Universität Gesamthochschule Paderborn
Fachbereich 17 · Mathematik/Informatik

Exponentiation in finite fields: theory and practice

Michael Nöcker

Diplomarbeit im Fach Informatik

Paderborn, im Oktober 1996

Betreuer:
Prof. Dr. Joachim von zur Gathen

Contents

1. Introduction **3**

2. Different exponentiation problems **5**

3. Addition chains **7**

3.1. Definitions and introduction 7
3.2. Binary method 12
3.3. Brauer's method 13
3.4. The q^r-ary method 16
3.5. The algorithm of Brickell, Gordon, McCurley & Wilson 19
3.6. Addition chain algorithms using data compression 23
3.7. Data Compression according to Ziv & Lempel 24
3.8. Addition chain algorithms with proper sets 29
3.9. A new algorithm based on data compression 35
3.10. Summarizing survey 41

4. Fast exponentiation **43**

4.1. The relation between addition chains and exponentiation 43
4.2. Results transferred from addition chains 44

5. Inversion in $\mathbb{F}_{q^n}$ **49**

5.1. Inversion based on Fermat's Little Theorem 49
5.2. Calculating the Inverse with Euclid 53
5.3. Comparison 55

6. Finite fields **56**

6.1. Introduction 56
6.2. Definitions 56
6.3. The representation of finite fields 57

7. Polynomial representation **59**

7.1. Irreducible polynomials 59
7.2. Fast multiplication for polynomials 59
7.3. Modular composition 65
7.4. Shoup's algorithm 71

8. Normal bases **75**
8.1. Definition and basic arithmetic operations 75
8.2. Normal bases generated by Gauß periods 79
8.3. Construction of the multiplication table $T_{\mathcal{N}}$ 82

9. Using fast multiplication within normal basis representation **87**
9.1. The basic idea 87
9.2. The residue class ring $\mathbb{F}_q[x]/(\Phi_r)$ 87
9.3. A transformation 88
9.4. Fast multiplication based on Gauß periods 89
9.5. A summarizing table 91

10. Practical results for addition chain heuristics **93**
10.1. The experiment 93
10.2. The classical algorithms 94
10.3. Algorithms based on data compression 96
10.4. Comparison between the two methods 97
10.5. Addition chains: theory vs. practice 97

11. Practical comparison of exponentiation algorithms **99**
11.1. The experiment 99
11.2. Remarks on the algorithms 102
11.3. Results 105

12. Conclusion **109**

References **111**

1. Introduction

Some cryptographical methods use exponentiation as a basic operation: e.g., the Diffie–Hellman method for key–exchange (Diffie & Hellman 1976), ElGamal's algorithm for digital signature (ElGamal 1985) or the RSA–scheme of Rivest *et al.* (1978). Using one of these public key cryptosystems one has to use large exponents in finite fields for encoded transmission. Therefore fast exponentiation, and as we will see in the sequel, also fast multiplication algorithms have to be developed.

Exponentiation in finite fields can be done by successive multiplication of smaller powers of the given basis. Hence we can speed up exponentiation by searching for a clever selection of smaller powers. This leads to the topic of addition chains because the problem of multiplication of powers of a given basis "can be easily reduced to addition, since the exponents are additive."(Knuth 1981, p. 444). Addition chains and their transfer to exponentiation algorithms are the first part (Sections 2–5) of this Diplomarbeit.

In Section 3 we present q-addition chains as a new generalization of addition chains which are helpful when discussing exponentiation over the finite field $\mathbb{F}_{q^n}$. We derive concrete upper bounds on the number of multiplications for exponentiation using addition chains and introduce a new addition chain algorithm based on data compression techniques. This algorithm is compared theoretically to the five best known addition chain algorithms that can be found in the literature.

We show in Section 5 that the problem of inversion in finite fields can be reduced to addition chains and compare this method to inversion using the fast Extended Euclidean Algorithm.

Another point to examine is how fast one single multiplication can be computed in finite fields. This coheres with the topic of representation of finite fields. Fast multiplication algorithms are based on polynomial arithmetic. Raising to a determined power can often be done much more efficiently by using normal bases. Hence the problem of representation of finite fields and the different exponentiation algorithms derived from this are contents of the second part (Sections 6–9) of this Diplomarbeit.

We give a survey on fast polynomial multiplication, fast matrix multiplication and modular composition in Section 7 and introduce the exponentiation algorithm of Shoup (1994) based on modular composition for arbitrary finite fields. Shoup (1994) restricted his algorithm to field extensions over $\mathbb{F}_2$.

We also analyze an exponentiation algorithm based on a normal basis representation of finite fields that uses a sparse multiplication table due to Ash *et al.* (1989) and Mullin *et al.* (1989). Both algorithms are theoretically compared in detail to an algorithmic idea of Gao *et al.* (1995a) that connects polynomial and normal basis representation via Gauß periods to get a fast exponentiation algorithm.

The last part (Sections 10–12) is concerned with practical results on implementations of both different addition chain algorithms and exponentiation algorithms using different ways to multiply. The implementations are written in C++.

In Section 10 we present our practical results on addition chains. For the first time all five algorithms that can be found in the literature are practically compared to each other and the new addition chain algorithm in detail. We also give a comparison between theoretical and practical results.

Section 11 is pointed out to be the first comparison of the three fastest exponentiation algorithms so far. We show that normal basis representation has to be combined with fast polynomial arithmetic to get optimal results.

Finally I would like to thank Prof. Dr. von zur Gathen and the members of his group for stimulating discussions, motivating support and excellent working conditions during the work on this Diplomarbeit.

2. Different exponentiation problems

The basic problem. The simplest way to compute b^e for $b \in G$, where G is a multiplicative group and $e \in \mathbb{N}$, is to start with b and multiply $e-1$ times by b. This brute force algorithm can be improved: "The time required for an exponentiation can be reduced by two orthogonal methods. On the one hand, one can reduce the *time per multiplication* by optimizing it. On the other hand, one can reduce the *number of multiplications*"(de Rooij 1995, p. 389). The first method also means to profit of special structures given for G: this can often reduce the time required for exponentiation. But first we concentrate on the idea to reduce the number of multiplications; methods to speed up multiplication will be discussed later.

PROBLEM 2.1. *Find an algorithm that needs a small number of multiplications to compute b^e for given $b \in G, e \in \mathbb{N}$.*

Three cases. There are three forms of the basic problem (see de Rooij 1995, pp. 389–390):

1. b and e are both variable. This problem is required e.g. for the ElGamal–algorithm (see ElGamal 1985).

2. b is fixed, e is variable. This case appears in many cryptosystems (see the references given by Brickell *et al.* 1993).

3. b is variable, e is fixed. This is the situation for RSA (see Rivest *et al.* 1978) when e is a key.

Since the first item is the most general case, we concentrate on this when introducing the different algorithms. When discussing these algorithms in detail we also examine their usage for the remaining two cases.

The representation of numbers. Before we work on methods to solve Problem 2.1 we have to look at the representation of numbers because several ideas are based on a special representation of the exponent e.

DEFINITION 2.2. *Given integers $m \in \mathbb{N}$ and $q \geq 2$, the q-ary representation of m is defined as $(m_{\lambda-1}, \ldots, m_0)$, with $\sum_{0\leq i<\lambda} m_i q^i = m$, $\lambda = \lfloor \log_q m \rfloor + 1$ and $m_0, \ldots, m_{\lambda-1} \in \{0, \ldots, q-1\}$. We write $(m_{\lambda-1}, \ldots, m_0) = (m)_q$.*

The q-ary representation for given m is unique. Because it is so important we give an example for the 2-ary or binary representation:

EXAMPLE 2.3. *Let* $m = 141, q = 2$. *Then* $m = 128 + 8 + 4 + 1 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 1$ *and we have* $(141)_2 = (10001101)$.

DEFINITION 2.4. *Let* $(m)_q = (m_{\lambda-1}, \ldots, m_0)$ *be the* q*-ary representation of* m. *The* q*-ary Hamming weight* $\nu_q(m)$ *is defined as* $\nu_q(m) = \#\{i\colon m_i \neq 0, 0 \leq i < \lambda\}$.

