Chapter 9

Exponentiation

Christophe Doche

Contents in Brief

Given an element x of a group $(G, \times)$ and an integer $n \in \mathbb{Z}$ one describes in this chapter efficient methods to perform the *exponentiation* x^n. Only positive exponents are considered since $x^n = (1/x)^{-n}$ but nothing more is assumed especially regarding the structure and the properties of G. See Chapter 11 for specific improvements concerning finite fields. Two elementary operations are used, namely multiplications and squarings. The distinction is made for performance reasons since squarings can often be implemented more efficiently; see Chapters 10 and 11 for details. In the context of elliptic and hyperelliptic curves, the computations are done in an abelian group denoted additively $(G, \oplus)$. The equivalent of the exponentiation x^n is the *scalar multiplication* $[n]P$. All the techniques described in this chapter can be adapted in a trivial way, replacing multiplication by addition and squaring by doubling. See Chapter 13 for additional details concerning elliptic curves and Chapter 14 for hyperelliptic curves.

Exponentiation is a very important operation in algorithmic number theory. For example, it is intensively used in many primality testing and factoring algorithms. Therefore efficient methods have been studied over centuries. In cryptosystems based on the discrete logarithm problem (cf. Chapter 1) exponentiation is often the most time-consuming part, and thus determines the efficiency of cryptographic protocols like key exchange, authentication, and signature.

Three typical situations occur. The *base point* x and the *exponent* n may both vary from one computation to another. Generic methods will be used to get x^n in this case. If the same exponent is used several times a closer study of n, especially the search of a short addition chain for n, can lead to substantial improvements. Finally, if different powers of the same element are needed, some precomputations, whose cost can be neglected, give a noticeable speedup.

Most of the algorithms described in the remainder of this chapter can be found in [MEOO+ 1996, GOR 1998, KNU 1997, STA 2003, BER 2002].

9.1 Generic methods

In this section both x and n may vary. Computing x^n naïvely requires $n-1$ multiplications, but much better methods exist, some of them being very simple.

9.1.1 Binary methods

It is clear that x^{2^k} can be obtained with only k squarings, namely $x^2, x^4, x^8, \ldots, x^{2^k}$. Building upon this observation, the following method, known for more than 2000 years, allows us to compute x^n in $O(\lg n)$ operations, whatever the value of n .

Algorithm 9.1 Square and multiply method

INPUT: An element x of G and a nonnegative integer $n = (n_{\ell-1} \ldots n_0)_2$.
OUTPUT: The element $x^n \in G$.

1. $y \leftarrow 1$ and $i \leftarrow \ell - 1$
2. **while** $i \geqslant 0$
3. $\quad y \leftarrow y^2$
4. $\quad$ **if** $n_i = 1$ **then** $y \leftarrow x \times y$
5. $\quad i \leftarrow i - 1$
6. **return** y

This method is based on the equality

$$x^{(n_{\ell-1} \ldots n_{i+1} n_i)_2} = \left(x^{(n_{\ell-1} \ldots n_{i+1})_2}\right)^2 \times x^{n_i}.$$

As the bits are processed from the most to the least significant one, Algorithm 9.1 is also referred to as the *left-to-right binary method*.

There is another method relying on

$$x^{(n_i n_{i-1} \ldots n_0)_2} = x^{n_i 2^i} \times x^{(n_{i-1} \ldots n_0)_2}$$

which operates from the right to the left.

Algorithm 9.2 Right-to-left binary exponentiation

INPUT: An element x of G and a nonnegative integer $n = (n_{\ell-1} \ldots n_0)_2$.
OUTPUT: The element $x^n \in G$.

1. $y \leftarrow 1$, $z \leftarrow x$ and $i \leftarrow 0$
2. **while** $i \leqslant \ell - 1$ **do**
3. $\quad$ **if** $n_i = 1$ **then** $y \leftarrow y \times z$

4. $z \leftarrow z^2$
5. $i \leftarrow i + 1$
6. **return** y

Remarks 9.3

(i) Algorithm 9.1 is related to Horner's rule, more precisely computing x^n is similar to evaluating the polynomial $\sum_{i=0}^{\ell-1} n_i X^i$ at $X = 2$ with Horner's rule.

(ii) Further enhancements may apply to the products $y \times x$ in Algorithm 9.1 since one of the operands is fixed during the whole computation. For example, if x is well chosen the multiplication can be computed more efficiently. Such an improvement is impossible with Algorithm 9.2 where different terms of approximately the same size are involved in the products $y \times z$.

(iii) In Algorithm 9.2, whatever the value of n, the extra variable z contains the successive squares $x^2, x^4, \ldots$ which can be evaluated in parallel to the multiplication.

The next example provides a comparison of Algorithms 9.1 and 9.2.

Example 9.4 Let us compute x^{314}. One has $314 = (100111010)_2$ and $\ell = 9$.

Algorithm 9.1									
i	8	7	6	5	4	3	2	1	0
n_i	1	0	0	1	1	1	0	1	0
y	x	x^2	x^4	x^9	x^{19}	x^{39}	x^{78}	x^{157}	x^{314}
Algorithm 9.2									
i	0	1	2	3	4	5	6	7	8
n_i	0	1	0	1	1	1	0	0	1
z	x	x^2	x^4	x^8	x^{16}	x^{32}	x^{64}	x^{128}	x^{256}
y	1	x^2	x^2	x^{10}	x^{26}	x^{58}	x^{58}	x^{58}	x^{314}

The number of squarings is the same for both algorithms and equal to the bit length $\ell \sim \lg n$ of n. In fact this will be the case for all the methods discussed throughout the chapter.

The number of required multiplications directly depends on $\nu(n)$ the *Hamming weight* of n, i.e., the number of nonzero terms in the binary expansion of n. On average $\frac{1}{2} \lg n$ multiplications are needed.

Further improvements introduced below tend to decrease the number of multiplications, leading to a considerable speedup. Many algorithms also require some precomputations to be done. In the case where several exponentiations with the same base have to be performed in a single run, these precomputations need to be done only once per session, and if the base is fixed in a given system, they can even be stored, so that their cost might become almost negligible. This is the case considered in Algorithms 9.7, 9.10, and 9.23. Depending on the bit processed, a single squaring or a multiplication and a squaring are performed at each step in both Algorithms 9.1 and 9.2. This implies that it can be possible to retrieve each bit and thus the value of the exponent from an analysis of the computation. This has serious consequences when the exponent is some secret key. See Chapters 28 and 29 for a description of side-channel attacks. An elegant technique, called *Montgomery's ladder*, overcomes this issue. Indeed, this variant of Algorithm 9.1 performs a squaring and a multiplication at each step to compute x^n.

Algorithm 9.5 Montgomery's ladder

INPUT: An element $x \in G$ and a positive integer $n = (n_{\ell-1} \dots n_0)_2$.
OUTPUT: The element $x^n \in G$.

1. $x_1 \leftarrow x$ and $x_2 \leftarrow x^2$
2. **for** $i = \ell - 2$ **down to** 0 **do**
3. **if** $n_i = 0$ **then**
4. $x_1 \leftarrow x_1^2$ and $x_2 \leftarrow x_1 \times x_2$
5. **else**
6. $x_1 \leftarrow x_1 \times x_2$ and $x_2 \leftarrow x_2^2$
7. **return** x_1

Example 9.6 To illustrate the method, let us compute x^{314} using, this time, Algorithm 9.5. Starting from $(x_1, x_2) = (x, x^2)$, the next values of (x_1, x_2) are given below.

i	7	6	5	4	3	2	1	0
n_i	0	0	1	1	1	0	1	0
(x_1, x_2)	(x^2, x^3)	(x^4, x^5)	(x^9, x^{10})	(x^{19}, x^{20})	(x^{39}, x^{40})	(x^{78}, x^{79})	(x^{157}, x^{158})	(x^{314}, x^{315})

See also Chapter 13, for a description of Montgomery's ladder in the context of scalar multiplication on an elliptic curve.

9.1.2 Left-to-right 2^k-ary algorithm

The general idea of this method, introduced by Brauer [BRA 1939], is to write the exponent on a larger base $b = 2^k$. Some precomputations are needed but several bits can be processed at a time.

In the following the function σ is defined by $\sigma(0) = (k, 0)$ and $\sigma(m) = (s, u)$ where $m = 2^s u$ with u odd.

Algorithm 9.7 Left-to-right 2^k-ary exponentiation

INPUT: An element x of G, a parameter $k \geqslant 1$, a nonnegative integer $n = (n_{\ell-1} \dots n_0)_{2^k}$ and the precomputed values $x^3, x^5, \dots, x^{2^k-1}$.
OUTPUT: The element $x^n \in G$.

1. $y \leftarrow 1$ and $i \leftarrow \ell - 1$
2. $(s, u) \leftarrow \sigma(n_i)$ $[n_i = 2^s u]$
3. **while** $i \geqslant 0$ **do**
4. **for** $j = 1$ **to** $k - s$ **do** $y \leftarrow y^2$
5. $y \leftarrow y \times x^u$
6. **for** $j = 1$ **to** s **do** $y \leftarrow y^2$

7. $i \leftarrow i - 1$
8. **return** y

Remarks 9.8

(i) Lines 4 to 6 compute $y^{2^k} x^{n_i}$ i.e., the exact analogue of $y^2 x^{n_i}$ in Algorithm 9.1. To reduce the amount of precomputations, note that $(y^{2^{k-s}} x^u)^{2^s} = y^{2^k} x^{n_i}$ is actually computed.

