*. Below is an example of an algorithm pseudocode called the naive method.

Algorithm - naïve method

Input: $g \in G$, $e \in N_+$

Output: g^e

1. $A \leftarrow g$
2. For *i* from 1 to $e-1$ do the following:
   2.1. $A \leftarrow A \cdot g$
3. Return A

This simple method of calculating power is often ineffective, especially when *e* is of a high value. Moreover the complexity of this algorithm requires calculating O(e) multiplications. For practical applications, where *e* is of several hundred, a thousand, or more bits, the application of this method is impractical.

1.2. Left-to-right algorithm

The number of operations in the algorithm of the previous section can be significantly reduced. In this and the next section we will present the algorithms described in literature as the binary exponentiation algorithm, iterated algorithm for raising to the square or the algorithm of fast exponentiation [1+3].

In this algorithm, the value of *A* (partial results of exponentiation) is raised to the square in each course of loop 2, ie. *t* + 1 times. In each execution of loop 2 the algorithm checks the next bit value, if this value is equal to 1, we assign multiplying *A* and *g* to result *A*.

Algorithm of left-to-right binary exponentiation

Input: $g \in G$, a positive integer $e = (e_t e_{t-1} \cdots e_1 e_0)_2$

Output: g^e

1. $A \leftarrow 1$
2. For *i* from *t* down to 0 do the following:
   2.1. $A \leftarrow A \cdot A$
   2.2. If $e_i = 1$ than $A \leftarrow A \cdot g$
3. Return A

Due to the fact that the most significant bit of exponent *e* is equal to one, we can omit the first execution of loop 2 and thus assign g to variable *A*. We take into account this fact when analysing the complexity of this algorithm. In more detailed analysis, the number of multiplications (operations in the group), of this algorithm is $\lfloor \log e \rfloor + v(e) - 1$, where $v(e)$ is the number of ones in the binary representation of *e* [4]. For *e* = 15, this method requires six multiplications, while g^{15} can be determined using only five multiplications. The algorithm does not work optimally, particularly for binary exponents that contain long sequences of ones. In conclusion, the asymptotic complexity of the left-to-right algorithm is O(log e).

1.2.1. Example

Below we present the left-to-right algorithm. Table 1 describes the obtained values, taken by *A* in succeeding loop 2 courses, for exponent *e* = 283.

Table 1

Operating of left-to-right algorithm for *e* = 283

i	e_i	$e_{(2)}$	**A**
8	1	1	g
7	0	10	g^2
6	0	100	g^4
5	0	1000	g^8
4	1	10001	g^{17}
3	1	100011	g^{35}
2	0	1000110	g^{70}
1	1	10001101	g^{141}
0	1	100011011	g^{283}

1.3. Right-to-left algorithm

This algorithm, though it differs from the previous one, works very similarly [1, 3]. The main difference lies in the use of binary exponent *e*. The exponent bits equal to 1 in this case are checked (in step 2.1 of the algorithm) from the least significant bit to the most significant one, so, from right to left. Loop 2 executes as long as $e \neq 0$, that is, de facto, as long as we have bits of the exponent. In each execution of loop 2, *S* is raised to the square. Thus, the set of values adopted by *S* is $\{ g^1, g^2, g^4, \ldots, g^{2^{\lfloor \log e \rfloor}} \}$. Result *A* is obtained by multiplying these powers *g* which correspond to value 1 in the binary form of exponent *e* (from the least significant bit). Just as in the left-to-right algorithm, the asymptotic complexity of this algorithm is O(log e).

Algorithm of right-to-left binary exponentiation

Input: $g \in G$, a integer $e \in N_+$

Output: g^e

1. $A \leftarrow 1$, $S \leftarrow g$
2. While $e \neq 0$ do the following:
   2.1. If e is odd then $A \leftarrow A \cdot S$
   2.2. $e \leftarrow \lfloor e/2 \rfloor$
   2.3. If $e \neq 0$ then $S \leftarrow S \cdot S$
3. Return A

1.3.1. Example

The example of the right-to-left algorithm is shown in Table 2. Variable *A* stores the partial results of the powers computed in succeeding courses of loop 2. Column 3 shows the binary exponent fixed at a given stage of the algorithm.