3. Addition chains

3.1. Definitions and introduction.

Original addition chains. Although Problem 2.1 deals with multiplication, the problem can be easily reduced to addition, since the exponents are additive. Therefore, we first concentrate on addition chains for finding algorithms to solve Problem 2.1.

DEFINITION 3.1. *1. An* (original) addition chain *for m is a sequence of integers $1 = a_0, a_1, \ldots, a_L = m$ with the property that $a_i = a_j + a_k$ for some $k \leq j < i$ for all $i = 1, 2, \ldots, L$ (Knuth 1981). L is its length.*

2. The smallest L for which there exists an addition chain of length L for m is denoted by $l(m)$ (see Knuth 1981).

Following Knuth (1981), where one can find an excellent survey on addition chains, we may assume without loss of generality that an addition chain is 'ascending':

$$1 = a_0 < a_1 < \ldots < a_L = m. \tag{3.1}$$

We also use a few special terms in connection with addition chains that were introduced by Knuth (1981). By definition we have, for $1 \leq i \leq L$, $a_i = a_j + a_k$ for some $0 \leq j \leq k < i$.

1. If $j = k \leq i - 1$ then we call step i of (3.1) a *doubling*.
2. If $j < k = i - 1$ then step i is called a *star step*.

Knuth (1981) uses the term doubling in a more restrictive way by imposing $j = k = i - 1$.

A generalization. For our algorithmic purposes it is useful to generalize the notion of addition chains in the following way:

DEFINITION 3.2. *Let $q, m \in \mathbb{N}$.*

*A q-*addition chain *for m is a sequence of integers $1 = a_0, a_1, \ldots, a_L = m$ with the property that $a_i = a_j + a_k$ for some $k \leq j < i$ or $a_i = q \cdot a_j$ for some $j < i$ for all $i = 1, 2, \ldots, L$.*

We denote the length of a shortest q-addition chain for a given m by $l_q(m)$. We call step i a *q-step* if $a_i = q \cdot a_j$. For $q = 2$ this is just a doubling.

Every q-addition chain can be rewritten as an original addition chain by expanding $a_i = q \cdot a_j$ to $2a_j, \ldots, qa_j = a_i$. This can be done using $\lceil \log_2 q \rceil$ doublings and at most $\lceil \log_2 q \rceil$ star steps. If we denote the number of doublings by D, the number of q-steps by Q and the number of remaining addition steps by A we can write

$$L = D + Q + A \tag{3.2}$$

for a q-addition chain of length L. We therefore get an upper bound on the length $L' = D' + A'$ of an original addition chain generated out of a q-addition chain of length $L = D + Q + A$:

$$\begin{aligned} D' &\leq D + \lceil \log_2 q \rceil Q, \\ A' &\leq \lceil \log_2 q \rceil Q + A, \\ L' &= D' + A' \leq D + A + 2\lceil \log_2 q \rceil Q. \end{aligned}$$

3.1.1. Complexity of addition chains. To find algorithms for Problem 2.1 we now have to find algorithms that generate short addition chains. This leads to a new problem:

PROBLEM 3.3. *Let m and k be positive integers. Does there exist an addition chain for m with length $L \leq k$?*

The answer was given by Downey *et al.* (1981):

FACT 3.4. *Problem 3.3 is NP–complete.*

Therefore, it would not be a promising approach to try and calculate an addition chain with shortest length; rather we look for one with short length.

Word chains. Let $\mathcal{A}$ be a finite set that we shall call an *alphabet.* A *q-letter alphabet* is an alphabet $\mathcal{A}$ with q elements. We can assume without loss of generality that $\mathcal{A} = \{0, \ldots, q-1\}$.

DEFINITION 3.5. *A* word *over the alphabet $\mathcal{A}$ is a finite sequence of elements of $\mathcal{A}$:*

$$(m_{\lambda-1}, m_{\lambda-2}, \ldots, m_1, m_0) \in \mathcal{A}^\lambda$$

for some $\lambda \in \mathbb{N}$. The set of all words *over $\mathcal{A}$ is denoted by $\mathcal{A}^*$.*

A survey on the topics of words can be found in Lothaire (1983).

DEFINITION 3.6 (CF. BERSTEL & BRLEK 1987). *1. A* word chain *for a word $w \in \mathcal{A}^*$ over a q-letter alphabet $\mathcal{A}$ is a sequence*

$$w_{1-q}, \ldots, w_0, w_1, \ldots, w_L$$

of words such that $\mathcal{A} = \{w_{1-q}, \ldots, w_0\}$, $w_L = w$ and for each $1 \le i \le L$ there exist j, k with $1 - q \le j, k < i$ such that $w_i = (w_j, w_k)$ is the concatenation of w_j and w_k. The length of the word chain is the integer L.

2. The shortest length L for which there exists a word chain for w is denoted by $l_{\mathcal{A}}(w)$.

REMARK 3.7. *1. (Original) addition chains correspond bijectively to word chains over a one–letter alphabet, and therefore word chains are a generalization of addition chains.*

2. Word chains provide a short notation for shifts and concatenations of the q-ary representations.

3. Let $w_{1-q}, \ldots, w_L$ be an addition chain over $\mathcal{A}$. Let $1 \le i \le L$ and $w_i = (m_{\lambda-1}, \ldots, m_0) \in \mathcal{A}^\lambda$. Then there exist $1 - q \le j, k < i$ with $w_i = (w_j, w_k)$ and $\lambda' \in \{1, \ldots, \lambda - 1\}$ with $w_j = (m_{\lambda-1}, \ldots, m_{\lambda'})$ and $w_k = (m_{\lambda'-1}, \ldots, m_0)$.

LEMMA 3.8. *Let $w \in \mathcal{A}^\lambda$. Then $l_{\mathcal{A}}(w) \ge \log_2 \lambda$ is a lower bound on the shortest length of a word chain for w. If $l_{\mathcal{A}} = \log_2 \lambda$, then $w = (w', w')$ for some $w' \in \mathcal{A}^*$.*

PROOF. (by induction on λ:) For $\lambda = 1$ we have $w \in \mathcal{A}$ and thus $l_{\mathcal{A}}(w) = 0 \ge \log_2 \lambda$. Let us assume that the induction hypothesis holds for all $w' \in \mathcal{A}^{\lambda'}$ with $\lambda' < \lambda$. Let $w_{1-q}, \ldots, w_L$ be a shortest word chain for w over $\mathcal{A}$ with $L = l_{\mathcal{A}}(w)$. Then there exist $1 - q \le j, k < L$ with $w = (w_j, w_k)$. But $l_{\mathcal{A}} = \max\{l_{\mathcal{A}}(w_j), l_{\mathcal{A}}(w_k)\} + 1 \ge \log_2 \frac{\lambda}{2} + 1 = \log_2 \lambda$. □

Comparison. We have introduced two generalizations of addition chains so far. We simulate word chains over a q-letter alphabet $\mathcal{A}$ by q-addition chains. This can be done by identifying $\mathcal{A}^*$ and $\mathbb{N}$ via the q-ary representation in the following way: Let $m \in \mathbb{N}$ with $(m)_q = (m_{\lambda-1}, \ldots, m_0)$. Then $(m_{\lambda-1}, \ldots, m_0) \in \mathcal{A}^*$. Vice versa let $m \in \mathcal{A}^*$ with $(m)_q = (m_{\lambda-1}, \ldots, m_0)$. Then $\sum_{0 \le i < \lambda} m_i q^i \in \mathbb{N}$.