(ii) The number of elementary operations performed is $\lg n + \lg n\big(1 + o(1)\big) \lg n / \lg\lg n$.

(iii) For optimal efficiency, k should be equal to the smallest integer satisfying

$$\lg n \leqslant \frac{k(k+1)2^{2k}}{2^{k+1} - k - 2}.$$

See [COH 2000] for details. This leads to the following table, which gives for all intervals of bit lengths the appropriate value for k.

k	1	2	3	4	5	6	7
No. of binary digits	$[1, 9]$	$[10, 25]$	$[26, 70]$	$[70, 197]$	$[197, 539]$	$[539, 1434]$	$[1434, 3715]$

Example 9.9 Take $n = 11957708941720303968251$ whose binary representation is

$$(10100010000011101010001100000111111101011001011110111000000001111111111011)_2.$$

As its binary length is 74, take $k = 4$. The representation of n in radix 2^4 is

$$(\underset{2\times1}{2}\ \underset{8\times1}{8}\ \underset{8\times1}{8}\ 3\ \underset{2\times5}{10}\ \underset{8\times1}{8}\ \underset{4\times3}{12}\ 1\ 15\ 13\ \underset{2\times3}{6}\ 5\ \underset{2\times7}{14}\ \underset{2\times7}{14}\ 0\ 1\ 15\ 15\ 11)_{2^4}.$$

Thus the successive values of y are $1, x, x^2, x^4, x^5, x^{10}, x^{20}, x^{40}, x^{80}, x^{81}, x^{162}, x^{324}, \ldots, x^n$. Let us denote a multiplication by M and a squaring by S. Then the precomputations cost $7\mathrm{M} + \mathrm{S}$ and additionally one needs $17\mathrm{M} + 72\mathrm{S}$, i.e., 97 elementary operations in total. By way of comparison, Algorithm 9.1 needs 112 operations, $39\mathrm{M} + 73\mathrm{S}$.

9.1.3 Sliding window method

The 2^k-ary method consists of slicing the binary representation of n into pieces using a window of length k and to process the parts one by one. Letting the window slide allows us to skip strings of consecutive zeroes. For instance, let $n = 334 = (101001110)_2$. Take a window of length 3 or, in other words, precompute x^3, x^5 and x^7 only. The successive values of y computed by Algorithm 9.7 are $1, x^5, x^{10}, x^{20}, x^{40}, x^{41}, x^{82}, x^{164}, x^{167}$ and x^{334} as reflected by

$$334 = (\underset{5}{101}\,\underset{1}{001}\,\underset{2\times3}{110})_2.$$

But one could compute $1, x^5, x^{10}, x^{20}, x^{40}, x^{80}, x^{160}, x^{167}, x^{334}$ instead. This saves one multiplication and amounts to allowing non-adjacent windows

$$334 = (\underset{5}{101}00\underset{7}{111}0)_2$$

where the strings of many consecutive zeroes are ignored.

Here is the general algorithm.

Algorithm 9.10 Sliding window exponentiation

INPUT: An element x of G, a nonnegative integer $n = (n_{\ell-1} \dots n_0)_2$, a parameter $k \geqslant 1$ and the precomputed values $x^3, x^5, \dots, x^{2^k-1}$.
OUTPUT: The element $x^n \in G$.

```
1.  y ← 1 and i ← ℓ − 1
2.  while i ⩾ 0 do
3.      if n_i = 0 then y ← y^2 and i ← i − 1
4.      else
5.          s ← max{i − k + 1, 0}
6.          while n_s = 0 do s ← s + 1
7.          for h = 1 to i − s + 1 do y ← y^2
8.          u ← (n_i ... n_s)_2                    [n_i = n_s = 1 and i − s + 1 ⩽ k]
9.          y ← y × x^u                            [u is odd so that x^u is precomputed]
10.         i ← s − 1
11. return y
```

Remarks 9.11

(i) In Line 6 the index i is fixed, $n_i = 1$ and the while loop finds the longest substring $(n_i \dots n_s)$ of length less than or equal to k such that $n_s = 1$. So $u = (n_i \dots n_s)_2$ is odd and belongs to the set of precomputed values.

(ii) Only the values x^u occurring in Line 9 actually need to be precomputed and not all the values $x^3, x^5, \dots, x^{2^k-1}$.

(iii) In certain cases it is possible to skip some squarings at the beginning, at the cost of an additional multiplication. For the sake of clarity assume that $k = 5$ and that the binary expansion of n is $(1000000)_2$. Then Algorithm 9.10 computes $x, x^2, x^4, x^8, x^{16}, x^{32}, x^{64}$. But one could perform $x^{31} \times x$ instead to obtain x^{32} directly, taking advantage of the precomputations. However, this trick is interesting only if the first value of u is less than 2^{k-1}.

Example 9.12 With $n = 11957708941720303968251$ and $k = 4$ the sliding window method makes use of the following decomposition

$$(\underbrace{101}_{5}000\underbrace{1}_{1}00000\underbrace{111}_{7}0\underbrace{101}_{5}000\underbrace{11}_{3}00000\underbrace{1111}_{15}\underbrace{111}_{7}0\underbrace{1011}_{11}00\underbrace{1011}_{11}\underbrace{1101}_{13}\underbrace{11}_{3}00000000\underbrace{1111}_{15}\underbrace{1111}_{15}\underbrace{1101}_{13}\underbrace{1}_{1})_2.$$

The successive values of y are $1, x^5, x^{10}, x^{20}, x^{40}, x^{80}, x^{81}, x^{162}, x^{324}, x^{648}, \dots, x^n$. In this case 93 operations are needed, namely $21\mathrm{M} + 72\mathrm{S}$, precomputations included.

9.1.4 Signed-digit recoding

When inversion in G is fast (or when x is fixed and x^{-1} precomputed) it can be very efficient to multiply by either x or x^{-1}. This can be used to save additional multiplications on the cost of allowing negative coefficients and hence using the inverse of precomputed values. The extreme

example is the computation of x^{2^k-1}. With the binary method, cf. Algorithm 9.1, one needs $k-1$ squarings and $k-1$ multiplications. But one could also perform k squarings to get x^{2^k} followed by a multiplication by x^{-1}. This remark leads to the following concept.

Definition 9.13 A *signed-digit representation of an integer n in radix b* is given by

$$n = \sum_{i=0}^{\ell-1} n_i b^i \quad \text{with} \quad |n_i| < b.$$

A *signed-binary* representation corresponds to the particular choice $b = 2$ and $n_i \in \{-1, 0, 1\}$.

It is denoted by $(n_{\ell-1} \dots n_0)_{\mathrm{s}}$ and usually obtained by some *recoding* technique. The representation is said to be in *non-adjacent form*, NAF for short, if $n_i n_{i+1} = 0$, for all $i \geqslant 0$. It is denoted by $(n_{\ell-1} \dots n_0)_{\mathrm{NAF}}$.

For example, take $n = 478$ and let $\bar{1} = -1$. Then $(10\bar{1}1100\bar{1}10)_{\mathrm{s}}$ is a signed-binary representation of n. The first recoding technique was proposed by Booth [Boo 1951]. It consists of replacing each string of i consecutive 1 in the binary expansion of n by 1 followed by a string of $i-1$ consecutive 0 and then $\bar{1}$. For $478 = (111011110)_2$ it gives $(100\bar{1}1000\bar{1}0)_{\mathrm{s}}$. Obviously, the signed-binary representation of n is not unique. However, the NAF of a given n is unique and its Hamming weight is minimal among all signed-digit representations of n. For example, the NAF of 478 is equal to $(1000\bar{1}000\bar{1}0)_{\mathrm{NAF}}$. On average the number of nonzero terms in an NAF expansion of length ℓ is equal to $\ell/3$. See [Bos 2001] for a precise analysis of the NAF density. There is a very simple algorithm to compute it [Rei 1962, MoOl 1990].

Algorithm 9.14 NAF representation

INPUT: A positive integer $n = (n_\ell n_{\ell-1} \dots n_0)_2$ with $n_\ell = n_{\ell-1} = 0$.
OUTPUT: The signed-binary representation of n in non-adjacent form $(n'_{\ell-1} \dots n'_0)_{\mathrm{NAF}}$.

1. $c_0 \leftarrow 0$
2. **for** $i = 0$ **to** $\ell - 1$ **do**
3. $\quad c_{i+1} \leftarrow \lfloor (c_i + n_i + n_{i+1})/2 \rfloor$
4. $\quad n'_i \leftarrow c_i + n_i - 2c_{i+1}$
5. **return** $(n'_{\ell-1} \dots n'_0)_{\mathrm{NAF}}$

Remarks 9.15

(i) Algorithm 9.14 subtracts n from $3n$ with the rule $0 - 1 = \bar{1}$ and discards the least significant digit of the result. For each i, c_i is the carry occurring in the addition $n + 2n$. Let $s_i = c_i + n_i + n_{i+1} - 2c_{i+1}$ so that the binary expansion of $3n$ is equal to $(s_{\ell-1} \dots s_0 n_0)_2$. Now $n'_i = c_i + n_i - 2c_{i+1} \in \{\bar{1}, 0, 1\}$. The following observation ensures the non-adjacent property of the expansion [JoYe 2000]. If $n'_i \neq 0$, we have $c_i + n_i = 1$, which implies that $c_{i+1} = n_{i+1}$. So $n'_{i+1} = 2(n_{i+1} - c_{i+2})$ must be zero.