Table 2

The right-to-left algorithm for *e* = 283

$e_{(10)}$	$e_{(2)}$	computed exponent	*A*	*S*
283	100011011		1	g
141	10001101	**1**	g	g^2
70	1000110	**1**1	g^3	g^4
35	100011	**0**11	g^3	g^8
17	10001	**1**011	g^{11}	g^{16}
8	1000	**1**1011	g^{27}	g^{32}
4	100	**0**11011	g^{27}	g^{64}
2	10	**0**011011	g^{27}	g^{128}
1	1	**0**0011011	g^{27}	g^{256}
0		**1**00011011	g^{283}	-

1.4. Montgomery Ladder

The Montgomery Ladder [5] is an interesting modification of the left-to-right algorithm. Just as in that algorithm, the Montgomery Ladder uses the binary representation of exponent *e* as well. The working of the Montgomery Ladder and the left-to-right algorithm, is related to reading the subsequent bits of the exponent, from the most to the least significant bit. The difference in the Montgomery Ladder is that it begins reading from the second bit. In the Montgomery Ladder, we calculate two possible partial results in one step, which are: raising to the square and raising to the square with multiplying by *g*. Depending on the value of the next exponent bit, we choose the proper partial result.

In this algorithm, there are two auxiliary variables - x_1 and x_2. The first one takes successively (during the execution of the loop 2) the value of $g^{(e_t)_2}$, $g^{(e_t e_{t-1})_2}$, etc., and finally reaches the desired value of $g^{(e_t e_{t-1}\cdots e_1 e_0)_2}$. The value of x_2 in subsequent loop 2 executions has a value of $g \cdot x_1$. Variables x_1 and x_2 are something like the complementary rungs of a ladder. When x_1 is of value $g^{(e_t e_{t-1}\cdots e_m)_2}$ and the next bit of the exponent (the next after *m*) is 1, then x_1 takes the value of $x_1 \cdot x_2$, so, $x_1 \cdot g \cdot x_1$. This means that in this case, the value of x_1 is $g^{(e_t e_{t-1}\cdots e_m 1)_2}$. When x_1 is of value $g^{(e_t e_{t-1}\cdots e_m)_2}$ and the next bit of the exponent is 0, then we assign $x_1 \cdot x_1$ to x_1, thus, we have computed $g^{(e_t e_{t-1}\cdots e_m 0)_2}$.

There are two multiplications performed in each execution of loop 2, and the loop runs $\lfloor \log_2 e \rfloor$ times. Although the Montgomery Ladder requires a greater number of operations than the left-to-right algorithm, it has an interesting property. The calculation of x_1 and x_2 in step 2.1 can be performed in parallel. Considering the asymptotic complexity of this algorithm, there are O(log *e*) multiplications to perform, the same as for the two previous algorithms.

Montgomery Ladder

Input: an element $g \in G$ and a positive integer $e = (e_t e_{t-1} \cdots e_1 e_0)_2$

Output: g^e

1. $x_1 \leftarrow g$, $x_2 \leftarrow g^2$
2. For *i* from $t-1$ down to 0 do the following:
   2.1. If $e_i = 0$ then: $x_2 \leftarrow x_1 \cdot x_2$ and $x_1 \leftarrow x_1^2$
   Otherwise: $x_1 \leftarrow x_1 \cdot x_2$ and $x_2 \leftarrow x_2^2$
3. Return x_1

1.5. Sliding-window algorithm

The idea of the sliding-window [3] algorithm is based on using the fact that certain parts of the binary form of the exponent are often repeated. Having determined the value of a certain partial power, we can use it repeatedly to

calculate the correct power. For this reason, we distinguish the precomputation stage in the sliding-window algorithm. The precomputation is to determine all odd powers of *g* from 1 to $2^{k-1}-1$ inclusive, where *k* is the size of the window.

The greater window size (the larger *k*), the more the operations necessary for calculating the precomputation, which should result in a fewer number of operations in the relevant part of the algorithm. However, if the window size is too large, the cost of the precomputation will not be compensated by a smaller number of multiplications in the relevant part of the algorithm.

The sliding-window algorithm

Input: g , $e=(e_te_{t-1}\cdots e_1e_0)_2$ where $e_t=1$, integer $k\geq 1$

Output: g^e

1. Precomputation:
   1.1. $g_1\leftarrow g$, $g_2\leftarrow g^2$
   1.2. For *i* from 1 to $2^{k-1}-1$ do the following: $g_{2i+1}\leftarrow g_{2i-1}\cdot g_2$
2. $A\leftarrow 1$, $i\leftarrow t$
3. While $i\geq 0$ do the following:
   3.1. If $e_i=0$ then: $A\leftarrow A^2$ and $x_1\leftarrow x_1^2$
   Otherwise:
   Find the longest bitstring $e_ie_{i-1}\cdots e_l$ such that
   $i-l+1\leq k$ and $e_l=1$, do the following:
   $A\leftarrow A^{2i-l+1}\cdot g(e_ie_{i-1}\cdots e_l)_2$, $i\leftarrow l-1$
4. Return A

The algorithm works similarly to the left-to-right algorithm. The difference is that it moves due to the window size *k*, instead of moving bit by bit. Thus, we can save to $k-1$ multiplications operations for all k-bits of the exponent.