Let $\mathcal{A}$ be a q-letter alphabet. Let $m \in \mathcal{A}^*$ with $(m)_q = (m_{\lambda-1}, \ldots, m_0)$. Then $m_i \in \{0, \ldots, q-1\}$ for $0 \le i < \lambda$. Let $w_{1-q}, \ldots, w_0, w_1, \ldots, w_L$ be a word chain for m. Then we have a q-addition chain $a_0, \ldots, a_{L'}$ by the rules:

1. Set $a_0 = w_{1-q+1} = 1, \ldots, a_{q-2} = w_0 = q-1$.

2. Let $1 \le i \le L$ and $1 - q \le j, k < i$ with $w_i = w_j w_k$. Let j', k' with $a_{j'} = w_j$ and $a_{k'} = w_k$. Let $\lambda = \lfloor \log_q w_k \rfloor + 1$. Then we can create $a_{j'}, q \cdot a_{j'}, \ldots, q^\lambda \cdot a_{j'}, q^\lambda \cdot a_{j'} + a_{k'} = a_{i'} = w_i$.

Therefore step i in a word chain can be simulated by a q-addition chain using λ q-steps plus one star step.

PROPOSITION 3.9. *A word chain of length L can be simulated by a q-addition chain of length $L' \le (q+1)L$.*

We illustrate this in an example.

EXAMPLE 3.10. *Let $m = 141$ and $q = 4$. Then the 4-ary representation of 141 is $(141)_4 = (2, 0, 3, 1)$.*

1. *A word chain for $m = (2,0,3,1)$: $w_{-3}, w_{-2}, w_{-1}, w_0, w_1, w_2, w_3 = 0, 1, 2, 3, (2,0), (3,1), (2,0,3,1)$. It can be easily seen that there is no shorter word chain for $m = (2,0,3,1) = (141)_4$ over $\{0,1,2,3\}$.*

2. *A 4-addition chain for $m = 141$ derived from the given word chain: $a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8 = 1, 2, 3, 4 \cdot 2, 4 \cdot 3, 12 + 1, 4 \cdot 8, 4 \cdot 32, 128 + 13$.*

3. *An (original) addition chain for $m = 141$ derived from the given 4-addition chain: $b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10}, b_{11}, b_{12} = 1, 2, 3, 2+2, 4+4, 3+3, 6+6, 12+1, 8+8, 16+16, 32+32, 64+64, 128+13$. This is not the shortest possible addition chain for $m = 141$. A shorter one is e.g.: $b'_0, b'_1, b'_2, b'_3, b'_4, b'_5, b'_6, b'_7, b'_8, b'_9, b'_{10} = 1, 2, 2+2, 4+4, 8+8, 16+1, 17+17, 34+1, 35+35, 70+70, 140+1$.*

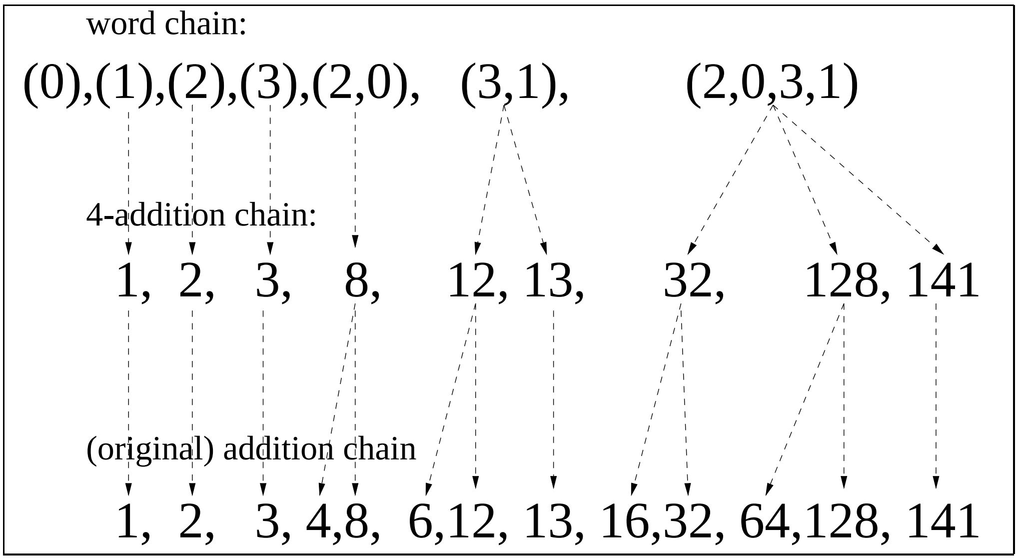

Figure 3.1: Simulation of a word chain for $(141)_4 = (2, 0, 3, 1)$ by an original addition chain via a 4-addition chain.

In the other direction, we cannot simulate all q-addition chains by word chains over a q-letter alphabet because we cannot express $a_i = a_j + a_k$ directly as a step in a word chain. Because some algorithms given below operate not only with shifts and concatenations over the q-ary representation (e.g. Algorithm 3.25) we concentrate on the q-addition chains which are just original addition chains for $q = 2$.

An upper bound on the number of star steps. Berstel & Brlek (1987) proved the following:

THEOREM 3.11. *Let $\mathcal{A}$ be a q-letter alphabet. For an arbitrary $\varepsilon > 0$ there is a constant λ_0 such that, for any word $w \in \mathcal{A}^*$ of length $\lambda \geq \lambda_0$, there exists a word chain computing w of length $\leq (1+\varepsilon)\frac{\lambda}{\log_q \lambda}$.*

REMARK 3.12. *Let $m \in \mathbb{N}$ and $\lambda = \lfloor \log_2 m \rfloor + 1$. According to Theorem 3.11 and the relationship between addition chains, q-addition chains and word chains, there exists an (original) addition chain with λ doublings and $\frac{\lambda}{\log \lambda}(1 + o(1))$ star steps. Corollary 3.28 will give the number of doublings and star steps more precisely.*

3.2. Binary method.

Basic idea. The most commonly used algorithm to generate addition chains is probably the binary method. The algorithmic idea is based on the binary representation $(m)_2 = (m_{\lambda-1}, \ldots, m_0)$ of m and the facts that

- $(\sum_{0\le j<\lambda-i-1} m_{i+1+j}2^j)_2 = (m_{\lambda-1}, \ldots, m_{i+1})$, $(2\sum_{0\le j<\lambda-i-1} m_{i+1+j}2^j)_2 = (\sum_{1\le j<\lambda-i} m_{i+j}2^j)_2 = (m_{\lambda-1}, \ldots, m_{i+1}, 0)$ and
- $\sum_{1\le j<\lambda-i} m_{i+j}2^j + m_i = \sum_{0\le j<\lambda-i} m_{i+j}2^j$ and $(\sum_{0\le j<\lambda-i} m_{i+j}2^j)_2 = (m_{\lambda-1}, \ldots, m_i)$.

The first equation denotes a doubling in the notation of addition chains and the second one is just a star step. In the literature (see e.g. Jungnickel 1993) it is often suggested to scan from low–order to high–order. The better way seems to be to scan in the opposite direction, since we then only have to deal with one intermediate result. In the other case we have to evaluate supplementary 2^i in the ith step.

Algorithm. We can derive an algorithm straight forward using the ideas above:

ALGORITHM 3.13. `binary`
Input: $m \in \mathbb{N}$ *with* $(m)_2 = (m_{\lambda-1}, \ldots, m_0)$ *and* $\lambda = \lfloor \log_2 m \rfloor + 1$.
Output: $1 = a_0, \ldots, a_L = m$*, an addition chain for* m *of length* L*.*

1. *Set* $d_0 = 1$ *and* $j = 1$*. Set* $a_0 = d_0$*.*
2. *For* $i = \lambda - 2$ *downto* 0 *compute*
   3. *Compute* $a_j = a_{j-1} + a_{j-1}$*. Set* $j = j + 1$ *[Comment: This is the doubling.]*
   4. *If* $m_i = 1$ *then compute* $a_j = a_{j-1} + d_0$ *and set* $j = j + 1$*. [Comment: The star step depends on* $m_i = 1$*.]*
5. *Return* $d_0, a_1, \ldots, a_{j-1}$*.*

LEMMA 3.14. *Algorithm* `binary` *computes an addition chain for* m*. It uses* $A = \nu_2(m) - 1$ *star steps and* $D = \lfloor \log_2 m \rfloor$ *doublings, where* $\nu_2(m)$ *is the Hamming weight of* $(m)_2$*.*

Proof. The loop invariant for the loop in Steps 2–4 can be chosen as follows: after round i we have an addition chain $1 = d_0, a_1, ..., a_{j-1}$ with $(a_{j-1})_2 = (m_{\lambda-1}, \ldots, m_i)$.

