

FAST EXPONENTIATION  
In practice

Mads B. Tandrup (20020185)  
Martin H. Jensen (20020555)  
Rasmus N. Andersen (20020543)  
Therese F. Hansen (19993591)

6th December 2004

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Motivation . . . . .	2
1.2	Goal . . . . .	2
<b>2</b>	<b>General Purpose Algorithms</b>	<b>3</b>
2.1	Right to Left Binary Exponentiation . . . . .	3
2.2	Left to Right Binary Exponentiation . . . . .	4
2.3	Left To Right K-ary Exponentiation . . . . .	4
2.4	Modified Left-to-Right k-ary Exponentiation . . . . .	5
2.5	Sliding-Window Exponentiation . . . . .	5
2.6	Simultaneous Multiple Exponentiation . . . . .	6
2.7	Montgomery Exponentiation . . . . .	7
2.8	General Test . . . . .	7
<b>3</b>	<b>Exponentiation using addition chain/Fixed exponent</b>	<b>10</b>
3.1	Addition chain definition: . . . . .	10
3.2	Example of an addition chain: . . . . .	10
3.3	Example of exponentiating using an addition chain . . . . .	10
3.4	Calculating addition chains . . . . .	10
3.4.1	Binary add chain calculation . . . . .	11
3.4.2	Sliding-window add chain calculations . . . . .	11
3.4.3	Other way of calculating addition chains . . . . .	11
3.5	Shortest addition chain: . . . . .	11
<b>4</b>	<b>Fixed Base Algorithms</b>	<b>13</b>
4.1	Windowing algorithm . . . . .	13
4.2	Euclidean Method . . . . .	14
4.3	Test results . . . . .	15
<b>5</b>	<b>Real world usage</b>	<b>17</b>
<b>6</b>	<b>Conclusion</b>	<b>18</b>

# Chapter 1

## Introduction

### 1.1 Motivation

Arithmetic on large integers is often necessary in cryptography. Many cryptosystems depend on the possibility of computing powers  $g^e$  (exponentiation) or power products  $\prod_{1 \leq j \leq k} g_j^{e_j}$  (multi-exponentiation). Although the context of the project is cryptography many other uses of exponentiation exists fx. in Neural Networks.

### 1.2 Goal

In this project we have 3 main goals to fulfill:

- to examine a number of exponentiation algorithms in three categories.
  1. general algorithms to do exponentiation.
  2. exponentiation algorithms with fixed base.
  3. exponentiation algorithms with fixed exponent.
- to implement these algorithms in ML.
- to compare the algorithms when possible.

# Chapter 2

## General Purpose Algorithms

A general purpose algorithm is in this setting an exponentiation algorithm which doesn't work to benefit from using either the base or the exponent as fixed in a large number of exponentiations.

### 2.1 Right to Left Binary Exponentiation

This algorithm is called right-to-left because it scans the exponent  $e$  bitwise from right to left.  $S$  is initialized to  $g$  and  $S \cdot S$  is calculated in each iteration.  $e$  is divided with 2 and floored in each iteration this is in effect the same as shifting  $e$  right, and if  $e$  is odd which means that the rightmost bit is 1, then  $A \cdot S$  is computed.

---

**Algorithm 1** Right To Left Binary Exponentiation

---

INPUT: an element  $g \in G$  and integer  $e \geq 1$

OUTPUT:  $g^e$

1.  $A \leftarrow 1, S \leftarrow g$
  2. While  $e \neq 0$  do the following:
    - if  $e$  is odd then  $A \leftarrow A \cdot S$
    - $e \leftarrow \lfloor \frac{e}{2} \rfloor$
    - if  $e \neq 0$  then  $S \leftarrow S \cdot S$
  3. return( $A$ )
- 

The idea is to decrease the number of multiplications used to compute  $g^e$ , the naive algorithm would compute  $g \cdot g \dots \cdot g$ ,  $e$  times, but instead it is more effective to compute  $g^1 \cdot g^2 \cdot g^4 \dots$  because the values  $g^2 = g \cdot g$  and  $g^4 = g^2 \cdot g^2$  can be computed using fewer multiplications. An example, compute  $g^{283}$ , this would require 283 multiplications to compute in a naive way, but observe that  $283 = 100011011_2$  and that means that  $g^{283} = g^{100011011_2} = g^{256} \cdot g^{16} \cdot g^8 \cdot g^2 \cdot g^1$  which require a lot fewer calculations. The table underneath shows the values of  $A, e, S$  when computing  $g^{283}$  using this algorithm.

$A$	1	$g$	$g^3$	$g^3$	$g^{11}$	$g^{27}$	$g^{27}$	$g^{27}$	$g^{27}$	$g^{283}$
$e$	283	141	70	35	17	8	4	2	1	0
$S$	$g$	$g^2$	$g^4$	$g^8$	$g^{16}$	$g^{32}$	$g^{64}$	$g^{128}$	$g^{256}$	

Table 2.1: Example of Right to Left

**Computational efficiency:** If the exponent  $e$  has  $t$  bits in its binary representation, then the loop of the algorithm will run  $t$  times however the greatest cost is not the loop, but the multiply

operations, because they are carried out on arbitrary large integers. Assume that  $e$  has  $wt(e)$  one's in its binary form, then  $wt(e) - 1$  multiplications of  $A \cdot S$  are performed,  $t$  squarings of  $S$  and  $t$  divisions which can be performed by shifting the exponent right.