(ii) Finding a signed-binary representation in non-adjacent form can be done by table lookup. Indeed c_{i+1} and n'_i, computed in Lines 3 and 4, only depend on n_{i+1}, n_i and c_i giving just eight cases.

(iii) Algorithm 9.14 operates from the right to the left. Since most of the exponentiation algorithms presented so far process the bits from the left to the right, the signed-binary representation must first be computed and stored. To enable "on the fly" recoding, which

is particularly interesting for hardware applications, cf. Chapter 26, Joye and Yen designed a left-to-right signed-digit recoding algorithm. The result is not necessarily in non-adjacent form but its Hamming weight is still minimal [JOYE 2000].

(iv) Algorithms 9.1, 9.2, 9.7, and 9.10 can be updated in a trivial way to deal with signed-binary representation.

(v) A generalization of the NAF is presented below; see Algorithm 9.20.

Example 9.16 Again take $n = 11957708941720303968251$. Algorithm 9.14 gives

$$n = (10100010000100\bar{1}0101001\bar{0}\bar{1}0000100000000\bar{1}0\bar{1}0\bar{1}010\bar{1}0000\bar{1}00\bar{1}000000010000000000\bar{1}0\bar{1})_{\text{NAF}}.$$

Now one can combine this representation to a sliding window algorithm of length 4 to get the following decomposition

$$(\underset{5}{101}000\underset{1}{1}0000\underset{7}{100\bar{1}}0\underset{5}{101}00\underset{3}{10\bar{1}}0000\underset{1}{1}0000000\underset{-5}{\bar{1}0\bar{1}}0\underset{-3}{\bar{1}01}0\underset{-1}{\bar{1}}0000\underset{-9}{\bar{1}00\bar{1}}0000000\underset{1}{1}0000000000\underset{-5}{\bar{1}0\bar{1}})_{\text{NAF}}.$$

The number of operations, precomputations included, is 90, namely $18\text{M} + 72\text{S}$.

Koyama and Tsuruoka [KOTS 1993] designed another transformation, getting rid of the condition $n_i n_{i+1} = 0$ but still minimizing the Hamming weight. Its average length of zero runs is 1.42 against 1.29 for the NAF.

Algorithm 9.17 Koyama–Tsuruoka signed-binary recoding

INPUT: The binary representation of $n = (n'_{\ell-1} \dots n'_0)_2$.
OUTPUT: The signed-binary representation $(n_\ell \dots n_0)_{\text{s}}$ of n in Koyama–Tsuruoka form.

1. $m \leftarrow 0$, $i \leftarrow 0$, $j \leftarrow 0$, $u \leftarrow 0$, $v \leftarrow 0$, $w \leftarrow 0$, $y \leftarrow 0$ and $z \leftarrow 0$
2. **while** $i < \lfloor \lg n \rfloor$ **do**
3. **if** $n'_i = 1$ **then** $y \leftarrow y + 1$ **else** $y \leftarrow y - 1$
4. $i \leftarrow i + 1$
5. **if** $m = 0$ **then**
6. **if** $y - z \geqslant 3$ **then**
7. **while** $j < w$ **do** $n_j = b_j$ and $j \leftarrow j + 1$
8. $n_j \leftarrow -1$, $j \leftarrow j + 1$, $v \leftarrow y$, $u \leftarrow i$ and $m \leftarrow 1$
9. **else if** $y < z$ **then** $z \leftarrow y$ and $w \leftarrow i$
10. **else**
11. **if** $v - y \geqslant 3$ **then**
12. **while** $j < u$ **do** $n_j = b_j - 1$ and $j \leftarrow j + 1$
13. $n_j \leftarrow 1$, $j \leftarrow j + 1$, $z \leftarrow y$, $w \leftarrow i$ and $m \leftarrow 0$
14. **else if** $y > v$ **then** $v \leftarrow y$ and $u \leftarrow i$
15. **if** $m = 0$ **or** ($m = 1$ **and** $v \leqslant y$) **then**
16. **while** $j < i$ **do** $n_j = b_j - m$ and $j \leftarrow j + 1$
17. $n_j \leftarrow 1 - m$ and $n_{j+1} \leftarrow m$
18. **else**
19. **while** $j < u$ **do** $n_j = b_j - 1$ and $j \leftarrow j + 1$
20. $n_j \leftarrow 1$ and $j \leftarrow j + 1$

21. **while** $j < i$ **do** $n_j = b_j$ and $j \leftarrow j + 1$
22. $n_j \leftarrow 1$ and $n_{j+1} \leftarrow 0$
23. **return** $(n_\ell \dots n_0)_{\mathrm{s}}$

This approach gives good results when combined with the sliding window method.

Example 9.18 For the same $n = 11957708941720303968251$, a sliding window exponentiation of length 4 based on the expansion given by Algorithm 9.17 corresponds to

$$(\underset{5}{101}000\underset{1}{1}0000\underset{1}{1}000\underset{-11}{\bar{1}0\bar{1}\bar{1}}000\underset{3}{11}0000\underset{1}{1}0000000\underset{-5}{\bar{1}0\bar{1}}00\underset{-13}{\bar{1}\bar{1}0\bar{1}}0000\underset{-9}{\bar{1}00\bar{1}}0000000\underset{1}{1}0000000000\underset{-5}{\bar{1}0\bar{1}})_{\mathrm{s}}.$$

In total 89 operations are necessary, i.e., $17\mathrm{M} + 72\mathrm{S}$, including the precomputations.

Now one introduces a generalization of the NAF, which combines window and signed methods as suggested in [MOOL 1990] and explained in [COMI$^+$ 1997, COH 2005].

Definition 9.19 Let w be a parameter greater than 1. Then every positive integer n has a unique signed-digit expansion

$$n = \sum_{i=0}^{\ell-1} n_i 2^i$$

where

- each n_i is zero or odd
- $|n_i| < 2^{w-1}$
- among any w consecutive coefficients at most one is nonzero.

An expansion of this particular form is called *width-w non-adjacent form*, NAF_w for short, and is denoted by $(n_{\ell-1} \dots n_0)_{\mathrm{NAF}_w}$.

In [AVA 2005a], Avanzi shows that the NAF_w is optimal, in the sense that it is a recoding of smallest weight among all those with coefficients smaller in absolute value than 2^{w-1}. See also [MUST 2004] for a similar result.

A generalization of Algorithm 9.14 allows us to compute the NAF_w of any number $n > 0$.

Algorithm 9.20 NAF_w representation

INPUT: A positive integer n and a parameter $w > 1$.
OUTPUT: The NAF_w representation $(n_{\ell-1} \dots n_0)_{\mathrm{NAF}_w}$ of n.

1. $i \leftarrow 0$
2. **while** $n > 0$ **do**
3. **if** n is odd **then**
4. $n_i \leftarrow n \bmod 2^w$
5. $n \leftarrow n - n_i$
6. **else** $n_i \leftarrow 0$
7. $n \leftarrow n/2$ and $i \leftarrow i + 1$
8. **return** $(n_{\ell-1} \dots n_0)_{\mathrm{NAF}_w}$

Remarks 9.21

(i) The function mods used in Line 4 of Algorithm 9.20 returns the smallest residue in absolute value. Hence, $n \bmod 2^w$ belongs to $[-2^{w-1}+1, 2^{w-1}]$.

(ii) For $w = 2$ the NAF_w corresponds to the classical NAF, cf. Definition 9.13.

(iii) The length of the NAF_w of n is at most equal to $\lfloor \lg n \rfloor + 1$. The average density of the NAF_w expansion of n is $1/(w+1)$ as n tends to infinity. For a precise analysis, see [COH 2005].

(iv) A left-to-right variant to compute an NAF_w expansion of an integer can be found both in [AVA 2005a] and in [MUST 2005]. The result may differ from the expansion produced by Algorithm 9.20 but they have the same digit set and the same optimal weight.

(v) Let $w > 1$ and precompute the values $x^{\pm 3}, \ldots, x^{\pm(2^{w-1}-1)}$. Then in Algorithm 9.1 it is sufficient to replace the statement

4. **if** $n_i = 1$ **then** $y \leftarrow x \times y$

by

4. **if** $n_i \neq 0$ **then** $y \leftarrow x^{n_i} \times y$

to take advantage of the NAF_w expansion of $n = (n_{\ell-1} \ldots n_0)_{\text{NAF}_w}$ to compute x^n.

(vi) See [MÖL 2003] for a further generalization called the *signed fractional window method*, where only a subset of $\{x^{\pm 3}, \ldots, x^{\pm(2^{w-1}-1)}\}$ is actually precomputed.