With $i = \lambda - 1$ and $j = 1$ we have $(1)_2 = (d_0)_2 = (m_\lambda - 1)$, an addition chain for 1 before entering the loop and the invariant holds. Let us now assume that the invariant also holds for $i < k < \lambda$. Then in round i we generate $a_j = 2a_{j-1}$ with $(a_{j-1})_2 = (m_{\lambda-1}, \ldots, m_{i+1})$ and $(a_j)_2 = (m_{\lambda-1}, \ldots, m_{i+1}, 0)$. If $m_i = 1$ we have to do Step 4: $a_{j+1} = a_j + a_0$ and $a_{j+1} = (m_{\lambda-1}, \ldots, m_{i+1}, m_i)$ and therefore — with ascending j — in both cases the invariant holds. In Step 5 the algorithm returns the addition chain $1 = d_0, a_1, \ldots, a_{j-1}$ with $(a_{j-1})_2 = (m_{\lambda-1}, \ldots, m_0) = (m)_2$. This shows partial correctness. Termination and thus total correctness are clear.

For the cost analysis, we note that for any $i < \lambda$, a star step is brought to the addition chain iff $m_i = 1$ for all $0 \leq i < \lambda - 1$ (see Step 4). Thus $A = \nu_2(m) - 1$ because $m_{\lambda-1} = 1$. At every lap we get one doubling and with $\lambda - 2 + 1$ laps we get $D = \lambda - 1$. □

Worst and Average Case. The worst case occurs at $m = 2^k - 1, k \in \mathbb{N}$. Then we have $D = \lfloor \log_2 (2^k - 1) \rfloor = k - 1$ and $A = \nu_2(2^k - 1) - 1 = k - 1$ star steps. According to Equation (3.2) we get $L = A + D = k - 1 + k - 1 = 2k - 2 = \log_2 2^{2(k-1)} = 2 \log_2 2^{k-1} < 2 \log_2 m$ as an upper bound on the length of the addition chain.

Let $k \in \mathbb{N}$ and $\Omega = \{m \in \mathbb{N}: m < 2^k\}$ be a probability space with the uniform distribution. For an arbitrary exponent $m \in \Omega$ we have $m_i = 0$ with probability $\frac{1}{2}$. Therefore we can expect $\nu_2(m) = \frac{1}{2} \cdot \lceil \log_2 m \rceil$ on average.

Corollary 3.15. *The binary method generates an addition chain on input $m \in \mathbb{N}$ with $A_{worst} = \lceil \log_2 m \rceil - 1$ star steps in the worst case and $A_{ave} = \frac{1}{2} \cdot \lceil \log_2 m \rceil - 1$ star steps on the average. There are always $D = \lfloor \log_2 m \rfloor$ doublings.*

3.3. Brauer's method.

Basic idea. The question whether the upper bound $l(m) \leq 2 \log_2 m$ given by the binary method could be improved leads to a generalization of the binary method. The following algorithm was suggested by Brauer (1939) who used it to create addition chains for m of length $L \leq (1 + o(1)) \log_2 m$. The idea is to use the 2^r-ary representation of m instead of the binary representation with $r \in \mathbb{N}$ a selectable parameter. To be able to do so we have to precompute all elements $d_0 = 1, \ldots, d_{2^r-2} = 2^r - 1$.

The algorithm.

NOTATION 3.16. *To distinguish between precomputed elements of the addition chain — i.e., elements that are (probably) used more than once — and intermediate results we use the following notation:*

- *The* precomputed elements *are denoted by* d_j. *The* set of all precomputed elements *is given by* $\mathcal{D}$.
- *Other elements are denoted by* a_j.

ALGORITHM 3.17. `brauer`
Input: $m, r \in \mathbb{N}$ *with* $(m)_{2^r} = (m_{\lambda-1}, \ldots, m_0)$ *and* $\lambda = \lfloor \log_2 m \rfloor + 1$.
Output: $L \in \mathbb{N}$ *and* $1 = d_0, \ldots, d_{2^r-2}, a_{2^r-1}, \ldots, a_L = m$, *a* 2^r*-addition chain for* m *of length* L.

1. *Set* $d_0 = 1$. *Compute* $d_j = d_{j-1} + d_0$ *for all* $j = 1, \ldots, 2^r - 2$. *Set* $\mathcal{D} = \{d_0, \ldots, d_{2^r-2}\}$.
2. *Set* $j = 2^r - 1$. *Set* $a_{j-1} = d_k$ *with* $d_k \in \mathcal{D}$ *and* $(d_k)_{2^r} = (m_{\lambda-1})$.
3. *For* $i = \lambda - 2$ *downto* 0 *do*
   4. *Compute* $a_j = 2^r \cdot a_{j-1}$. *Set* $j = j + 1$.
   5. *If* $m_i \neq 0$ *then compute* $a_j = a_{j-1} + d_k$ *with* $d_k \in \mathcal{D}$ *and* $(d_k)_{2^r} = m_i$ *and set* $j = j + 1$.
6. *Return* $d_0, \ldots, d_{2^r-2}, a_{2^r-1}, \ldots, a_{j-1}$.

LEMMA 3.18. *Algorithm* `brauer` *computes a* 2^r*-addition chain for given* $m \in \mathbb{N}$.

PROOF. The correctness of the algorithm can be proven simular to Lemma 3.14 noting that all possible values for m_i, $0 \leq i < \lambda$, with $m_i \neq 0$ can be found in the set $\mathcal{D}$ of the precomputed values. □

LEMMA 3.19. *The addition chain algorithm* `brauer` *generates a* 2^r*-addition chain for* m *with*

$$\begin{aligned} A &= \nu_{2^r}(m) + 2^r - 3 \text{ star steps and} \\ Q &= \lfloor \log_{2^r} m \rfloor \; 2^r\text{-steps.} \end{aligned}$$

The algorithm can be modified to compute an (original) addition chain with

$$\begin{aligned} A &= \nu_{2^r}(m) + 2^r - 3 \text{ star steps and} \\ D &= r\lfloor \log_{2^r} m \rfloor - (r - \lfloor \log_2 \lfloor \frac{m}{2^{r\lfloor \log_{2^r} m \rfloor}} \rfloor \rfloor) \text{ doublings.} \end{aligned}$$

PROOF. The precomputation of $d_1 = 2, \ldots, d_{2^r-2} = 2^r - 1$ can be done with $2^r - 2$ star steps. Let $(m)_{2^r} = (m_{\lambda-1}, \ldots, m_0)$ be the 2^r-ary representation of m with $\lambda = \lfloor \log_{2^r} m \rfloor + 1$ and $m_{\lambda-1} = (\lfloor \frac{m}{2^{r(\lambda-1)}} \rfloor)_{2^r}$. The algorithm can be summarized as follows: In round i a shift has to be done which needs one 2^r-step. If $m_i \neq 0$ a further star step has to be computed. There are $\lambda - 1$ rounds. Therefore the algorithm produces a 2^r-addition chain with $Q = \lambda - 1$ 2^r-steps and $A = \nu_{2^r}(m) - 1 + 2^r - 2$ star steps (because $m_{\lambda-1} \neq 0$).

To compute an (original) addition chain we exchange Step 4 with

4'. Compute $a_{j+k} = a_{j+k-1} + a_{j+k-1}$ for all $k = 0, \ldots, r-1$. Set $j = j + 1$.