## 2.2 Left to Right Binary Exponentiation

This algorithm first appeared 200 BC in Pingala's Hindu classic 'Chandah-sutra' but is known as 'left to right binary exponentiation' or a 'square and multiply' algorithm.

The Right To Left Algorithm described earlier computes  $A \cdot S$  if  $e$  is odd. The computation  $A \times g$  can sometimes be more effective for a fixed  $g$  instead of an arbitrary large  $S$ . The Left To Right Algorithm scans the exponent from left to right, and changes the multiply operation to be more effective.

---

### Algorithm 2 Left To Right Binary Exponentiation

---

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

OUTPUT:  $g^e$

1.  $A \leftarrow 1$
  2. While  $i$  from  $t$  down to 0 do the following:
    - $A \leftarrow A \cdot A$
    - if  $e_i = 1$  then  $A \leftarrow A \cdot g$
  3. return( $A$ )
- 

Note that this algorithm uses a temporary register less than the right-to-left version because the variable  $S$  is discarded. The same example as previous, compute  $g^{283}$ .

$i$	8	7	6	5	4	3	2	1	0
$e_i$	1	0	0	0	1	1	0	1	1
$A$	$g$	$g^2$	$g^4$	$g^8$	$g^{17}$	$g^{35}$	$g^{70}$	$g^{141}$	$g^{283}$

Table 2.2: Example of Left to Right

**Computational efficiency:** This algorithm performs the same number of multiplications as the right to left binary exponentiation algorithm, if the exponent  $e$  has  $t$  bits and  $wt(e)$  one's then the algorithm performs  $wt(e) - 1 + t$  multiplications. The major difference in the multiplications is that here they are performed with a fixed value  $g$ , this is more effective.

## 2.3 Left To Right K-ary Exponentiation

The previous algorithm can be generalized to any arbitrary number system with base  $b$ . This means that when processing the exponent, instead of one bit at a time, several bits are processed at the same time. This algorithm uses precomputations which is a tool to speed up the main part of the algorithm, but of course also takes time to do.

**Computational efficiency:** Assume the exponent  $e$  has  $t$  digits in the number system with base  $b$ , the main loop in this algorithm iterates over the number digit's -  $t$  in this case. The reason for this is to process a larger part of the exponent in each iteration.

**Algorithm 3** Left To Right k-ary Exponentiation

---

 INPUT: an element  $g \in G$  and positive integer  $e = (e_t e_{t-1} \dots e_1 e_0)_b$  where  $b = 2^k$  for some  $k \geq 1$ 
OUTPUT:  $g^e$ 

1. Precomputation
    - $g_0 \leftarrow 1$
    - For  $i$  from 1 to  $(2^k - 1)$  do:  $g_i \leftarrow g_{i-1} \cdot g$  (thus,  $g_i = g^i$ )
  2.  $A \leftarrow 1$
  3. For  $i$  from  $t$  down to 0 do the following:
    - $A \leftarrow A^{2^k}$
    - $A \leftarrow A \cdot g_{e_i}$
  4. return( $A$ )
- 

## 2.4 Modified Left-to-Right k-ary Exponentiation

The Modified Left-to-Right k-ary Exponentiation algorithm is a repeated square-and-multiply algorithm almost like the Left-to-Right k-ary Exponentiation algorithm and the Left-to-Right binary Exponentiation algorithm described before, but where the Left-to-Right binary Exponentiation algorithm takes a binary number as input the Left-to-Right k-ary exponentiation algorithms (both modified and original) takes an exponent as a number in the numerical system with base  $b = 2^k$ ,  $k \geq 0$ .

The following notation is used: for each  $i$ ,  $0 \leq i \leq t$ , if  $e_i = 0$  then let  $h_i = 0$  and  $u_i = 0$  else  $e_i = 2^{h_i} \cdot u_i$ , where  $u_i$  is odd. Intuitively we divide the  $e_i$  into an odd and an even part.

We can now rewrite the main part of the Left-to-Right k-ary Exponentiation algorithm from  $A^{2^k} * g^{e_i}$  to this expression  $(A^{2^{k-h_i}} * g^{u_i})^{2^{h_i}}$ , where  $u_i$  is an odd number. These are the main rewriting steps:  $A^{2^k} * g^{u_i * 2^{h_i}} = A^{2^k} * (g^{u_i})^{2^{h_i}} = (A^{2^k * \frac{2^{h_i}}{2^{h_i}}}) * (g^{u_i})^{2^{h_i}} = (A^{2^k * 2^{-h_i}})^{2^{h_i}} * (g^{u_i})^{2^{h_i}} = (A^{2^{k-h_i}} * g^{u_i})^{2^{h_i}}$ , where  $g^{u_i} = g_{u_i}$  and  $u_i$  is odd.

**Algorithm 4** Modified left-to-right k-ary exponentiation

---

 INPUT:  $g$  and  $e = (e_t e_{t-1} \dots e_1 e_0)_b$  where  $b = 2^k$  for some  $k \geq 1$ .
OUTPUT:  $g^e$ .