Example 9.22 For $n = 11957708941720303968251$ and $w = 4$ one has

$$n = (5000100000007000500003000010000000\overline{1}000\overline{5}0003000\overline{1}0007000000010000000000005\overline{5})_{\text{NAF}_w}$$

where $\overline{n_i} = -n_i$. With this representation and the modification of Algorithm 9.1 explained above x^n can be obtained with $3\text{M} + \text{S}$ for the precomputations and then $12\text{M} + 69\text{S}$, that is 85 operations in total.

9.1.5 Multi-exponentiation

The group G is assumed to be abelian in this section.

It is often needed in cryptography, for example during a signature verification, cf. Chapter 1, to evaluate $x_0^{n_0} x_1^{n_1}$ where $x_0, x_1 \in G$ and $n_0, n_1 \in \mathbb{Z}$. Instead of computing $x_0^{n_0}$ and $x_1^{n_1}$ separately and then multiplying these terms, it is suggested in [ELG 1985] to adapt Algorithm 9.1 in the following way, in order to get $x_0^{n_0} x_1^{n_1}$ in one round. Indeed, start from $y \leftarrow 1$. Scan the bits of n_0 and n_1 simultaneously from the left to the right and do $y \leftarrow y^2$. Then if the current bits of $\binom{n_0}{n_1}$ are $\binom{1}{0}$, $\binom{0}{1}$ or $\binom{1}{1}$ multiply by x_0, x_1 or $x_0 x_1$ accordingly.

For example, to compute $x_0^{51} x_1^{166}$ write the binary expansion of 51 and 166

$$51 = (00110011)_2$$
$$166 = (10100110)_2$$

and apply the rules above so that the successive values of y are at each step 1, x_1, x_1^2, $x_0 x_1^5$, $x_0^3 x_1^{10}$, $x_0^6 x_1^{20}$, $x_0^{12} x_1^{41}$, $x_0^{25} x_1^{83}$, and finally $x_0^{51} x_1^{166}$.

This trick is often credited to Shamir although it is a special case of an idea of Straus [STR 1964] described below. Note that the binary coefficients of n_j are denoted by $n_{j,k}$. If necessary, the expansion of n_j is padded with zeroes in order to be of length ℓ.

Algorithm 9.23 Multi-exponentiation using Straus' trick

INPUT: The elements $x_0, \ldots, x_{r-1} \in G$ and the ℓ-bit positive exponents $n_0, \ldots, n_{r-1}$. For each $i = (i_{r-1} \ldots i_0)_2 \in [0, 2^r - 1]$, the precomputed value $g_i = \prod_{j=0}^{r-1} x_j^{i_j}$.
OUTPUT: The element $x_0^{n_0} \cdots x_{r-1}^{n_{r-1}}$.

1. $y \leftarrow 1$
2. **for** $k = \ell - 1$ **down to** 0 **do**
3. $\quad y \leftarrow y^2$
4. $\quad i \leftarrow \sum_{j=0}^{r-1} n_{j,k} 2^j$ $\qquad [i = (n_{r-1,k} \ldots n_{0,k})_2]$
5. $\quad y \leftarrow y \times g_i$
6. **return** y

Remarks 9.24

(i) Computing $x_0^{n_0} \ldots x_{r-1}^{n_{r-1}}$ in a naïve way requires $r\ell$ squarings and $r\ell/2$ multiplications on average. With Algorithm 9.23, precomputations cost $2^r - r - 1$ multiplications, then only ℓ squarings and $(1 - 1/2^r)\ell$ multiplications are necessary on average. However one needs to store $2^r - r$ values.

(ii) One can use Algorithm 9.23 to compute x^n. To do so, write $n = (n_{\ell-1} \ldots n_0)_b$ in base b, then set $x_i = x^{b^i}$ and compute $x^n = \prod_{i=0}^{\ell-1} x_i^{n_i}$. This approach can be seen as a baby-step giant-step algorithm, where the giant steps x^{b^i} are computed first.

(iii) All the improvements of Algorithm 9.1 described previously apply to Algorithm 9.23 as well. In particular the use of parallel sliding window leads to a faster method; see [AVA 2002, AVA 2005b, BER 2002] for a general overview on multi-exponentiation.

Example 9.25 Let us compute x^{31021}. One has $31021 = (7\ 36\ 45)_{64}$, so that $x^n = x_0^{45} x_1^{36} x_2^7$ where $x_0 = x$, $x_1 = x^{64}$ and $x_2 = x^{64^2}$. First precompute the g_i's

i	0	1	2	3	4	5	6	7
g_i	1	x_0	x_1	x_0x_1	x_2	x_0x_2	x_1x_2	$x_0x_1x_2$

Then one gets

k	5	4	3	2	1	0
$n_{2,k}$	0	0	0	1	1	1
$n_{1,k}$	1	0	0	1	0	0
$n_{0,k}$	1	0	1	1	0	1
i	3	0	1	7	4	5
y	x_0x_1	$x_0^2x_1^2$	$x_0^5x_1^4$	$x_0^{11}x_1^9x_2$	$x_0^{22}x_1^{18}x_2^3$	$x_0^{45}x_1^{36}x_2^7$

To improve Straus' method in case of a double exponentiation within a group where inversion can be performed efficiently, Solinas [SOL 2001] made signed-binary expansions come back into play.

Definition 9.26 The *joint sparse form*, JSF for short, of the ℓ-bit integers n_0 and n_1 is a representation of the form

$$\begin{pmatrix} n_0 \\ n_1 \end{pmatrix} = \begin{pmatrix} n_{0,\ell} \ldots n_{0,0} \\ n_{1,\ell} \ldots n_{1,0} \end{pmatrix}_{\text{JSF}}$$

such that

- of any three consecutive positions, at least one is a zero column, that is for all i and all positive j one has $n_{i,j+k} = n_{1-i,j+k} = 0$, for at least one k in $\{0, \pm 1\}$
- adjacent terms do not have opposite signs, i.e., it is never the case that $n_{i,j}n_{i,j+1} = -1$
- if $n_{i,j+1}n_{i,j} \neq 0$ then one has $n_{1-i,j+1} = \pm 1$ and $n_{1-i,j} = 0$.

The *joint Hamming weight* is the number of positions different from a zero column.

Solinas also gives an algorithm to compute the JSF of two integers.

Algorithm 9.27 Joint sparse form recoding

INPUT: Nonnegative ℓ-bit integers n_0 and n_1 not both zero.
OUTPUT: The joint sparse form of n_0 and n_1.

1. $j \leftarrow 0$, $S_0 = ()$, $S_1 = ()$, $d_0 \leftarrow 0$ and $d_1 \leftarrow 0$
2. **while** $n_0 + d_0 > 0$ **or** $n_1 + d_1 > 0$ **do**
3. $\quad \ell_0 \leftarrow d_0 + n_0$ and $\ell_1 \leftarrow d_1 + n_1$
4. $\quad$ **for** $i = 0$ **to** 1 **do**
5. $\quad\quad$ **if** $\ell_i \equiv 0 \pmod 2$ **then** $r_i \leftarrow 0$
6. $\quad\quad$ **else**
7. $\quad\quad\quad r_i \leftarrow \ell_i \bmod 4$
8. $\quad\quad\quad$ **if** $\ell_i \equiv \pm 3 \pmod 8$ **and** $\ell_{1-i} \equiv 2 \pmod 4$ **then** $r_i \leftarrow -r_i$
9. $\quad\quad S_i \leftarrow r_i \,\|\, S_i$ [r_i prepended to S_i]
10. $\quad$ **for** $i = 0$ **to** 1 **do**
11. $\quad\quad$ **if** $2d_i = 1 + n'_i$ **then** $d_i \leftarrow 1 - d_i$
12. $\quad\quad n_i \leftarrow \lfloor n_i/2 \rfloor$
13. $\quad j \leftarrow j + 1$
14. **return** S_0 and S_1

Remarks 9.28

(i) The joint sparse form of n_0 and n_1 is unique. The joint Hamming weight of the JSF of n_0 and n_1 is equal to $\ell/2$ on average and the JSF is optimal in the sense that it has the smallest joint Hamming weight among all joint signed representations of n_0 and n_1 [SOL 2001].

(ii) The naïve computation of $x_0^{n_0}x_1^{n_1}$ involving NAF representations of n_0 and n_1 requires 2ℓ squarings and $2\ell/3$ multiplications on average. Only ℓ squarings and $\ell/2$ multiplications are necessary with the JSF, neglecting the cost of the precomputations of x_0x_1 and x_0/x_1. Applying Straus' trick with two integers in NAF results in a Hamming weight of $5\ell/9$ on average.

(iii) In [GRHE$^+$ 2004], Grabner et al. introduce the *simple joint sparse form* whose joint Hamming weight is also minimal but which can be obtained in an easier way.

Example 9.29 Let us compute $x_0^{51}x_1^{166}$. The joint NAF expansion of 51 and 166 is

$$\begin{aligned} 51 &= (010\bar{1}010\bar{1})_{\text{NAF}} \\ 166 &= (10101010\bar{1}0)_{\text{NAF}}. \end{aligned}$$

Its joint Hamming weight is 8. The JSF of 51 and 166, as given by Algorithm 9.27, is

$$\begin{pmatrix} 51 \\ 166 \end{pmatrix} = \begin{pmatrix} 0010\bar{1}0011 \\ 10\bar{1}0\bar{1}\bar{1}0\bar{1}0 \end{pmatrix}_{\text{JSF}}$$

with a joint Hamming weight equal to 6.