This gives r doublings for every 2^r-step. But let us have a closer look at the first round. We have $m_{\lambda-1} = (\lfloor \frac{m}{2^{r(\lambda-1)}} \rfloor)_{2^r}$ and we create r elements: $2m_{\lambda-1}, 2^2 m_{\lambda-1}, \ldots, 2^r m_{\lambda-1}$. But we don't need to count the elements that occur twice because they have already been added to the addition chain by precomputation: Let $k \in 0, \ldots, r-1$ with $2^k m_{\lambda-1} \leq 2^r < 2^{k+1} m_{\lambda-1}$ with k denoting the number of elements that are counted twice. Then $k \leq \log_2 \frac{2^r}{m_{\lambda-1}} = r - \log_2 m_{\lambda-1}$. Therefore $k = r - \lfloor \log_2 m_{\lambda-1} \rfloor$ elements are counted twice if we use doublings. Hence, the total number of doublings is $D = r\lambda - 1 - (r - \lfloor \log_2 m_{\lambda-1} \rfloor)$. Using 2^r-steps would not generate any elements twice. □

Conclusions. Using this result and the fact that $L = A + D$ for an original addition chain we can easily prove two further corollaries:

COROLLARY 3.20. *Let $l(m)$ be the shortest addition chain for m. Let $\mu = \log_2 m$. There are upper bounds given by*

$$l(m) \leq \mu(1 + \frac{2}{\log_2 \mu} + \frac{2}{\sqrt{\mu}}) \leq \mu + 2\frac{\mu}{\log_2 \mu}(1 + o(1)).$$

PROOF. (cf. Brauer 1939) Let $r = \lfloor \frac{1}{2} \log_2 \mu \rfloor + 1$. Then the following inequalities hold:

$$\nu_{2^r}(m) \leq \log_{2^r} m + 1 = \frac{1}{r}\mu + 1$$

$$\begin{aligned}
&= \frac{\mu}{\lfloor\frac{1}{2}\log_2\mu\rfloor+1}+1 \le \frac{2\mu}{\log_2\mu}+1, \\
r\lfloor\log_{2^r} m\rfloor &= r\lfloor\frac{1}{r}\mu\rfloor \le \mu, \\
2^r-3 &= 2^{\lfloor\frac{1}{2}\log_2\mu\rfloor+1}-3 \le 2\cdot 2^{\frac{1}{2}\log_2\mu}-3 \\
&= 2\cdot\sqrt{\mu}-3.
\end{aligned}$$

Using these inequalities and the fact that $l(m) \le L$ where $L = A + D$ is the length of the addition chain for m created by Algorithm `brauer`, we can write:

$$\begin{aligned}
l(m) \le L &= A + D \\
&= r\lfloor\log_{2^r} m\rfloor + \nu_{2^r}(m) + 2^r - 3 - (r - \lfloor\log_2 \frac{m}{\lfloor 2^{r\lfloor\log_{2^r} m\rfloor}\rfloor}\rfloor) \\
&\le \mu + \frac{2\mu}{\log_2\mu} + 1 + 2\sqrt{\mu} - 3 \\
&< \mu(1 + \frac{2}{\log_2\mu} + \frac{2}{\sqrt{\mu}}) \\
&= \mu + 2\frac{\mu}{\log_2\mu}(1 + \frac{\log_2\mu}{\sqrt{\mu}}) \\
&\le \mu + 2\frac{\mu}{\log_2\mu}(1 + o(1)). \quad \square
\end{aligned}$$

Brauer uses $r = \lfloor\ln\ln m\rfloor + 1$ to prove the upper bound $l(m) \le \mu(1 + \frac{1}{\ln\ln m} + \frac{2\ln 2}{(\ln m)^{1-\ln 2}})$. For implementation purposes the above chosen r depending on logarithms in basis 2 is more suited.

COROLLARY 3.21. *The binary method generates addition chains for $m \in \mathbb{N}$ of length*

$$\lfloor\log_2 m\rfloor + \nu_2(m) - 1.$$

PROOF. Algorithm `brauer` with $r = 1$ is just the binary method. Lemma 3.19 about the length of the addition chains for Algorithm `brauer` proves the statement because $\log_2\lfloor\frac{m}{2^{\lfloor\log_2 m\rfloor}}\rfloor = 0$. $\square$

3.4. The q^r-ary method.

Basic idea. Brauer's method generalizes the binary method by taking r bits as one new element. This idea of using the 2^r-ary representation can also be generalized by using just the q^r-ary representation with $q \in \mathbb{N}$ and r as a selectable parameter again. For given $m \in \mathbb{N}$ we have

$$m = (m_{\lambda-1}, \ldots, m_0)_{q^r} \text{ with } \lambda = \lfloor \log_{q^r} m \rfloor + 1 \text{ and } 0 \leq m_i < q^r$$

for all $i \in \{0, \ldots, \lambda - 1\}$.

To get a q^r-addition chain we can use the following equality, where the right side is well known as *Horner's rule*:

$$m = \sum_{0 \leq i < \lambda} m_i (q^r)^i = q^r(\cdots q^r(q^r m_{\lambda-1} + m_{\lambda-2}) + \cdots + m_1) + m_0. \tag{3.3}$$

We therefore can create a q^r-addition chain by using this grouping of additions and multiplications if we first precompute $d_0 = 1, \ldots, d_{q^r-2} = q^r - 1$. The algorithm derivated from this is quite simular to Algorithm `brauer` — we only have to substitute '2' by 'q'. The algorithm can easily be modified to create a q-addition chain. Indeed for $q = 2$ we just get Algorithm `brauer` and for $q = 2$ and $r = 1$ we have the binary method.

Remarks. We emphasize two points:

1. Perhaps not all of the elements $d_0, \ldots, d_{q^r-2}$ have to be used to calculate an addition chain for m. To avoid unnecessary calculations other algorithmic ideas have to be added.

2. There is no star step in the addition chain when $m_i = 0$. In this case, we have to do only q^r-steps to shift to the next element of the q^r-ary representation of m.

Number of steps. Because of the previous remark it makes sense to separate the analysis: we first work on the number of steps used in precomputation and then take a look at the computation of the other elements of the addition chain for m. We analyze the number of q-steps more precisely.

1. Because we distinguish between q-steps and 'ordinary' steps we count all $i \in \{1, \ldots, q^r - 2\}$ with $i \not\equiv 0 \bmod q$. Since each step yields a new element of the addition chain we count $A_1 = q^r - q^{r-1} - 1$ star steps and $Q_1 = r - 1$ q-steps.

2. In Equation (3.3) we have $\lambda - 1$ additions and $\lambda - 1$ multiplications with q^r. Therefore we get at most $A_2 = \lambda - 1$ star steps and $Q_2 = (\lambda - 1)r$ q-steps.

The following lemma summarizes the results. A simular result relative to exponentiation can be found in von zur Gathen (1992).

LEMMA 3.22. *Let $(m)_{q^r} = (m_{\lambda-1}, \ldots, m_0)$ be the q^r-ary representation of m with $\lambda = \lfloor \log_{q^r} m \rfloor + 1$. Then we can compute a q-addition chain using at most*

$$\begin{aligned} A &= q^r - q^{r-1} - 1 + \lfloor \log_{q^r} m \rfloor \text{ star steps and} \\ Q &= r - 1 + r\lfloor \log_{q^r} m \rfloor \text{ } q\text{-steps.} \end{aligned}$$

COROLLARY 3.23. *Let $m, q \in \mathbb{N}$, $q \geq 2$, and $\mu = \log_q m$. There is a q-addition chain for m of length at most*

$$\mu + q\frac{\mu}{\log_q \mu}(1 + \frac{\log_q \mu}{\sqrt[q]{\mu}} + \frac{1}{\mu}) \leq \mu + q\frac{\mu}{\log_q \mu}(1 + o(1)).$$

PROOF. Choose $r = \lfloor \frac{1}{q} \log_q \mu \rfloor + 1$. Then we get the followings estimates for A and Q:

$$\begin{aligned} A &= q^r - q^{r-1} - 1 + \lfloor \log_{q^r} m \rfloor \\ &= q^{r-1}(q-1) - 1 + \lfloor \frac{1}{r} \log_q m \rfloor \\ &\leq q^{\frac{1}{q}\log_q \mu}(q-1) + \frac{\mu}{\frac{1}{q}\log_q \mu} \\ &\leq q\sqrt[q]{\mu} + q\frac{\mu}{\log_q \mu} \\ &= q\frac{\mu}{\log_q \mu}(1 + \frac{\log_q \mu}{\sqrt[q]{\mu}}) \text{ and} \\ Q &= r - 1 + r\lfloor \log_{q^r} m \rfloor \\ &\leq r - 1 + \mu \\ &\leq \frac{1}{q}\log_q \mu + \mu. \end{aligned}$$