1.5.1. Example

We want to count the number of g^{11749}. In this case, the binary form of exponent $e=11749_{(10)}=10110111100101_{(2)}$. As its length is 14, value $t=13$. The size of the window we set *k* = 3. The value of variable subscript *g* corresponds to the value of its power. In precomputation we determine:

– In step 1.1:
  $g_1\leftarrow g$, $g_2\leftarrow g\cdot g$
– In step 1.2:
  $g_3\leftarrow g_1\cdot g_2$, $g_5\leftarrow g_3\cdot g_2$, $g_7\leftarrow g_5\cdot g_2$.

The example of the algorithm for $e=11749$ is shown in Table 3. *i* runs for 14 values which corresponds to the length of the binary exponent (from 0 to 13).

In the case of the use of a window, we reduce value *i* of the window size. We raise partial result *A* to the square as many as is the window size and multiply by the value of the used pattern. If the window is not matched, variable *i* is decremented (partial result *A* is raised to the square).

Table 3

Working of sliding-window algorithm, for *e* = 11749

i	computed exponent	*A*	window
13		1	101
10	101	g^5	101
7	101101	$(g^5)^8 g^5 = g^{45}$	111
4	101101111	$(g^{45})^8 g^7 = g^{367}$	-
3	1011011110	$(g^{367})^2 = g^{734}$	-
2	10110111100	$(g^{734})^2 = g^{1468}$	101
0	10110111100101	$(g^{1468})^8 g^5 = g^{11749}$	-

1.6. "Power tree"

The analysis of algorithms for exponentiation can be performed based on the power tree [4]. The nodes of this tree are values, which exponent *e* can adopt. The following levels of the tree are obtained as a result of the multiplication of nodes at the levels above the level analyzed. The root of the tree represents the the value of 1. The path in the tree is the algorithm course for a fixed value of *e*. The path is thus a sequence of additives, for a fixed exponent *e* [4]. The path length (number of path branches) expresses the number of multiplications which the algorithm performs.

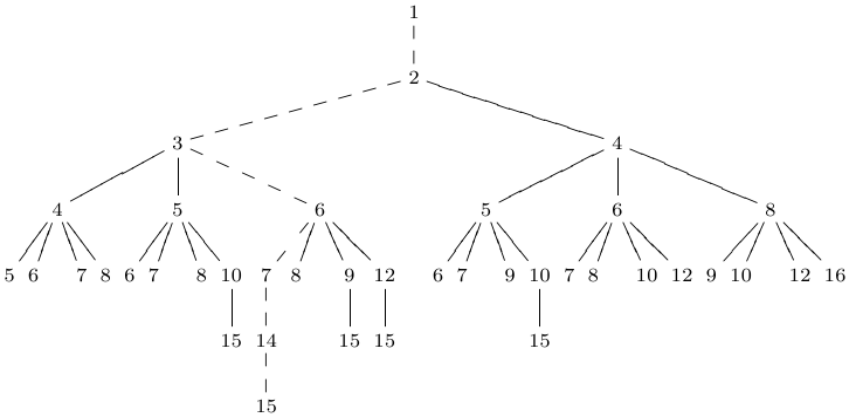

Fig. 1. Left-to-right algorithm - power tree

Analysing the course of the left-to-right algorithm for exponent e = 15 (Fig. 1, dashed line) in the power tree, we can see that the value of 15 occurs at the seventh level. Therefore, six branches connect this level with the tree root. This means that the algorithm requires in this case six multiplications. The value of 15 is for the first time at the sixth level, this means that the optimum number of multiplications necessary to calculate g^{15} is 5. The algorithm from left to right is not optimal.