The next section is devoted to the case when several exponentiations to the same exponent n have to be performed.

9.2 Fixed exponent

The methods considered in this section essentially give better algorithms when the exponent n is fixed. They rely on the concept of addition chains. However, the computation of a short addition chain for a given exponent can be very costly. But if the exponent is to be used several times it is probably a good investment to carry out this search.

In the following, different kinds of addition chains are discussed, then efficient methods to actually find short addition chains are introduced before related exponentiation algorithms are described.

9.2.1 Introduction to addition chains

Definition 9.30 An *addition chain* computing an integer n is given by two sequences v and w such that

$$\begin{aligned} v &= (v_0, \ldots, v_s),\ v_0 = 1,\ v_s = n \\ v_i &= v_j + v_k \ \text{ for all } \ 1 \leqslant i \leqslant s \ \text{ with respect to} \\ w &= (w_1, \ldots, w_s),\ w_i = (j,k) \quad \text{and} \quad 0 \leqslant j,k \leqslant i-1. \end{aligned} \tag{9.1}$$

The *length* of the addition chain is s.

A *star addition chain* satisfies the additional property that at each step $v_i = v_{i-1} + v_k$ for some k such that $0 \leqslant k \leqslant i-1$.

Note that one should write $v_i = v_{j(i)} + v_{k(i)}$ since the indexes depend on i. They are omitted for the sake of simplicity. Sometimes only v is given since it is easy to retrieve w from v. For example $v = (1, 2, 3, 6, 7, 14, 15)$ is an addition chain for 15 of length 6. It is implicit in the computation of x^{15} by Algorithm 9.1. In fact binary or window methods can be seen as methods producing and using special classes of addition chains but it is often possible to do better, that is to find a shorter chain. For instance $(1, 2, 3, 6, 12, 15)$ computes also 15 and is of length 5.

For a given n, the smallest s for which there exists an addition chain of length s computing n is denoted by $\ell(n)$. It is not hard to see that $\ell(15) = 5$ but the determination of $\ell(n)$ can be a difficult problem even for rather small n.

As complexities of squarings and multiplications are usually slightly different, note that a carefully chosen complexity measure should take into account not only the length of the chain but also the respective numbers of squarings and multiplications involved. For example $(1, 2, 4, 5, 6, 11)$ and

$(1,2,3,4,8,11)$ compute 11 and have the same length 5. However the first chain needs 3 multiplications whereas the latter requires only 2.

There is an abundant literature about addition chains. It is known [SCH 1975, BRA 1939] that

$$\lg(n) + \lg\bigl(\nu(n)\bigr) - 2.13 \leqslant \ell(n) \leqslant \lg(n) + \lg(n)\bigl(1 + o(1)\bigr)/\lg\bigl(\lg(n)\bigr),$$

where $\nu(n)$ is the Hamming weight of n.

As said before, finding an addition chain of the shortest length can be very hard. To make this process easier, it seems harmless to restrict the search to star addition chains. But Hansen [HAN 1959] proved that for some n, the smallest being $n = 12509$, there is no star addition chain of minimal length $\ell(n)$. The shortest length $\ell(n)$ has been determined for all n up to 2^{20}, pruning trees to speed up the search [BLFL]. See also Thurber's algorithm, which is able to find all the addition chains for a given n [THU 1999]. The hardness of this search depends primarily on $\nu(n)$, so that it is longer to find the minimal length of $191 = (10111111)_2$ than $1048577 = (10000000000000000001)_2$, but the running time can be quite long, even for small integers with a rather low density.

The concept of addition chain can be extended in at least three different ways.

Definition 9.31 An *addition-subtraction chain* is similar to an addition chain except that the condition $v_i = v_j + v_k$ is replaced by $v_i = v_j + v_k$ or $v_i = v_j - v_k$.

For example, an addition chain for 314 is $v = (1,2,4,8,9,19,38,39,78,156,157,314)$. The addition–subtraction chain $v = (1,2,4,5,10,20,40,39,78,156,157,314)$ is one term shorter.

Definition 9.32 An *addition sequence* for the set of integers $S = \{n_0,\ldots,n_{r-1}\}$ is an addition chain v that contains each element of S. In other words, for all i there is j such that $n_i = v_j$.

For example, an addition sequence computing $\{47,117,343,499,933,5689\}$ is

$(1,2,4,8,10,11,18,36,\underline{47},55,91,109,\underline{117},226,\underline{343},434,489,\underline{499},\underline{933},1422,2844,5688,\underline{5689})$.

In [YAO 1976], it is shown that the shortest length of an addition sequence computing the set of integers $\{n_0,\ldots,n_{r-1}\}$ is less than

$$\lg N + cr\lg N/\lg\lg N,$$

where $N = \max_i\{n_i\}$ and c is some constant.

Definition 9.33 Let k and s be positive integers. A *vectorial addition chain* is a sequence V of k-dimensional vectors of nonnegative integers v_i for $-k+1 \leqslant i \leqslant s$ together with a sequence w, such that

$$\begin{aligned}
v_{-k+1} &= [1,0,0,\ldots,0,0]\\
v_{-k+2} &= [0,1,0,\ldots,0,0]\\
&\vdots\\
v_0 &= [0,0,0,\ldots,0,1]\\
v_i &= v_j + v_k \text{ for all } 1 \leqslant i \leqslant s \text{ with } -k+1 \leqslant j,k \leqslant i-1\\
v_s &= [n_0,\ldots,n_{r-1}]\\
w &= (w_1,\ldots,w_s),\ w_i = (j,k).
\end{aligned} \tag{9.2}$$

For example, a vectorial addition chain for $[45,36,7]$ is

$$\begin{aligned}
V = \bigl([1,0,0],[0,1,0],[0,0,1],[1,1,0],[2,2,0],[4,4,0],[5,4,0],[10,8,0],\\
[11,9,0],[11,9,1],[22,18,2],[22,18,3],[44,36,6],[45,36,6],[45,36,7]\bigr)
\end{aligned}$$

$$w \;=\; \big((-2,-1),(1,1),(2,2),(-2,3),(4,4),(1,5),(0,6),(7,7),(0,8),(9,9),(-2,10),(0,11)\big).$$

Since $k=3$, the first three terms of V are $v_{-2}=[1,0,0]$, $v_{-1}=[0,1,0]$, and $v_0=[0,0,1]$. This chain is of length 12 and is implicitly produced by Algorithm 9.23.

Addition sequences and vectorial addition chains are in some sense dual. We refer the interested reader to [BER 2002, STA 2003] for details and to [KNPA 1981] for a more general approach. In [OLI 1981] Olivos describes a procedure to transform an addition sequence computing $\{n_0,\ldots,n_{r-1}\}$ of length ℓ into a vectorial addition chain of length $\ell+r-1$ for $[n_0,\ldots,n_{r-1}]$.

To illustrate his method let us deduce a vectorial addition chain for $[45,36,7]$ from the addition sequence $v=(1,2,4,6,7,9,18,36,45)$ computing $\{7,36,45\}$. Let $\{e_j \mid 0 \leqslant j \leqslant k\}$ be the canonical basis of $\mathbb{R}^{k+1}$. The idea is then to build an array by induction, starting in the lower right corner with a 2-by-2 array, and then processing the successive elements v_h of the addition sequence, following two rules:

- if $v_h = 2v_i$ then the line to be added on top is the double of line i and the new two columns on the left are $2e_h$ and $2e_h+e_i$
- if v_h satisfies $v_h=v_i+v_j$ then the new line on top is the sum of lines i and j and the two columns on the left are e_h+e_i and e_h+e_j.

The expression of v_h in terms of the v_i's is written on the right. The first steps are:

$$\left|\begin{array}{cc|} 2&2\\ 0&1 \end{array}\right. \quad 2=1+1 \quad\Longrightarrow\quad \left|\begin{array}{cc|} 2&2\\ 0&1 \end{array}\right. \left|\begin{array}{cc|} 4&4\\ 2&2\\ 0&1 \end{array}\right. \begin{array}{l} 4=2+2\\ 2=1+1\\ 1 \end{array} \quad\Longrightarrow\quad \left|\begin{array}{cc|} 1&1\\ 1&0\\ 0&1\\ 0&0 \end{array}\right. \left|\begin{array}{cc|} 2&3\\ 2&2\\ 0&1\\ 0&0 \end{array}\right. \left|\begin{array}{cc|} 6&6\\ 4&4\\ 2&2\\ 0&1 \end{array}\right. \begin{array}{l} 6=4+2\\ 4=2+2\\ 2=1+1\\ 1 \end{array}$$