Using the fact that $L = A + D$ completes the proof. □

COROLLARY 3.24. *Let $k, m^{(1)}, \ldots, m^{(k)} \in \mathbb{N}$. Then a q-addition chain containing $m^{(1)}, \ldots, m^{(k)}$ can be computed in at most*

$$\begin{aligned} A &= q^r - q^{r-1} - 1 + \sum_{1 \leq i \leq k} \lfloor \log_{q^r} m^{(i)} \rfloor \\ &\leq q^r - q^{r-1} - 1 + k\lfloor \log_{q^r} m \rfloor \text{ star steps and} \\ Q &= r - 1 + r \sum_{1 \leq i \leq k} \lfloor \log_{q^r} m^{(i)} \rfloor \leq r - 1 + kr\lfloor \log_{q^r} m \rfloor \text{ } q\text{-steps,} \end{aligned}$$

where $m = \max_{1 \le i \le k} m^{(i)}$.

PROOF. We can use the fact that the precomputation has to be done only once. Hence we have to do $A_1 = q^r - q^{r-1} - 1$ star steps and $Q_1 = r - 1$ q-steps for precomputation and $A_{2,i} = \lfloor \log_{q^r} m^{(i)} \rfloor$ star steps and $Q_{2,i} = r \lfloor \log_{q^r} m^{(i)} \rfloor$ q-steps for $1 \le i \le k$. □

3.5. The algorithm of Brickell, Gordon, McCurley & Wilson.

Basic idea. The q^r-ary method computes an addition chain for m according to $m = \sum_{i=0}^{\lambda} m_i (q^r)^i$ by first (pre)computing all possible non–zero values for m_i: $d_0 = 1, d_1 = 2, \ldots, d_{q^r-2} = q^r - 1$. The algorithm of Brickell *et al.* (1993) which is introduced now is also based on the q^r-ary representation of m but it uses a different arrangement. It (pre)computes $q^r, \ldots, (q^r)^{\lambda-1}$ where $\lambda = \lfloor \log_{q^r} m \rfloor + 1$ as before. Then it uses the grouping

$$m = \sum_{1 \le j < q^r} \Big(\sum_{m_i \ge j} q^{ri} \Big) \tag{3.4}$$

to generate a q-ary addition chain for m.

The algorithm. Equation 3.4 is based on the idea of rewriting m as a sum of special smaller summands, and cleverly computing this sum (see de Rooij 1995). It leads to the following algorithm to generate a q-addition chain for m (cf. Brickell *et al.* 1993):

ALGORITHM 3.25. `bgmw`
Input: $m \in \mathbb{N}$ *with* $(m)_{q^r} = (m_{\lambda-1}, \ldots, m_0)$, *the* q^r*-ary representation of* m *and* $\lambda = \lfloor \log_{q^r} m \rfloor + 1$.
Output: $1 = a_0, \ldots, a_L = m$, *a* q*-addition chain for* m *of length* L.

1. *Set* $a_0 = 1$. *Compute* $\mathcal{D} = \{d_{ri} : i = 0, \ldots, \lambda - 1\}$ *by successively computing* $a_{(\lambda-1)(i-1)+j} = q \cdot a_{(\lambda-1)(i-1)+(j-1)}$ *for all* $i = 1, \ldots, \lambda - 1$ *and* $j = 1, \ldots, r$ *and set* $d_{ri} = a_{ri}$ *for all* $i = 0, \ldots, \lambda - 1$. *[Comment: This is a precomputation.]*
2. *Initialize* $\alpha = 0$ *and* $\beta = 0$. *Set* $j = r(\lambda - 1)$.
3. *For* $k = q^r - 1$ *downto* 1 *compute*
    4. *For each* $i \in \{0, \ldots, \lambda - 1\}$ *such that* $m_i = k$ *do*

5. *Compute* $\alpha = \alpha + d_{ri}$ *with* $d_{ri} = q^{ri}$. *If* $\alpha \notin \{a_l : 0 \leq l < j\}$ *then set* $a_j = \alpha$ *and* $j = j + 1$.

6. *Compute* $\beta = \beta + \alpha$. *If* $\beta \notin \{a_l : 0 \leq l < j\}$ *then set* $a_j = \beta$ *and* $j = j + 1$.

7. *Return* $1 = a_0, \ldots, a_{j-1} = m$.

Correctness.

LEMMA 3.26. *Algorithm* `bgmw` *computes a* q*-addition chain for* $m \in \mathbb{N}$ *correctly.*

PROOF. The precomputation in Step 1 calculates an addition chain with $a_0 = 1$ and $a_{r(i-1)+j} = q^{r(i-1)+j}$ for $i = 1, \ldots, \lambda - 1$ and $j = 1, \ldots, r$. In particular, $d_{ri} = q^{ri}$ for $i = 0, \ldots, \lambda - 1$ are precomputed.

The main part of the algorithm (Steps 2–7) has two loops:

1. The inner loop (Steps 4+5) is the loop that sorts the summands q^{ri} for $0 \leq i < \lambda - 1$ according to descending m_i beginning with $m_i = q^r - 1$. With $\alpha_j = \sum_{m_i = j} q^{ri}$ for $1 \leq j < q^r$, the invariant for this loop can be formulated as follows:
$$\alpha = \sum_{j \leq k < q^r} \alpha_k.$$

2. The outer loop (Step 3–6) concatenates the result of the inner loop with the intermediate result of the previous turn. With $\beta_j = \sum_{j \leq k < q^r} (\alpha_k \cdot (k - j + 1))$ for $1 \leq j < q^r$, we can formulate as invariant:
$$\beta = \beta_j.$$

With $k = q^r$ before Step 3 the invariants hold because $\alpha = \beta = 0$ (Step 2). Assume that the invariants are also true before round k of the outer loop. Then with Steps 4+5 we have

$$\alpha = \sum_{j+1 \leq k < q^r} \alpha_k + \sum_{m_i = j} q^{ri} = \Big(\sum_{k=j+1}^{q^r - 1} \alpha_k\Big) + \alpha_j = \sum_{j \leq k < q^r} \alpha_k,$$

and the inner invariant holds.

With this new α Step 6 calculates

$$\begin{aligned}
\beta &= \beta_{j+1} + \alpha = \beta_{j+1} + \sum_{j \le k < q^r} \alpha_k \\
&= \sum_{j+1 \le k < q^r} (\alpha_k \cdot (k - (j+1) + 1)) + \sum_{j \le k < q^r} \alpha_k \\
&= (\sum_{j+1 \le k < q^r} (\alpha_k \cdot (k - (j+1) + 1 + 1)) + \alpha_j \cdot (j - j + 1)) \\
&= \sum_{j \le k < q^r} (\alpha_k \cdot (k - j + 1)) = \beta_j.
\end{aligned}$$

This shows the right choice of the second invariant. Therefore the algorithm returns

$$\beta = \beta_1 = \sum_{1 \le k < q^r} (\alpha_k \cdot (k - 1 + 1)) = \sum_{1 \le j < q^r} (\sum_{m_i \ge j} q^{ri})$$

which is just Equation (3.4). This shows partial correctness. The termination of both loops is clear and thus the algorithm works correctly. □

Number of steps. Again we analyze the number of star steps and the number of q-steps for precomputation separately:

1. The precomputation in Step 1 uses $Q_1 = (\lambda - 1) \cdot r$ q-steps which can be seen directly. There are no further addition steps, for this means $A_1 = 0$.

2. The outer loop is repeated $q^r - 1$ times. Therefore there are at most $q^r - 2$ addition steps not counting the first one. For the first one we have $\beta = \alpha + 0$. But the element α has already been added to the addition chain. Additional there are the steps of the inner loop. But these are at most $\lambda - 1$ — not counting the first one with 0 — because every m_i for $i = 0, \dots, \lambda - 1$ appears exactly once. Therefore the main part uses $A_2 = q^r - 2 + \lambda - 1$ addition steps and no q-steps ($Q_2 = 0$).