1. Precomputation:
    - 1.1  $g_0 \leftarrow 1, g_1 \leftarrow g, g_2 \leftarrow g^2$ .
    - 1.2 For  $i$  from 1 to  $(2^{k-1} - 1)$  do:  $g_{2i+1} \leftarrow g_{2i-1} \cdot g_2$ .
  2.  $A \leftarrow 1$ .
  3. For  $i$  from  $t$  down to 0 do:  $A \leftarrow (A^{2^{k-h_i}} \cdot g_{u_i})^{2^{h_i}}$
  4. Return( $A$ )
- 

**Computational efficiency:**

Because it is known that  $u_i$  is always odd the Modified Left-to-Right k-ary Exponentiation algorithm differs from the original Left-to-Right k-ary Exponentiation algorithm by the fact that the modified version does fewer precomputations namely the odd exponentiations of  $g$ . The original version does  $2^k - 1$  precalculations and the modified version only does  $2^{k-1} - 1$  precalculations.

In the main computations there are still  $t+1$  steps where  $t+1$  is the length of the representation of the exponent  $e$  as a number with base  $b = 2^k$ .

## 2.5 Sliding-Window Exponentiation

The Sliding-Window Exponentiation algorithm is a repeated square-and-multiply algorithm, which takes a base  $g$ , an exponent  $e$  as a binary number and the most significant bit  $e_t = 1$  and an integer  $k \geq 1$  as the window size. The output is  $g^e$ . The new thing here is the window size, as is reflected in the name. This is a variable given to the algorithm that specifies how many precomputations

that is done and how big steps we take in the main part of the algorithm. The larger the window size the larger the possibility of fewer multiplications not counting precomputations. This time there is also only need for precomputing the odd exponents of  $g$  because a bitstring with a one at its least significant bit is odd. It is recommended to experiment with different window sizes within the use of same exponent to find the optimal one.

---

**Algorithm 5** Sliding-window exponentiation
 

---

INPUT:  $g, e = (e_t e_{t-1} \dots e_1 e_0)_2$  with  $e_t = 1$ , and 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:  $g_{2i+1} \leftarrow g_{2i-1} \cdot g_2$ .
  2.  $A \leftarrow 1, i \leftarrow t$ .
  3. While  $i \geq 0$  do the following:
    - 3.1 If  $e_i = 0$  then do:  $A \leftarrow A^2, i \leftarrow i - 1$
    - 3.2 Otherwise ( $e_i \neq 0$ ), find the longest bit string  $e_i e_{i-1} \dots e_l$  such that  $i - l + 1 \leq k$  and  $e_l = 1$ , and do the following
 
$$A \leftarrow A^{2^{i-l+1}} \cdot g_{(e_i e_{i-1} \dots e_l)_2}, i \leftarrow l - 1$$
  4. Return( $A$ )
- 

**Computational efficiency:**

The Sliding Window Exponentiation algorithm is potentially very fast. It is very difficult to convey an accurate idea about the complexity of this algorithm, but the worst case (where  $e = 2^t$  and there only is one 1) takes the main loop in steps of only one and therefore takes  $t + 1$  times, where  $t + 1$  is the length of the binary representation of  $e$ . The good case is where  $k$  divides  $t + 1$  and there is enough ones to make the longest bit string of length  $k$  every time. Then you only have to do  $\frac{t+1}{k}$  iterations and in every iteration  $2^k$  squarings and 1 multiplication. There is  $2^{k-1} - 1$  multiplications in the precomputation.

## 2.6 Simultaneous Multiple Exponentiation

The Simultaneous Multiple Exponentiation algorithm is an algorithm closely related to the Left-to-Right Binary Exponentiation algorithm but instead of taking a base  $g$  and an exponent  $e$  it takes a list of bases  $g_0 \dots g_{k-1}$  and a list of exponents  $e_0 \dots e_{k-1}$  and gives  $g_0^{e_0} * g_1^{e_1} * \dots * g_{k-1}^{e_{k-1}}$ . This  $k$  decides how many precomputations is done. The larger the  $k$  the more complex precomputations consisting of many multiplications and the larger number of precomputations ( $2^k : G_0$  to  $G_{2^k-1}$ ). Excluding precomputations the algorithm does a squaring and a multiplication  $t$  times, where  $t$  is the length of the binary representation of the exponents. The smaller the exponents the fewer multiplications excluding precomputations. The trade off here is either large exponents and therefore large  $t$  or many individual exponentiations and therefore a large  $k$  with large and complicated precomputations containing many multiplications.

---

**Algorithm 6** Simultaneous multiple exponentiation
 

---

INPUT: group elements  $g_0, g_1, \dots, g_{k-1}$  and non-negative  $t$ -bit integers  $e_0, e_1, \dots, e_{k-1}$ .

OUTPUT:  $g_0^{e_0} \cdot g_1^{e_1} \cdot \dots \cdot g_{k-1}^{e_{k-1}}$ .