At the end one has

$$\left|\begin{array}{cc|} 1&1\\ 1&0\\ 0&0\\ 0&1\\ 0&0\\ 0&0\\ 0&0\\ 0&0\\ 0&0 \end{array}\right. \left|\begin{array}{cc|} 2&2\\ 2&2\\ 0&1\\ 0&0\\ 0&0\\ 0&0\\ 0&0\\ 0&0\\ 0&0 \end{array}\right. \left|\begin{array}{cc|} 4&5\\ 4&4\\ 2&2\\ 1&1\\ 1&0\\ 0&0\\ 0&0\\ 0&1\\ 0&0 \end{array}\right. \left|\begin{array}{cc|} 5&5\\ 4&4\\ 2&2\\ 1&1\\ 1&0\\ 0&0\\ 0&0\\ 0&1\\ 0&0 \end{array}\right. \left|\begin{array}{cc|} 5&5\\ 4&4\\ 2&2\\ 1&1\\ 1&0\\ 0&0\\ 0&0\\ 0&1\\ 0&0 \end{array}\right. \left|\begin{array}{cc|} 5&10\\ 4&8\\ 2&4\\ 1&2\\ 1&1\\ 1&1\\ 0&0\\ 0&1\\ 0&0 \end{array}\right. \left|\begin{array}{cc|} 10&20\\ 8&16\\ 4&8\\ 2&4\\ 1&2\\ 1&1\\ 0&0\\ 0&1\\ 0&0 \end{array}\right. \left|\begin{array}{cc|} 40&45\\ 32&36\\ 16&18\\ 8&9\\ 4&7\\ 2&6\\ 2&4\\ 1&2\\ 1&1 \end{array}\right. \begin{array}{ll} 45=36+9 & \leftarrow\\ 36=18+18 & \leftarrow\\ 18=9+9 & \\ 9=7+2 & \\ 7=6+1 & \leftarrow\\ 6=4+2 & \\ 4=2+2 & \\ 2=1+1 & \\ 1 & \end{array}$$

Then discard all the lines except the ones marked by an arrow and corresponding to 7, 36, and 45. Consider the columns from the left to the right, eliminate redundancies and finally add the canonical vectors of $\mathbb{R}^r$ so that a vectorial addition sequence for $[45,36,7]$ is

$$\big([1,0,0],[0,1,0],[0,0,1],[1,1,0],[2,2,0],[4,4,0],[5,4,0],$$
$$[5,4,1],[10,8,1],[10,8,2],[20,16,3],[40,32,6],[45,36,7]\big).$$

Conversely the procedure to get an addition sequence from a vectorial addition exists as well [OLI 1981].

Before explaining how to find short chains, let us remark that the set of vectors $(n_0,\ldots,n_{r-1},\ell)$ such that there is an addition sequence of length ℓ containing $n_0,\ldots,n_{r-1}$ has been shown to be **NP**-complete [DOLE$^+$ 1981]. This does not imply, as it is sometimes claimed, that finding a shortest addition chain for n is **NP**-complete. However, we have seen that dedicated algorithms to find a shortest addition chain are in practice limited to small exponents.

9.2.2 Short addition chains search

In the following, different heuristics to find short addition chains are discussed. They are rather efficient but do not necessarily find a shortest possible chain. Most of the methods described here use the concept of dictionary.

Definition 9.34 Given an integer n, a *dictionary* $\mathcal{D}$ for n is a set of integers such that

$$n = \sum_{i=0}^{k} b_{i,d} d 2^i, \text{with } b_{i,d} \in \{0, 1\} \text{ and } d \in \mathcal{D}.$$

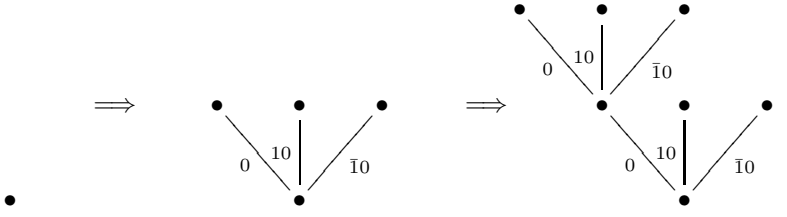
Note that all the algorithms introduced in Section 9.1 can be used to produce addition chains and implicitly use a dictionary. For example, the dictionary associated to window methods of length k is the set $\{1, 3, \ldots, 2^k - 1\}$. For the NAF$_w$ it is $\{\pm 1, \pm 3, \ldots, \pm (2^{k-1} - 1)\}$.

In [GOHA$^+$ 1996, O'CO 2001] the dictionary is simply made of the elements $2^i - 1$ for $0 < i \leqslant w$, for some fixed parameter w.

The power tree method [KNU 1997] is quite simple to implement but it does not always return an optimal addition chain, the first counter example being $n = 77$. Like other algorithms exploring trees it cannot be used for exponents of cryptographic relevance, as the size grows too fast in the bit size of the exponent and is too large for the required sizes.

A more sophisticated method is described in [KUYA 1998] and is related to the Tunstall method; see [TUN 1968]. Namely, choose a parameter k, let p be the number of zeroes in the expansion of n divided by its length ℓ and let $q = 1 - p$. If the expansion is signed let $\hat{q} = (1 - p)/2$. Then form a tree having a root of weight 1 and while the number of leaves is less than $k + 1$ add leaves to this tree according to the following procedure. Take the leaf of highest weight w and create two children with weight wp and wq, labeled respectively by 0 and 1. If the expansion is signed create three children with weight wp, $w\hat{q}$, and $w\hat{q}$, labeled respectively by 0, $\bar{1}0$, and 10, instead. At the end read the labels from the root to the leaves and concatenate 1 (10 if signed) at the beginning of each sequence. The dictionary $\mathcal{D}$ is the set of odd integers obtained by removing the zeroes at the end of each sequence. The result is a function of ℓ and of the number of zeroes in the signed-binary expansion of n. The best choice for the size of the dictionary depends on ℓ and can be as large as 20 for 512-bit exponents [KUYA 1998].

Example 9.35 Take $n = 587257$ and $k = 4$. The signed-binary recoding Algorithm 9.14 gives $n = (1001000010\bar{1}0\bar{1}000001\bar{1}001)_{\text{NAF}}$. One has $p = 7/10$ and $\hat{q} = 3/20$. After two iterations, there are five leaves and $\mathcal{D} = \{(00), (010), (0\bar{1}0), (10), (\bar{1}0)\}$ as shown below



Then concatenate (10) at the beginning of each sequence of $\mathcal{D}$ and remove all the zeroes at the end to finally get $\mathcal{D} = \{1, 3, 5, 7, 9\}$. From this one can compute $n = 2^{19} + 2^{16} - 5 \times 2^9 - 7$.

Yacobi suggests a completely different approach [YAC 1998], namely to use the well-known Lempel–Ziv compression algorithm [ZILE 1977, LEZI 1978] to get the dictionary.

At the beginning the dictionary is empty and new elements are added while the binary expansion of the exponent is scanned from the right to the left. Take the longest word in the dictionary that

fits as prefix of the unscanned part of the exponent and concatenate it to the next unscanned digit of the exponent to form a new element of the dictionary. Repeat the process until all the digits are processed. There is also a sliding version that skips strings of zeroes.

Example 9.36 For $n = 587257$ one gets $(10\underline{00}\,\underline{1111}\,\underline{010}\,\underline{111}\,\underline{11}\,\underline{10}\,\underline{0}\,\underline{1})_2$ and the dictionary $\{1, 0, 2, 3, 7, 2, 15, 0, 2\}$, which actually gives rise to $\mathcal{D} = \{1, 3, 7, 15\}$. One has $n = 1 \times 2^{19} + 15 \times 2^{12} + 1 \times 2^{10} + 7 \times 2^6 + 3 \times 2^4 + 1 \times 2^3 + 1$ so that an addition chain for n is

$$(1, 2, 3, 4, 7, 8, 15, 16, 32, 64, 128, 143, 286, 572, 573, 1146, 2292, 4584,\\ 9168, 9175, 18350, 36700, 36703, 73406, 73407, 146814, 293628, 587256, 587257)$$

whose length is 28.

The sliding version returns $\mathcal{D} = \{1, 3, 5, 7\}$ from the decomposition

$$(1000\,\underline{0111}\,\underline{101}\,\underline{01}\,\underline{111}\,\underline{11}00\,\underline{1})_2.$$

In this case $n = 1 \times 2^{19} + 7 \times 2^{13} + 5 \times 2^{10} + 1 \times 2^8 + 7 \times 2^6 + 3 \times 2^3 + 1$ and an addition chain for n is

$$(1, 2, 3, 5, 7, 8, 16, 32, 64, 71, 142, 284, 568, 573, 1146, 2292, 2293, 4586,\\ 9172, 18344, 18351, 36702, 73404, 73407, 146814, 293628, 587256, 587257)$$

of length 27.

This method can also be used with signed-digit representations and is particularly efficient when the number of zeroes is small.

In [BEBE$^+$ 1989] continued fractions and the Euclid algorithm are used to produce short addition chains. First let $\otimes$ and $\oplus$ be two simple operations on addition chains, defined as follows. If $v = (v_0, \ldots, v_s)$ and $w = (w_0, \ldots, w_t)$ then

$$v \otimes w = (v_0, \ldots, v_s, v_s w_0, \ldots, v_s w_t)$$

and if j is an integer

$$v \oplus j = (v_0, \ldots, v_s, v_s + j).$$

Now let $1 < k < n$ be some integer. Then

$$(1, \ldots, k, \ldots, n) = (1, \ldots, n \bmod k, \ldots, k) \otimes (1, \ldots, \lfloor n/k \rfloor) \oplus (n \bmod k). \tag{9.3}$$

The point is to choose the best possible k. If $k = \lfloor n/2 \rfloor$ then the addition chain is equal to the one obtained with binary methods. Instead the authors propose a *dichotomic strategy*, that is to take

$$k = \gamma(n) = \left\lfloor \frac{n}{2^{\lceil \lfloor \lg n \rfloor / 2 \rceil}} \right\rfloor.$$

The rule (9.3) is then applied recursively in `minchain`(n), which uses the additional procedure `chain`(n, k),

```
minchain(n)
1. if n = 2^ℓ then return (1, 2, 4, ..., 2^ℓ)
2. if n = 3 then return (1, 2, 3)
3. return chain(n, 2^⌊lg n/2⌋)
```

and

```
chain(n, k)
1. q ← ⌊n/k⌋ and r ← n mod k
2. if r = 0 then return (minchain(k) ⊗ minchain(q))
3. else return chain(k, r) ⊗ minchain(q) ⊕ r
```

Note that these algorithms are able to find short addition sequences as well.