LEMMA 3.27. *Let $m \in \mathbb{N}$. Then a q-addition chain for m can be computed in at most $Q = r\lfloor \log_{q^r} m \rfloor$ q-steps (only appearing in precomputation) and $A = q^r + \lfloor \log_{q^r} m \rfloor - 2$ further addition steps (only appearing after precomputation). We therefore get the length L of the q-addition chain as*

$$L \le A + Q = q^r + (r+1)\lfloor \log_{q^r} m \rfloor - 2.$$

COROLLARY 3.28. *Let $m, q \in \mathbb{N}$, $q \geq 2$, and $\mu = \log_q m$. There is a q-addition chain for m of length at most*

$$\mu + \frac{\mu}{\log_q \mu}(1 + \frac{q}{\log_q \mu} + \frac{2\log_q \log_q \mu}{\log_q \mu - 2\log_q \log_q \mu}) \leq \mu + \frac{\mu}{\log_q \mu}(1 + o(1)).$$

PROOF. We have $l_q(m) \leq L = Q + A$. But $Q = r\lfloor \log_{q^r} m \rfloor \leq r \cdot \frac{1}{r}\mu = \mu$ and $A = q^r + \lfloor \log_{q^r} m \rfloor - 2 < q^r + \frac{1}{r}\mu$ for any $r \in \mathbb{N}$.

Select $r = \lfloor \log_q \mu - 2\log_q \log_q \mu \rfloor + 1$. Then we get:

$$\begin{aligned}
A &< \frac{q^{1+\log_q \mu}}{q^{2\log_q \log_q \mu}} + \frac{\mu}{\lfloor \log_q \mu - 2\log_q \log_q \mu \rfloor + 1} \\
&\leq \frac{q\mu}{(\log_q \mu)^2} + \frac{\mu}{\log_q \mu - 2\log_q \log_q \mu} \\
&= \frac{q\mu}{(\log_q \mu)^2} + \frac{\mu}{\log_q \mu} \cdot \frac{\mu}{\log_q \mu - 2\log_q \log_q \mu}\frac{\log_q \mu}{\mu} \\
&= \frac{q\mu}{(\log_q \mu)^2} + \frac{\mu}{\log_q \mu} \cdot \varepsilon \\
&= \frac{\mu}{\log_q \mu}(\frac{q}{\log_q \mu} + \varepsilon),
\end{aligned}$$

where

$$\begin{aligned}
\varepsilon &= \frac{\mu}{\log_q \mu - 2\log_q \log_q \mu}\frac{\log_q \mu}{\mu} \\
&= \frac{\log_q \mu}{\log_q \mu - 2\log_q \log_q \mu} \\
&= \frac{\log_q \mu - 2\log_q \log_q \mu + 2\log_q \log_q \mu}{\log_q \mu - 2\log_q \log_q \mu} \\
&= 1 + \frac{2\log_q \log_q \mu}{\log_q \mu - 2\log_q \log_q \mu}.
\end{aligned}$$

Hence, we have

$$\begin{aligned}
A &\leq \frac{\mu}{\log_q \mu}(1 + \frac{q}{\log_q \mu} + 2\frac{\log_q \log_q \mu}{\log_q \mu - 2\log_q \log_q \mu}) \\
&= \frac{\mu}{\log_q \mu}(1 + o(1))
\end{aligned}$$

which proves the upper bound. □

Corollary 3.29. *Let $k \in \mathbb{N}$. Then a q-addition chain containing the positive integers $m^{(1)}, \ldots, m^{(k)}$ can be computed in at most*

$$\begin{aligned} Q &= r\lfloor \log_{q^r} m \rfloor \text{ } q\text{-steps and} \\ A &= k(q^r - 2) + \sum_{1 \le i \le k} \lfloor \log_{q^r} m^{(i)} \rfloor \le k(q^r + \lfloor \log_{q^r} m \rfloor - 2) \text{ further steps,} \end{aligned}$$

where $m = \max_{1 \le i \le k} m^{(i)}$.

Proof. Splitting the number of steps in precomputation and further steps we have to do $Q = r\lfloor \log_{q^r} m \rfloor$ q-steps only once. For each $i \in \{1, \ldots, k\}$ we have to do $A_i = q^r - 2 + \lfloor \log_{q^r} m^{(i)} \rfloor$ further steps. □

3.6. Addition chain algorithms using data compression. Many algorithms for addition chains use the same basic idea: they (pre)compute some elements that hopefully will be reused more than once. Let $\mathcal{D}$ be the set of all precomputed elements again. Then the following is obvious: if $\mathcal{D}$ is built of such q-ary subsequences which often appear in $(m)_q$ we can reduce the number of unnecessarily precomputed elements of the addition chain and nevertheless use the advantages of it.

But the problem to extract the most probable subsequences from a given sequence also appears in data compression. In this context subsequences that often appear in a sequence should be compressed and encoded by fewer bits than others (see Bocharova & Kudryashov 1995). Therefore it is worth while looking at data compression methods. We present different techniques to find a proper set $\mathcal{D}$ to (pre)compute: the first one is the method of Ziv & Lempel (1978) that was first used for addition chains by Yacobi (1991). The second one was suggested by Bocharova & Kudryashov (1995) and is based on an algorithm of Tunstall (1968) (given in Jelinek & Schneider 1972) to get a proper set $\mathcal{D}$. We finally construct another algorithm which extracts only the subsequences that can be found more than once in a given sequence.

Notation 3.30. *Let $m \in \mathbb{N}$ and $(m)_2 = (m_{\lambda-1}, \ldots, m_0)$ be the binary representation of m. We call m_i for $0 \le i < \lambda$* a bit. *$(m_{i+s}, \ldots, m_i)$ with $s \ge 1$ is called* a bitstring.

3.7. Data Compression according to Ziv & Lempel.

The main ideas. The q^r-ary method builds the set $\mathcal{D}$ of all subsequences $1, 2, \ldots, q^r - 1$. But often it is not necessary to precompute all elements $d \in \{1, 2, \ldots, q^r - 1\}$ because some d do not appear further along in the addition chain for m. Yacobi (1991) therefore suggests to build $\mathcal{D}$ during the construction of the addition chain. The second property of the q^r-ary method is the fact that all $d \in \mathcal{D}$ have a fixed predetermined length of r according to the q-ary representation of m. Yacobi (1991) does not impose this restriction.

His two main ideas have also been established within the compression algorithm of Ziv & Lempel (1978). Therefore Yacobi uses a modified version of Ziv & Lempel's algorithm to determinate $\mathcal{D}$: Parse $(m)_q$ from one end to the other and create a binary 'compression' tree where the path from the root to a node is a subsequence $(d)_q$ of the exponent $(m)_q$ and this node contains d.

The algorithm. We describe the algorithm informally and use no explicit data structure. We concentrate on $q = 2$ and $r = 1$ like Yacobi (1991) but we scan $(m)_2$ from left to right.

ALGORITHM 3.31. `yacobi`
Input: $m \in \mathbb{N}$ *with* $m = (m_{\lambda-1}, \ldots, m_0)_2$ *and* $\lambda = \lfloor \log_2 m \rfloor + 1$.
Output: $1 = a_0, \ldots, a_L = m$ *an addition chain for* m.