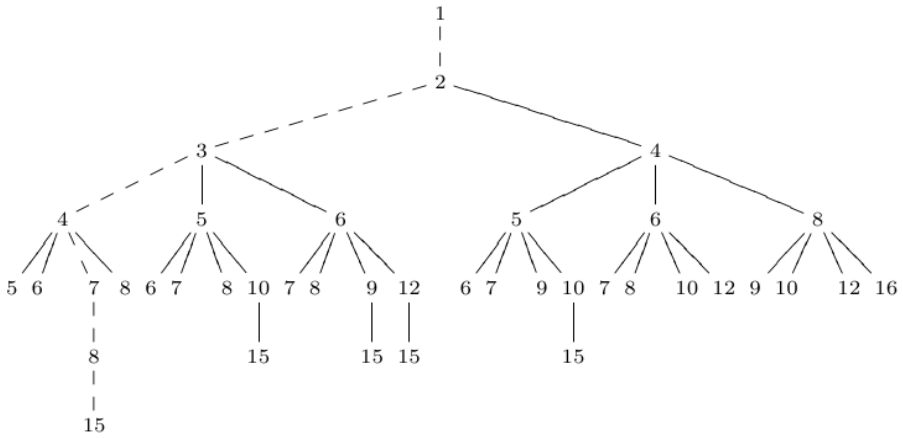

Fig. 2. Right-to-left algorithm - power tree

The dashed line in Figure 2 shows the analysis of the right-to-left algorithm for exponent e = 15. As in the previous power tree, the value of 15 occurs at the seventh level. For this exponent, the algorithm does not perform an optimum number of multiplications. The right-to-left algorithm is not optimal either.

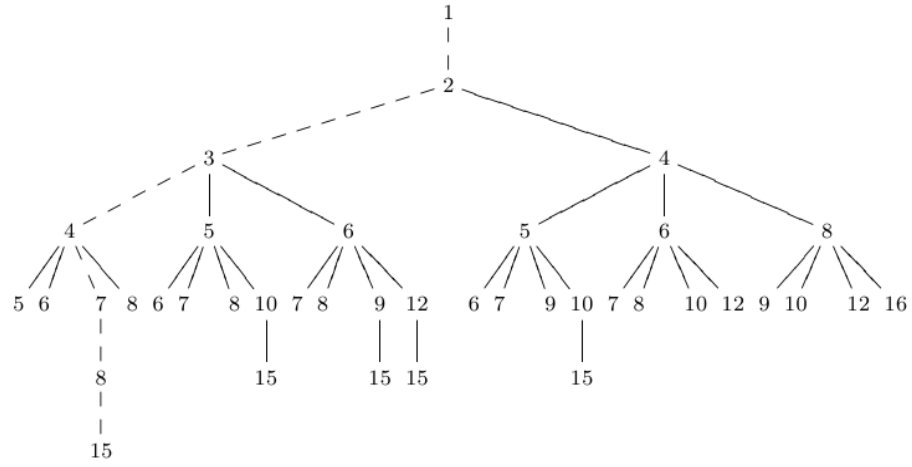

Fig. 3. Montgomery Ladder - power tree

Looking at Figure 3, we can see that using the Montgomery Ladder for exponent e = 15, the algorithm in the power tree passes through the same nodes as the right-to-left algorithm. We conclude that for the same reasons as the above given algorithms, this algorithm is not optimal either.

2. The algorithm of zero-one sequences

The idea of this method comes from the fact that it is best to reproduce the binary pattern consisting of bits of value 1. Such a pattern (we call it the pattern of ones) can be obtained by multiplying two consecutive patterns (whose length differs by one), consisting of bits 1 and 0 alternately (we call them zero-one sequences). In this algorithm, we move through the bits of exponent e to find the length of the longest sequence of bits of value 1 (marked by d).

Algorithm of zero-one sequences

Input: g , $e=(e_te_{t-1}\cdots e_1e_0)_2$ where $e_t=1$

Output: g^e

1. Precomputation:
   1.1. Find the longest string of ones and assign its length to variable d
   1.2. $g_1 \leftarrow g$, $n_0 \leftarrow 0$
   1.3. For i from 1 to $(d-1)/2$ do the following:
      1.3.1. $n_i \leftarrow 4n_{i-1}+1$
      1.3.2. $g_{2n_i} \leftarrow g_{n_i}^2$, $g_{4n_i} \leftarrow g_{2n_i}^2$, $g_{4n_i+1} \leftarrow g_{4n_i} \cdot g$
      1.3.3. If d even then: $n_{d/2} \leftarrow 4d_{d/2-1}+1$, $g_{2n_{d/2}} \leftarrow g_{n_{d/2}}^2$
2. $A \leftarrow 1$, $i \leftarrow t$
3. While $i \geq 0$ do the following:
   3.1. If $e_i=0$ then: $A \leftarrow A^2$ and $i \leftarrow i-1$
      Otherwise:
         Find the longest bitstring $e_ie_{i-1}\cdots e_l$ that matches the bit pattern,
         such that $i-l+1 \leq d$ and do the following:
         $A \leftarrow A^{2i-l+1} \cdot g_{(e_ie_{i-1}\cdots e_l)_2}$, $i \leftarrow l-1$
4. Return A

In the precomputation, we construct a base consisting of all the zero-one patterns and patterns of ones of a length less than or equal to value d. Table 4 shows the calculation of successive zero-one elements for a length not exceeding 9. The patterns of ones are always obtained by multiplying two successive patterns of different lengths. The pattern of ones of length 4 will be obtained by multiplying the patterns of No. 4 and No. 5 in Table 4.