Example 9.37 For $n = 87$, one has $k = \lfloor 87/8 \rfloor = 10$ and the successive calls are

$$\begin{aligned}
&\texttt{chain}(87, 10)\\
&\texttt{chain}(10, 7) \otimes \texttt{minchain}(8) \oplus 7\\
&(\texttt{chain}(7, 3) \otimes \texttt{minchain}(1) \oplus 3) \otimes \texttt{minchain}(8) \oplus 7\\
&\big((\texttt{minchain}(3) \otimes \texttt{minchain}(2) \oplus 1) \otimes \texttt{minchain}(1) \oplus 3\big) \otimes \texttt{minchain}(8) \oplus 7
\end{aligned}$$

so that the final result is the optimal addition chain $(1, 2, 3, 6, 7, 10, 20, 40, 80, 87)$.

In [BEBE$^+$ 1994], the authors generalize this approach, introducing new strategies to determine a set of possible values for k. So the choice of k is no longer deterministic and it is necessary to backtrack the best possible k. For the *factor method*, see also [KNU 1997], one has

$$\begin{cases} k \in \{n-1\} & \text{if } n \text{ is prime} \\ k \in \{n-1, p\} & \text{if } p \text{ is the smallest prime dividing } n. \end{cases}$$

For the *total strategy*, $k \in \{2, 3, \ldots, n-1\}$. For the *dyadic strategy*,

$$k \in \left\{ \left\lfloor \frac{n}{2^j} \right\rfloor, j = 1, \ldots \right\}.$$

Note that only *Fermat's strategy*, where

$$k \in \left\{ \left\lfloor \frac{n}{2^{2^j}} \right\rfloor, j = 0, 1, \ldots \right\}$$

has a reasonable complexity and is well suited for large exponents.

Example 9.38 The corresponding results for 87 are all optimal and given in the next table.

Strategy	Initial k	Result
Factor	3	$(1, 2, 3, 6, 12, 24, 48, 72, 84, 87)$
Total	17	$(1, 2, 4, 8, 16, 17, 34, 68, 85, 87)$
Dyadic	2	$(1, 2, 4, 6, 10, 20, 40, 80, 86, 87)$
Fermat	5	$(1, 2, 3, 5, 10, 20, 40, 80, 85, 87)$

In [BOCO 1990] Bos and Coster use similar ideas to produce an addition sequence. See also [COSTER]. Starting from $1, 2$ and the requested numbers $\ldots, f_2, f_1, f$ they replace at each step the last term by new elements produced by one of four different methods. A weight function helps

to decide which rule should be used at each stage. Here is a brief description of these strategies with some examples.

Approximation

Condition $0 \leqslant f - (f_i + f_j) = \varepsilon$ with $f_i \leqslant f_j$ and ε small
Insert $f_i + \varepsilon$
Example `49 67 85 117` → `49 50 67 85 (because 117 - (49 + 67) = 1)`

Division

Condition f is divisible by $p \in \{3, 5, 9, 17\}$. Let $(1, 2, \ldots, \alpha_r = p)$ be an addition chain for p
Insert $f/p, 2f/p, \ldots, \alpha_{r-1}f/p$
Example `17 48` → `16 17 32 (because 48/3 = 16 and (1,2,3) computes 3)`

Halving

Condition $f/f_i \geqslant 2^u$ and $\lfloor f/2^u \rfloor = k$
Insert $k, 2k, \ldots, k2^u$
Example `14 382` → `14 23 46 92 184 368 (because 382/14` $\geqslant 2^4$ `and` $\lfloor 382/2^4 \rfloor$ `= 23)`

Lucas

Condition f and f_i belong to a Lucas series ($f_i = u_0$, $f = u_k$, $k \geqslant 3$ and $u_{i+1} = u_i + u_{i-1}$)
Insert $u_1, u_2, \ldots, u_{k-1}$
Example `4 23` → `4 5 9 14 (because 4,5,9,14,23 is a Lucas series)`

For faster results use only **Approximation** and **Halving** steps. The choice is simpler and does not require any weight function.

Example 9.39 This method applied to $\{1, 2, 47, 117, 343, 499, 933, 5689\}$ returns

$$(1, 2, 4, 8, 10, 11, 18, 36, \underline{47}, 55, 91, 109, \underline{117}, 226, \underline{343}, 434, 489, \underline{499}, \underline{933}, 1422, 2844, 5688, \underline{5689}).$$

The method of Bos and Coster when combined to a sliding window of big length allows us to compute x^n with a dictionary of small size and no precomputation. The following example is taken from [BOCO 1990].

Example 9.40 Let $n = 26235947428953663183191$ and take a window of length 10 (except for the first digit corresponding to a window of length 13). Then

$$n = (\underbrace{1011000111001}_{5689}0000001\underbrace{1101001001}_{933}00\underbrace{1110101}_{117}000000\underbrace{101111}_{47}00000\underbrace{111110011}_{499}00\underbrace{101010111}_{343})_2$$

so that the dictionary is $\mathcal{D} = \{47, 117, 343, 499, 933, 5689\}$ and the corresponding addition chain built with $\mathcal{D}$ is of length 89. By way of comparison, Algorithms 9.1 and 9.10 need respectively 110 and 93 operations.

Finally, see [NEMA 2002] for techniques related to genetic algorithms.

9.2.3 Exponentiation using addition chains

Once an addition chain for n is found it is straightforward to deduce x^n.

Algorithm 9.41 Exponentiation using addition chain

INPUT: An element x of G and an addition chain computing n i.e., v and w as in (9.1).
OUTPUT: The element x^n.

1. $x_1 \leftarrow x$
2. **for** $i = 1$ **to** s **do** $x_i \leftarrow x_j \times x_k$ $[w(i) = (j, k)]$
3. **return** x_s

Example 9.42 Let us compute x^{314}, from the addition chain for 314 given below

i	0	1	2	3	4	5	6	7	8	9	10	11	12
v_i	1	2	4	8	9	18	19	38	39	78	156	157	314
w_i	—	$(0,0)$	$(1,1)$	$(2,2)$	$(3,0)$	$(4,4)$	$(5,0)$	$(6,6)$	$(7,0)$	$(8,8)$	$(9,9)$	$(10,0)$	$(11,11)$
x_i	x	x^2	x^4	x^8	x^9	x^{18}	x^{19}	x^{38}	x^{39}	x^{78}	x^{156}	x^{157}	x^{314}

Vectorial addition chains are well suited to multi-exponentiation. Here again G is assumed to be abelian.

Algorithm 9.43 Multi-exponentiation using vectorial addition chain

INPUT: Elements $x_0, \ldots, x_{r-1}$ of G and a vectorial addition chain of dimension r computing $[n_0, \ldots, n_{r-1}]$ as in (9.2).
OUTPUT: The element $x_0^{n_0} \cdots x_{r-1}^{n_{r-1}}$.

1. **for** $i = -k + 1$ **to** 0 **do** $y_i \leftarrow x_{i+k-1}$
2. **for** $i = 1$ **to** s **do** $y_i \leftarrow y_j \times y_k$ $[w(i) = (j, k)]$
3. **return** y_s

The vectorial addition chain for $[45, 36, 7]$ implicitly produced by Algorithm 9.23 is of length 12. A careful search reveals a chain of length 10 as it can be seen in the next table, which displays the execution of Algorithm 9.43 while computing $x_0^{45}x_1^{36}x_2^7$ with it. Recall that $y_{-2} = x_0$, $y_{-1} = x_1$ and $y_0 = x_2$.

i	1	2	3	4	5	6	7	8	9	10
w_i	$(-2,-1)$	$(1,1)$	$(2,2)$	$(-2,3)$	$(0,4)$	$(4,5)$	$(0,6)$	$(6,7)$	$(8,8)$	$(5,9)$
y_i	x_0x_1	$x_0^2x_1^2$	$x_0^4x_1^4$	$x_0^5x_1^4$	$x_0^5x_1^4x_2$	$x_0^{10}x_1^8x_2$	$x_0^{10}x_1^8x_2^2$	$x_0^{20}x_1^{16}x_2^3$	$x_0^{40}x_1^{32}x_2^6$	$x_0^{45}x_1^{36}x_2^7$

9.3 Fixed base point

In some situations the element x is always the same whereas the exponent varies. Precomputations are the key point here.