1. Precomputation:
    - For  $i$  from 0 to  $(2^k - 1)$ :  $G_i \leftarrow \prod_{j=0}^{k-1} g_j^{i_j}$  where  $i = (i_{k-1} \dots i_0)_2$ .
  2.  $A \leftarrow 1$ .
  3. For  $i$  from 1 to  $t$  do the following:  $A \leftarrow A \cdot A, A \leftarrow A \cdot G_{I_i}$ .
  4. Return( $A$ )
- 

**Computational efficiency:**

The complexity of this algorithm is closely tied to the complexity of the Left-to-Right Binary Exponentiation algorithm because this algorithm is just a simultaneous use of that algorithm. This algorithm does a squaring and a multiplication  $t - 1$  times (here the trivial squaring  $1^2$  and the trivial multiplication  $1 * G_{I_i}$  is not included in the count) and precomputates at most  $(2^k - 2)$  multiplications (the trivial not counted).

## 2.7 Montgomery Exponentiation

This algorithm combines Left-to-Right binary exponentiation and Montgomery multiplication. The basic idea in this algorithm is the same as the original except that the multiplications are replaced by Montgomery multiplications which given  $m, x, y$  in number system  $b$  each with  $n$  digits and  $R = b^n$  where  $0 \leq x, y \leq m$  and  $\gcd(m, b) = 1$  returns  $xyR^{-1} \bmod m$ .

We have decided not to include this multiply operation in our test suite, but it is used in Java's BigInteger implementation as mentioned in the chapter about exponentiation in practice.

## 2.8 General Test

The General test consists of a series of exponentiations where the algorithms are allowed to first make some precalculations and then the main calculation. The exponentiations are carried out separately, which means that the algorithms cannot save results from earlier exponentiations in order to gain a speed up.

The numbers used in the test consists of a base, exponent and a modulo. The bases used are around 32 bit in size, the exponents are about 7000 bit's in size and modulo number is about 1800 bits in length

The test suite and all of the algorithms have been implemented in ML and compiled using MLton version 20041109, this has some very apparent drawbacks. Some of the algorithms need to index into arrays but the implementation of arrays in ML are very inefficient, which the graf also shows for Sliding Window. As shown in the section about sliding window, it uses an array of size  $2^{k-1} - 1$  where  $k$  is the window size, that means that in the example with window size 8, an array of size 127 is needed, we decided to dump the test with a window size of 16 because it require an array of size 32767 and has a running time completely out of proportions.

We wanted to compare our implementations against the one in GMP, but unfortunately ML only supports the single precision power function in GMP that limits the exponent to 32 bits, the other implementation that allows arbitrary large integers is not supported by ML.

	Precomputations (sec.)	Computation (sec.)	All computations (sec.)
Right-to-Left Binary	0	0,34	0,34
Left-to-Right Binary	0	0,22	0,22
Left-to-Right k-ary - base 2	0,26	0,64	0,9
Left-to-Right k-ary - base 8	0,26	0,65	0,92
Left-to-Right k-ary - base 16	0,26	0,64	0,9
Sliding Window - window size 2	0,79	0,27	1,06
Sliding Window - window size 4	0,79	0,39	1,18
Sliding Window - window size 6	0,79	1,19	1,98
Sliding Window - window size 8	0,79	5,77	6,56