Just as in the left-to-right algorithm, we move here through the bits of exponent *e*, from the most significant to the least significant. When the pattern occurs in exponent *e*, we reproduce it (multiply by the base element). When we have a sequence composed of bits with value 0, then we raise the value of our temporary result to the square.

Table 4

Zero-one sequences, precomputation

No.	$e_{(2)}$	$e_{(10)}$	cost
1	1	1	0
2	10	2	1
3	100	4	2
4	101	5	3
5	1010	10	4
6	10100	20	5
7	10101	21	6
8	101010	42	7
9	1010100	84	8
10	1010101	85	9
11	10101010	170	10
12	101010100	340	11
13	101010101	341	12

2.1. Example

Suppose we want to count g^{2805}. In this case, the binary form of exponent $e = 101011110101_{(2)}$. Because its length is 12, value t = 11. The longest string of ones of the exponent has a length of 4. Table 5 presents all the elements of precomputation.

Table 5

Precomputation for exponent e = 2805

$e_{(2)}$	$e_{(10)}$
1	1
10	2
100	4
101	5
1010	10
1111	15

The example of the algorithm for e = 2805 is shown in Table 6. The first column shows the values successively adopted by variable i. The value is reduced by the

length of the matched pattern that we have in column 4. Variable *A* holds the successively determined powers.

Table 6

Algorithm of zero-one sequences for exponent *e* = 2805

i	computed exponent	*A*	window
11		1	1010
7	1010	g^{10}	1111
3	10101111	$(g^{10})^{16} g^{15} = g^{175}$	-
2	101011110	$(g^{175})^2 = g^{350}$	101
0	101011110101	$(g^{350})^8 g^5 = g^{2805}$	-

Below we present the comparison of the zero-one sequences algorithm, sliding-window algorithm and left-to-right algorithm, for exponent $e = 2805$:
– 15 multiplications for algorithm of zero-one sequences
– 16 multiplications for sliding-window algorithm
– 18 multiplications for left-to-right algorithm and for right-to-left algorithm.

3. Comparison of presented algorithms effectiveness

The comparison of different exponentiation algorithms has been based on the average number of operations (multiplications), for 1000 randomly selected 128-bit exponents. The results are presented in Table 7. The results of the two best algorithms are shown in bold.

Table 7

Average number of multiplications performed for 128-bit exponents

Algorithm	Average number of operations for 1000 tests
left-to-right, binary	188.939
left-to-right, k-ary, $k = 3$	189.995
left-to-right, k-ary, $k = 4$	172.852
left-to-right, k-ary, $k = 5$	168.675
sliding window, $k = 3$	172.277
sliding window, $k = 4$	164.451
sliding window, $k = 5$	166.836
zero-one sequences	164.396

As can be seen, the best results were achieved by the sliding-window algorithm and, by the paper authors' algorithm - the zero-one sequences algorithm. The best

position of the zero-one sequences algorithm results from a good selection of pre-computation - the number of pre-designated powers and their relationships.

Conclusion

In practice, the algorithm of zero-one sequences for large exponents is less effective than the sliding-window algorithm for a properly selected value of k. This is because the number of matching and reproduced patterns for the sliding-window algorithm is greater than for the zero-one sequences algorithm. This is due to a smaller number of elements generated in the the precomputation of this algorithm.

In future, we intend to improve the algorithm of zero-one sequences in such a way that it could use a larger number of rapidly generating patterns.

References

[1] Cohen H., A Course in Computational Algebraic Number Theory, Springer-Verlag, Berlin 1993.

[2] Bach E., Shallit J.O., Algorithmic Number Theory, volume I: Efficient Algorithms, The MIT Press, Cambridge 1996.

[3] Menezes A., van Oorschot P., Vanstone S.A., Handbook of Applied Cryptography, CRC Press, Boca Raton 1999.

[4] Knuth D., The Art of Computer Programming, volume 2. Seminumerical Algorithms, Addison Wesley Longman, 1998

[5] Joye M., Yen S.M., The Montgomery powering ladder, Cryptographic Hardware and Embedded Systems-CHES 2002, s. 1-11