9.3.1 Yao's method

A simpler version of Algorithm 9.44, which can be seen as the dual of the 2^k-ary method, was first described in [YAO 1976]. A slightly improved form is presented in [KNU 1981, answer to exercise 9, Section 4.6.3]. Note that it is identical to the patented BGMW's algorithm [BRGO$^+$ 1993].

Let n, n_i, b_i, ℓ and h be integers. Suppose that

$$n = \sum_{i=0}^{\ell-1} n_i b_i \text{ with } 0 \leqslant n_i < h \text{ for all } i \in [0, \ell-1]. \tag{9.4}$$

Let $x_i = x^{b_i}$. The method relies on the equality

$$x^n = \prod_{i=0}^{\ell-1} x_i^{n_i} = \prod_{j=1}^{h-1} \left[\prod_{n_i=j} x_i \right]^j.$$

Algorithm 9.44 Improved Yao's exponentiation

INPUT: The element x of G, an exponent n written as in (9.4) and the precomputed values $x^{b_0}, x^{b_1}, \ldots, x^{b_{\ell-1}}$.
OUTPUT: The element x^n.

1. $y \leftarrow 1, u \leftarrow 1$ and $j \leftarrow h-1$
2. **while** $j \geqslant 1$ **do**
3. **for** $i = 0$ **to** $\ell - 1$ **do if** $n_i = j$ **then** $u \leftarrow u \times x^{b_i}$
4. $y \leftarrow y \times u$
5. $j \leftarrow j - 1$
6. **return** y

Remarks 9.45

(i) The term $\left[\prod_{n_i=j} x_i \right]^j$ is computed by repeated iterations in Line 4.
One obtains the correct powers $x_i^{n_i}$ as in each round the result is multiplied with u and x_i is included in u for n_i rounds.

(ii) The choice of h and of the b_i's is free. One can set $h = 2^k$ and $b_i = h^i$ so that the n_i's are simply the digits of n in base h.

(iii) One needs $\ell + h - 2$ multiplications and $\ell + 1$ elements must be stored to compute x^n.

Example 9.46 Let us compute x^{2989}. Set $h = 4$, $b_i = 4^i$ then $2989 = (2\,3\,2\,2\,3\,1)_4$ and $\ell = 6$. Suppose that $x, x^4, x^{16}, x^{64}, x^{256}$ and x^{1024} are precomputed and stored.

j	3	2	1
u	$x^4 x^{256} = x^{260}$	$x^{260} x^{16} x^{64} x^{1024} = x^{1364}$	$x^{1364} x = x^{1365}$
y	x^{260}	$x^{260} x^{1364} = x^{1624}$	$x^{1624} x^{1365} = x^{2989}$

9.3.2 Euclidean method

The Euclidean method was first introduced in [ROO 1995], see also [SEM 1983]. Algorithm 9.47 computes x^n generalizing a method to compute the double exponentiation $x_0^{n_0}x_1^{n_1}$ discussed by Bergeron et al. [BEBE$^+$ 1989] which is similar to the technique introduced in Section 9.2.2 to find short addition chains. The idea is to recursively use the equality

$$x_0^{n_0}x_1^{n_1} = (x_0x_1^q)^{n_0} \times x_1^{(n_1 \bmod n_0)} \text{ where } q = \lfloor n_1/n_0 \rfloor.$$

Algorithm 9.47 Euclidean exponentiation

INPUT: The element x of G, an exponent n as in (9.4) and the precomputed values $x_0 = x^{b_0}, x_1 = x^{b_1}, \ldots, x_{\ell-1} = x^{b_{\ell-1}}$.
OUTPUT: The element x^n.

1. **while true do**
2. Find M such that $n_M \geqslant n_i$ for all $i \in [0, \ell-1]$
3. Find $N \neq M$ such that $n_N \geqslant n_i$ for all $i \in [0, \ell-1],\ i \neq M$
4. **if** $n_N \neq 0$ **then**
5. $q \leftarrow \lfloor n_M/n_N \rfloor$, $x_N \leftarrow {x_M}^q \times x_N$ and $n_M \leftarrow n_M \bmod n_N$
6. **else break**
7. **return** $x_M^{n_M}$

Example 9.48 Take the same exponent $2989 = (2\,3\,2\,2\,3\,1)_4$ and let us evaluate x^{2989}.

n_5	n_4	n_3	n_2	n_1	n_0	M	N	q	x_5	x_4	x_3	x_2	x_1	x_0
—	—	—	—	—	—	—	—	—	x^{1024}	x^{256}	x^{64}	x^{16}	x^4	x
2	3	2	2	3	1	4	1	1	x^{1024}	x^{256}	x^{64}	x^{16}	x^{260}	x
2	0	2	2	3	1	1	2	1	x^{1024}	x^{256}	x^{64}	x^{276}	x^{260}	x
2	0	2	2	1	1	5	3	1	x^{1024}	x^{256}	x^{1088}	x^{276}	x^{260}	x
0	0	2	2	1	1	3	2	1	x^{1024}	x^{256}	x^{1088}	x^{1364}	x^{260}	x
0	0	0	2	1	1	2	1	2	x^{1024}	x^{256}	x^{1088}	x^{1364}	x^{2988}	x
0	0	0	0	1	1	1	0	1	x^{1024}	x^{256}	x^{1088}	x^{1364}	x^{2988}	x^{2989}
0	0	0	0	0	1	0	1	—	x^{1024}	x^{256}	x^{1088}	x^{1364}	x^{2988}	x^{2989}

9.3.3 Fixed-base comb method

This algorithm is a special case of Pippenger's algorithm [BER 2002, PIP 1979, PIP 1980]. It is also often referred to as Lim–Lee method [LILE 1994]. It is essentially a special case of Algorithm 9.23 where the different base points are in fact distinct powers of a single base. Suppose that $n = (n_{\ell-1} \ldots n_0)_2$. Select an integer $h \in [1, \ell]$. Let $a = \lceil \ell/h \rceil$ and choose $v \in [1, a]$. Let $r = \lceil a/v \rceil$ and write the n_i's in an array with h rows and a columns as below (pad the representation of n with zeroes if necessary)

$a-1$	$\cdots$	1	0
n_{a-1}	$\cdots$	n_1	n_0
n_{2a-1}	$\cdots$	n_{a+1}	n_a
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n_{ah-1}	$\cdots$	n_{ah-a+1}	n_{ah-a}

For each s, the column number s can be read as the binary representation of an integer denoted by $I(s)$. For example the last column $I(0)$ is the binary representation of $I(0) = (n_{ah-a} \dots n_a n_0)_2$. The algorithm relies on the following equality

$$x^n = \prod_{k=0}^{r-1} \left(\prod_{j=0}^{v-1} G[j, I(jr+k)] \right)^{2^k}$$

where

$$G[j,i] = \left(\prod_{s=0}^{h-1} x^{i_s 2^{as}} \right)^{2^{jr}} \quad \text{for } j \in [0, v-1] \text{ and } i = (i_{h-1} \dots i_0)_2 \in [0, 2^h - 1].$$

Algorithm 9.49 Fixed-base comb exponentiation

INPUT: The element x of G and an exponent n. Let h, a, v, r and $G[j,i]$ be as above.
OUTPUT: The element x^n.

1. $y \leftarrow 1$ and $k \leftarrow r-1$
2. **for** $k = r-1$ **down to** 0 **do**
3. $\quad y \leftarrow y^2$
4. $\quad$ **for** $j = v-1$ **down to** 0 **do**
5. $\quad\quad I \leftarrow \sum_{s=0}^{h-1} n_{as+jr+k} 2^s$ $\quad [I = I(jr+k)]$
6. $\quad\quad y \leftarrow y \times G[j, I]$
7. **return** y

Example 9.50 Once again, let us compute x^{2989}. Set $h = 3$ and $v = 2$ so that $a = 4 = rv$, hence $r = 2$. Form the following array from the digits of $2989 = (101110101101)_2$

s	3	2	1	0
$n_s \dots n_0$	1	1	0	1
$n_{a+s} \dots n_a$	1	0	1	0
$n_{2a+s} \dots n_{2a}$	1	0	1	1
$I(s)$	7	1	6	5

The precomputed values are

i	0	1	2	3	4	5	6	7
$G[0,i]$	1	x	x^{16}	xx^{16}	x^{256}	xx^{256}	$x^{16}x^{256}$	$xx^{16}x^{256}$
$G[1,i]$	1	x^4	x^{64}	x^4x^{64}	x^{1024}	x^4x^{1024}	$x^{64}x^{1024}$	$x^4x^{64}x^{1024}$

The algorithm proceeds as follows

k	1	1	1	0	0	0
j	—	1	0	—	1	0
$jr+k$	—	3	1	—	2	0
I	—	7	6	—	1	5
$G[j,I]$	—	x^{1092}	x^{272}	—	x^{4}	x^{257}
y	1	x^{1092}	x^{1364}	x^{2728}	x^{2732}	x^{2989}

Remarks 9.51

(i) One needs at most $a+r-2$ multiplications and $v(2^h-1)$ precomputed values. If squarings can be achieved efficiently or "on the fly" (for example in finite fields of even characteristic represented through normal bases, see Section 11.2.1.c), only 2^h-1 values must be stored.

(ii) The adaptation of this method to larger base representations or signed representations such as the non-adjacent form is straightforward.