Table 2.3: General test results

# General

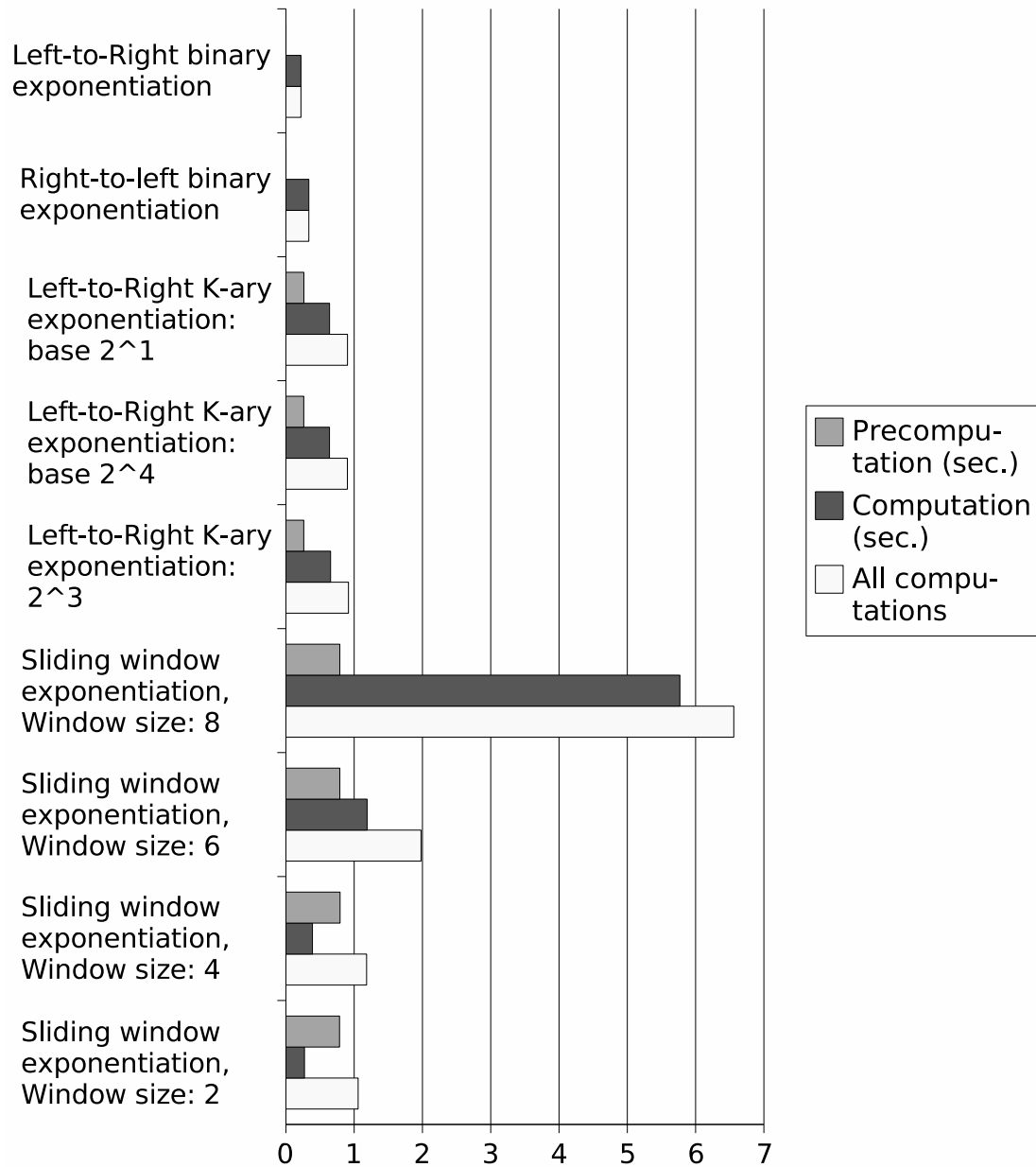


Figure 2.1: General test results

## Chapter 3

# Exponentiation using addition chain/Fixed exponent

### 3.1 Addition chain definition:

An addition chain of length  $r$  for the positive integer  $r$  is a sequence  $a_0, a_1, \dots, a_r$  of positive integers with  $a_0 = 1$  and  $a_r = e$ , and an associated sequence  $w_1, w_2, \dots, w_r$  of pairs  $w_i = (i_1, i_2)$ ,  $0 \leq i_1, i_2 \leq i$ , such that  $a_i = a_{i_1} + a_{i_2}$  for every  $a_i$ , where  $i \in [0; r]$ .

### 3.2 Example of an addition chain:

Let  $a_0, a_1, a_2, a_3, a_4, a_5 = 1, 2, 3, 5, 10, 13$  be a sequence and  $w_1, w_2, w_3, w_4, w_5 = (0, 0), (0, 1), (1, 2), (3, 3), (2, 4)$  be the associated sequence. It is easy to check that this is an addition chain for 13 by the definition.

### 3.3 Example of exponentiating using an addition chain

Calculation  $b^g$  can be done by the help of additions chains. For example calculating  $b^{13}$ , can be done by making use of the addition chain above. Start by making an array  $v$  of the same length as the addition chain and with  $v[0]=b$ . Then use  $w_1 = (i_1, i_2)$  to calculate  $v[1]$ , by setting  $v[1] = v[i_1] * v[i_2] = v[0] * v[0] = b^2$ . Next use  $w_2 = (i_1, i_2)$  to calculate  $v[2]$ , by setting  $v[2] = v[i_1] * v[i_2] = v[0] * v[1] = b^3$ . Continue this until we have gone through the whole addition chain. Then will the last index in the array be the result  $v[5] = b^{13}$ . It is not hard to see why this is a valid approach at calculating powers.

---

**Algorithm 7** Algorithm for exponentiation using addition chains

---

Input: An Addition chain  $A=(a_0, a_1, \dots, a_r)$  of length  $r$  for a positive integer  $e$  and the associated sequence  $w_1, w_2, \dots, w_r$ , where  $w_i = (i_1, i_2)$ . A group element  $b$ .

Output:  $b^e$

1. Make an array  $g$  of length  $r$ , with  $g[0]=g$
  2. For  $i$  from 1 to  $s$  do:  $g[i] = g[i_1] * g[i_2]$
  3. Return  $g[s]$
- 

### 3.4 Calculating addition chains

The algorithm above is just given an addition chain as input, without saying anything about how it is computed. An efficient way of computing the add chain is needed, and the shorter the

generated add chain is, the better. But luckily it is not hard to come up with ways of computing add chains.

### 3.4.1 Binary add chain calculation

For example look at the left-to-right binary exponentiation algorithm described elsewhere in the report, it kind of calculates an add chain on the fly for the exponent, while calculating the exponentiation. The algorithm works by looking at the binary representation of the exponent, for example  $13 = 1101$ , and iterate though the bits from left to right, in every iteration the result is squared and if there is a 1 bit in the binary representation, then it is also multiplied by the base. This can be transformed into calculate an add chain for the exponent, also by looking at the binary representation as below.

Binary add chain algorithm: Start by finding the first 1 bit from left-to-right in the binary, set  $a_0 = 1$  and iterate though the rest of the bits from left-to-right as follows.

If we at  $a_i$  in the add chain and want to calculate the next value in the chain, we look at the next bit in the binary representation, if it is a 0 bit,  $w_{i+1} = (i, i)$  is added to the associated sequence and  $a_{i+1} = a_i + a_i$ . But if it is 1 bit,  $w_{i+1} = (i, i)$  is added to the associated sequence and  $a_{i+1} = a_i + a_i$  like before, but now  $w_{i+2} = (0, i + 1)$  is also added to the associated sequence and  $a_{i+2} = a_{i+1} + a_0$ .

So  $13 = 1101$ , will give an addition chain  $(1,2,3,6,12,13)$  with associated sequence  $(0,0), (0,1), (2,2), (3,3), (4,0)$  using the binary add chain calculation.

Add chain calculation using the binary way usually don't give very short chain, so finding better ways which finds shorter chains would be desirable, because the shorter the add chain is, the fewer multiplications will be needed.

### 3.4.2 Sliding-window add chain calculations

Again looking at an algorithm discussed earlier, Sliding-window exponentiation algorithm, which also can derive an algorithm for calculation add chains. In algorithm below the associated chain is not shown, but should be easy to add.

Sliding-window add chain algorithm: Want to compute add chain of  $e$  using window size  $k$ .

Precompute all the odd integers from 1 to  $2^k - 1$  and insert odd integer  $i$  into the add chain at position  $i/2$  rounded up.

Start at the left most 1 bit in the binary representation. Start the iteration though the binary representation. If bit is a 0 set the next number  $a_{i+1}$  in the add chain to be  $a_{i+1} = a_i + a_i$  and go to the next bit in the binary representation. If the bit is a 1 bit, find the longest length, smaller than or equal to  $k$ , of consecutive bits  $e_j, e_{j-1}, \dots, e_l$  which ends in a 1 bit and start in the position we are at in the binary representation. Then generate a sequence in the add chain which double the last element in the add chain  $i - l - 1$  times and set the next element in the add chain to be  $a_{i+1} = a_i + a_s$  where  $s = (e_j, e_{j-1}, \dots, e_l)_2 / 2$  round up. Jump the  $e_{l-1}$  bit and continue the iteration.

### 3.4.3 Other way of calculating addition chains

The sliding-window method will on average give shorter chains than the binary, but the fun don't stop here: Many other ways of computing add chains exists. For example can the  $k$ -ary and the naive algorithms for calculating exponentiations also be transformed into calculating addition chains.

## 3.5 Shortest addition chain:

- Lower bound of the length of the addition chain for a positive integer  $e$ :  $\log_2(e) + \log_2 H(e) - 2.13$

- Upper bound of the length of the addition chain for a positive integer  $e$ :  $\log_2(e) + H(e) - 1$

where  $H(e)$  is the Hamming weight of  $e$ .

It would be very nice if we could efficiently compute the shortest addition chain, but unfortunately this is a NP-problem. Notice however that when a addition chain for a exponentiation has been calculated, it can be used many times. So applications where the exponent is fixed can maybe benefit from precalculating the addition chain and using it many times.

# Chapter 4

## Fixed Base Algorithms

When computing a large number of exponentiations with a fixed base  $g$  and an varying exponent  $e$ , it is possible to speed up the exponentiation calculation, by precomputing some values based on  $g$ . The algorithms described in this text will all use the same precomputations. They will simple for some given set  $\{b_0, b_1, \dots, b_t\}$  calculate  $\{g^{b_0}, g^{b_1}, \dots, g^{b_t}\}$ . To calculate these exponents the precomputation will use another exponentiation algorithm to precompute a number of exponentiations of  $g$ .

In this chapter  $\{b_0, b_1, \dots, b_t\}$  is a none-empty set of positive integers, such that any exponent  $1 \leq e \leq \text{some max}$  can be written as  $e = \sum_{i=0}^t e_i b_i$ , where  $0 \leq e_i < h$ . The most obvious choice for  $h$  and  $\{b_0, b_1, \dots, b_t\}$  would normally be dictated by the base  $b$  used to represent  $e$ , in this scenario one would select  $h = b$  and  $b_i = b^i$ . This will however for a binary representation of  $e$  lead to more precomputations and more storage needed to store the result of the precomputations. So depending on the application one will have to determine a balance between the amount of precomputation versus the time needed for one exponentiation.

### 4.1 Windowing algorithm

The algorithm uses the fact that, given the above definition,  $g^e$  can be rewritten as follows:

$$\begin{aligned} g^e &= g^{b_0 e_0 + b_1 e_1 + \dots + b_t e_t} = \prod_{i=0}^t g^{b_i e_i} \\ &= \prod_{j=1}^{h-1} \left( \prod_{i \in \{k | e_k = j\}} g^{b_i e_i} \right) = \prod_{j=1}^{h-1} \left( \prod_{i \in \{k | e_k = j\}} g^{b_i} \right)^j \end{aligned}$$

But there are still some exponentiations, but since  $h$  in most cases is selected small, it will only be a small number of small exponentiations. But these can be calculated during the product operation. We can write each iteration as:

$j$	$A$	$B$
$h-1$	$s_{h-1}$	$s_{h-1}$
$h-2$	$s_{h-1} \cdot (s_{h-1} \cdot s_{h-2})$	$s_{h-1} \cdot s_{h-2}$
$\vdots$	$\vdots$	$\vdots$
$j$	$s_{h-1}^{h-j} \cdot s_{h-2}^{h-j-1} \cdot \dots \cdot s_j$	$s_{h-1} \cdot s_{h-2} \cdot \dots \cdot s_j$
$\vdots$	$\vdots$	$\vdots$
$1$	$s_{h-1}^{h-1} \cdot s_{h-2}^{h-2} \cdot \dots \cdot s_1$	$s_{h-1} \cdot s_{h-2} \cdot \dots \cdot s_1$

where  $s_j = \left( \prod_{i \in \{k | e_k = j\}} g^{b_i} \right)$ .

---

**Algorithm 8** Windowing method for a fixed base

---

INPUT:  $h, \{g^{b_0}, g^{b_1}, \dots, g^{b_t}\}$  and  $\{e_0, e_1, \dots, e_t\}$  such that  $e = \sum_{i=0}^t e_i b_i$ .OUTPUT:  $g^e$ . $A \leftarrow 1, B \leftarrow 1$ .For  $j$  from  $h - 1$  down to 1 do  For each  $i \in \{k | e_k = j\}$  do     $B \leftarrow B \cdot g^{b_i}$ .   $A \leftarrow A \cdot B$ .Return  $A$ .

---

**Computational efficiency:** Since there are only  $t + 1$   $e_i$ 's and  $B$  is initialized to one, at most  $t$  multiplications will be done in the inner loop overall. The outer loop loops  $h - 1$  times, but the first time  $A$  will be initialized to one, so with proper treatment the outer loop will only do  $h - 2$  multiplications (excluding the inner loop). In total the algorithm will perform  $t + h - 2$  multiplications.

**Storage efficiency:** The algorithm will use storage for the  $t + 1$  elements  $g^{b_i}$ . So for a small  $h$  a large number of values will need to be precomputed in order to be able to use sufficiently large exponents.

## 4.2 Euclidean Method

The algorithm uses the same precomputations as the Windowing algorithm on the preceding page, but the calculations are different. The algorithm determines two different integers  $M$  and  $N$  such that  $\forall i \notin \{M, N\} : x_M \geq x_N \geq x_i$ .

---

**Algorithm 9** Euclidean Method for a fixed base

---

INPUT:  $\{g^{b_0}, g^{b_1}, \dots, g^{b_t}\}$  and  $\{e_0, e_1, \dots, e_t\}$  such that  $e = \sum_{i=0}^t e_i b_i$ .OUTPUT:  $g^e$ .For  $i$  from 0 to  $t$  do   $g_i \leftarrow g^{b_i}$    $x_i \leftarrow e_i$ Find  $M, N$  such that  $M \neq N \wedge \forall i \notin \{M, N\} : x_M \geq x_N \geq x_i$ .While  $x_N \neq 0$  do   $q \leftarrow \left\lfloor \frac{x_M}{x_N} \right\rfloor$    $g_N \leftarrow (g_M)^q \cdot g_N$    $x_M \leftarrow x_M \bmod x_N$   Find  $M, N$  such that  $M \neq N \wedge \forall i \notin \{M, N\} : x_M \geq x_N \geq x_i$ .Return  $g_M^{x_M}$ 

---

**Computational efficiency:** In most cases the  $q$  computed in the while loop, will be one or very small.

**Storage efficiency:** Since the division  $\frac{x_M}{x_N}$  can be done logarithmic in the size of the input, it is possible to take advantage of larger  $h$  (leading to larger  $x_i$  values) which will limit the required number of precomputed values.

	Average computation time (sec.)
Left-to-Right binary exponentiation	0.21
Right-to-left binary exponentiation	0.34
Left-to-Right K-ary exponentiation: base $2^1$	0.94
Left-to-Right K-ary exponentiation: base $2^4$	0.93
Left-to-Right K-ary exponentiation: base $2^3$	0.91
Sliding window exponentiation, Window size: 6	2.25
Sliding window exponentiation, Window size: 4	1.23
Sliding window exponentiation, Window size: 2	1.06
Fixed Base Window, $t = 2000$	0.07
Fixed Base Euclidean, $t = 2000$	0.41

Table 4.1: Fixed base tests

### 4.3 Test results

The fixed base tests were performed with a fixed 32 bit base and  $n$  different 6500 bits exponents. All calculations were done modulo a 1800 bit number. The generic algorithms from Chapter 2 were simply tested  $n$  times with the base and the  $i$ 'th exponent. The result can be seen in Table 4.1 and Figure 4.1 on page 16 (the computation time is excluding the time required to do the precomputations). Both fixed base algorithms uses the same precomputation and took 3.5 minutes. But the average time for one exponentiation of the Fixed base window algorithm (8) is 3 times faster than the Left-to-right algorithm.

When deciding between using one of the generic algorithms described in Chapter 2 and one of the above algorithms, the time used for precomputations should be compared to the number of exponentiations should be weighted. Based on the test results in Table 4.1 the break-even point, when the Fixed base window algorithm surpasses the Left-to-right algorithm, can be computed to be approximately 1520 exponentiations.

However real-world implementations might lead to other break-even point. Consider some kind of embedded device that will need to perform numerous exponentiations of a fixed base given. In such a situation, one might choose to do the precompute all the necessary data on a fast computer, and download them to the embedded device. The device will then be able to perform lightning fast exponentiations on one fixed exponent.

As seen in the test data, the Fixed base Euclidean algorithm is almost 6 times slower than the Fixed base Window. This might be due to the problems with the slow indexing into ML arrays as the Fixed base Euclidean algorithm is implemented using arrays.

## Fixed base



Figure 4.1: Fixed base tests

## Chapter 5

# Real world usage

In order to compare the results from this project with real world implementations we have examined some examples, both by looking at the source code of the software and documentation provided by the authors. Figure 5.1 below shows some software projects and their implementations.

- GMP (GNU Multiple Precision Arithmetic Library): 2 kinds of exponentiation algorithms  
Normal powering: Square and multiply  
Modular powering: Sliding-window exponentiation
- GPG(Open-source implementation of PGP): Slightly modified version of GMP
- GNU Scientific Library: Square and multiply
- ACME Labs Software Bigint: Square and multiply
- Perl: Square and multiply
- Java 1.5.0: Square and multiply, and Montgomery multiplication

Figure 5.1: Real world Implementations

The striking thing about the implementations is that they all use general purpose algorithms, some with slight modifications to increase running times. The software development kits in the list eg. Java Developers Kit, Perl and GMP has to provide algorithms that performs well in the most applications because they are domain independent programming languages that has no way of predicting the eventual application where they are used.

## Chapter 6

# Conclusion

This report emphasizes several important aspects of exponentiation algorithms in practice.

In the implementation of the all purpose algorithms the comparison show that the left-to-right algorithm is the simplest and always the fastest. The left-to-right algorithm is also one of the simplest algorithm to implement, so for all purpose use there is not really any other algorithm to consider. Theoretically some of the other algorithms performs fewer multiplications, but the overhead of reducing the number of multiplications does not pay off in our implementation.

For exponentiation of a fixed base to a large number of different exponents, the comparison show that the fixed base algorithms perform better after the precomputations have been done. Unfortunately the precomputations are many magnitudes slower than a small number of exponentiations using simple left-to-right algorithm. The test of the implementation shows that exponentiation is 3 times faster than left-to-right implementation, so for a large number of exponentiations the fixed base algorithm will surpass the left-to-right algorithm including time for the precomputation for the fixed base algorithms.

By choice the fixed exponent algorithm were not implemented because a large number of the ideas of calculating addition chains were already used in general purpose algorithms. Calculating a shortest addition chain is a NP-hard problem, so we could not find an optimal solution and implementing many of the heuristics we found, would only be ineffective implementations of some of the ideas used in the general purpose algorithms.

The implementation side of the project was problematic due to the use of Standard ML, a choice made because of the nice mathematically functional programming style. But since some of the algorithms use direct indexing into large arrays and the ML implementation of arrays is slow, the result is slow performance of the implementation. So every time the algorithms is implemented using arrays the time estimate is flawed.

# Bibliography

- [1] Handbook of Applied Cryptography - A. Menezes, P. van Oorschot and S. Vanstone, CRC Press 1996
- [2] Cetin Kaya Koc: RSA implementation - <http://islab.oregonstate.edu/koc/ece575/notes/rsa4.pdf>
- [3] Cetin Kaya Koc: High speed RSA implementation - <ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf>
- [4] GNU Multiple Precision Arithmetic Library: <http://www.swox.com/gmp/gmp-man-4.1.4.pdf>
- [5] Gnu science librar -: <ftp://ftp.gnu.org/gnu/gsl/> and [http://www.gnu.org/software/gsl/manual/gsl-ref\\_4.html#SEC33](http://www.gnu.org/software/gsl/manual/gsl-ref_4.html#SEC33)
- [6] Acme's implementation of a BigInt library - <http://www.acme.com/software/bigint/>
- [7] Perl source code - <http://www.perl.com/CPAN/src/>
- [8] GPG: Open source implementation of PGP - <http://www.gnupg.org/>
- [9] JDK 1.5.0 source code
- [10] MLton version 20041109 - <http://www.mlton.org>
- [11] SML Implementation of algorithms - <http://www.daimi.au.dk/~tandrup/crypto/algorithms.tar.gz>
- [12]  $2^k$ -ary exponentiation - Bodo Möller - <http://www.win.tue.nl/~henkvt/2k-ary-exp.pdf>

# List of Algorithms

1	Right To Left Binary Exponentiation . . . . .	3
2	Left To Right Binary Exponentiation . . . . .	4
3	Left To Right k-ary Exponentiation . . . . .	5
4	Modified left-to-right k-ary exponentiation . . . . .	5
5	Sliding-window exponentiation . . . . .	6
6	Simultaneous multiple exponentiation . . . . .	6
7	Algorithm for exponentiation using addition chains . . . . .	10
8	Windowing method for a fixed base . . . . .	14
9	Euclidean Method for a fixed base . . . . .	14