1. *Set* $a_0 = d_0 = 1$ *and* $\mathcal{D} = \{d_0\}$. *Set* $i = \lambda - 2$ *and* $j = 1$.
2. *While* $i \geq 0$ *do*
   3. *If* $m_i = 0$ *then evaluate* $a_j = a_{j-1} + a_{j-1}$. *Set* $j = j + 1$ *and* $i = i - 1$.
   4. *else the next sequence* $\mathcal{S}$ *beginning with* 1 *has been detected: Let this sequence be* $\mathcal{S} = (m_i, \ldots, m_{i-s+1})$ *with* $s = \max\{s' : \exists d \in \mathcal{D}$ *with* $(d)_2 = (m_i, \ldots, m_{i-s'+1})\}$. *Do Steps 5–7.*
      5. *Compute* $a_{j+k} = a_{j+k-1} + a_{j+k-1}$ *for all* $k = 0, \ldots, s - 1$. *Set* $j = j + s$ *and* $i = i - s + 1$.
      6. *If* $i = 0$ *then compute* $a_j = a_{j-1} + d$ *with* $(d)_2 = \mathcal{S}$ *and* $d \in \mathcal{D}$. *Set* $j = j + 1$ *and* $i = i - 1$.
      7. *else set* $i = i - 1$ *and do Steps 8–11.*
         8. *[Comment: actualize* $\mathcal{D}$*.] Let* $d \in \mathcal{D}$ *with* $(d)_2 = \mathcal{S}$. *Set* $d_1 = d + d$. *Set* $k' = 1$.

9. *If $m_i = 1$ then set $d_2 = d_1 + d_0$ and set $a_j = d_2$ and $j = j+1$. Set $k' = 2$.*
10. *Set $\mathcal{D} = \mathcal{D} \cup \{d_{k'}\}$.*
11. *[Comment: Add a new element to the addition chain.] Set $a_j = a_{j-1} + a_{j-1}$ and $a_{j+1} = a_j + d_{k'}$. Set $j = j + 2$.*

12. *Return the addition chain built by concatenating $a_0, \dots, a_{j-1}$ and $\mathcal{D}$.*

An example. We give an example to illustrate this algorithm in Figure 3.2. This example with $m = 5541$ shows how the bitstring for m is scanned, which values are computed for the addition chain and which values are stored. $\mathcal{D}$ is illustrated by a binary tree according to the stored sequences.

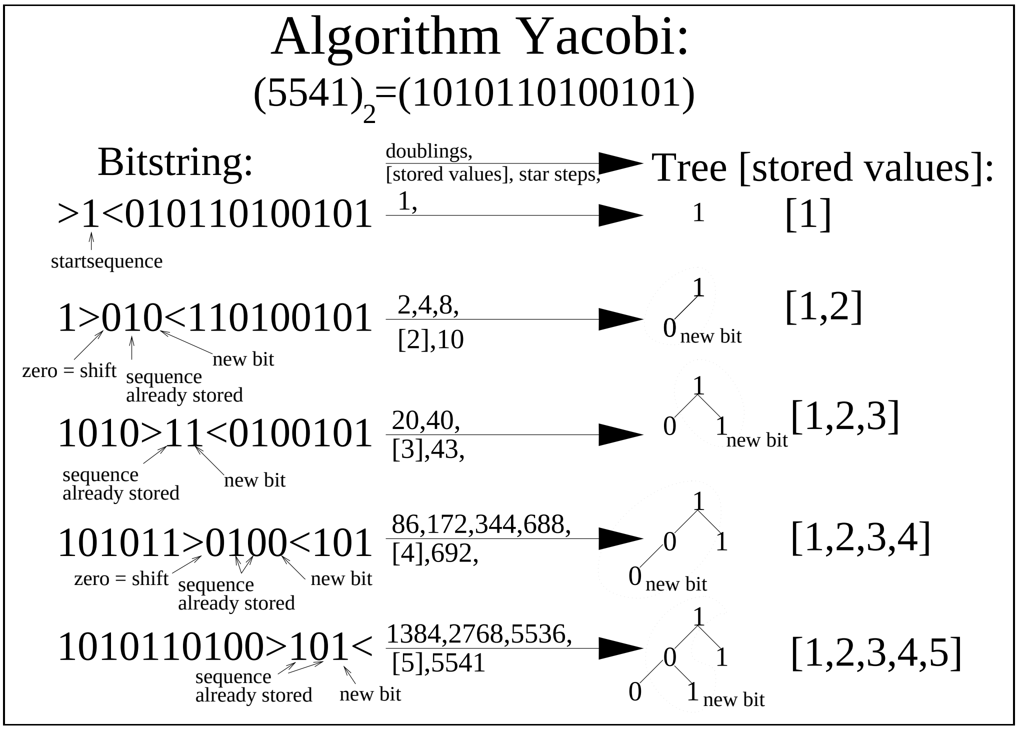

Figure 3.2: A schematic illustration of Algorithm `yacobi` on input $m = 5541$.

Correctness.

LEMMA 3.32. *Algorithm* `yacobi` *computes an addition chain for given $m \in \mathbb{N}$.*

PROOF. To prove partial correctness we first define an invariant for the loop given in Steps 2–11:

$a_0, \ldots, a_{j-1}$ together with $\mathcal{D}$ is an addition chain for $(m_{\lambda-1}, \ldots, m_{i+1})$.

After Step 1 $a_0 = d_0 = 1$ and $m_{\lambda-1} = 1$. Therefore the invariant holds before entering the loop.

Assume this is also true for i: $a_0, \ldots, a_{j-1}, \mathcal{D}$ is an addition chain for $(m_{\lambda-1}, \ldots, m_{i+1})_2$. Because of Step 3 we have $(a_j)_2 = (m_{\lambda-1}, \ldots, m_{i+1}, 0)$ and $m_i = 0$. Because a_{j-1} has already been in the addition chain and i is decremented the invariant holds.

If $m_i = 1$ we start with a new sequence $\mathcal{S}$ in $(m)_2$. A sequence is a bitstring starting with '1' that has already be found in $(m)_2$ before. s is the length of the maximal sequence in $(m)_2$ starting at position i. After Step 5 we have s new elements with $(a_{j-1})_2 = (m_{\lambda-1}, \ldots, m_{i+s}, 0, \ldots, 0)$ and $a_0, \ldots, a_{j-1}, \mathcal{D}$ is an addition chain for $(m_{\lambda-1}, \ldots, m_{i+s}, 0, \ldots, 0)$. If there are no more bits left in $(m)_2$, we can add $a_j = a_{j-1} + d$ with $d \in \mathcal{D}$ and $(d)_2 = \mathcal{S}$. We have $(a_j)_2 = (m_{\lambda-1}, \ldots, m_{i+s}, \ldots, m_i)$ because we already calculated d. With $i = i - 1$ the invariant holds.

In the other case we have $i = i - 1$ and we can inspect one further m_i. After Step 10 we have computed $(d_{k'})_2 = (\mathcal{S}, m_i)$ and $d_{k'} \in \mathcal{D}$ using only elements evaluated so far. Therefore in Step 11 we can compute $a_j = a_{j-1} + d_{k'}$ with $(a_j)_2 = (m_{\lambda-1}, \ldots, m_{i+s+1}, m_{i+s}, \ldots, m_i)$. With respect to the decrementation of i the invariant holds after the loop.

We therefore return an addition chain for m. Because i is decreased in order to the scanned elements of $(m)_2$ the algorithm terminates after scanning all $\lambda + 1$ bits of $(m)_2$. This shows total correctness. $\square$

Number of steps. The number of doublings to scan $(m)_2$ is $\lambda - 1 = \lfloor \log_2 m \rfloor$ because any decrementation of i is connected with a doubling step and $i = 0, \ldots, \lambda - 2$.

The number of further steps is not as simple to see as the previous one. To give a transparent discussion of this topic we use the data structure of a tree. Then the following remark is obvious:

REMARK 3.33. *The $d \in \mathcal{D}$ calculated within Algorithm* `yacobi` *can be arranged in a binary tree according to the rules:*

1. *$d_0 = 1$ is declared to be the root.*

2. *$2d + j, j \in \{0, 1\}$ can be computed from d by doing a doubling (for $j = 0$) or by doing a doubling and a star step ($j = 1$). Let $2d$ be the left and $2d + 1$ be the right son of d.*

Then we can count one doubling step for each son and one further star step for any right son in the tree.

Additionally we have to add a star step to the addition chain for any new sequence $\mathcal{S}$. The number of sequences — which we denote by S and which is just the number of sons in the tree — is bounded by $\nu_2(m) - 1$ because a new sequence has to start with '1'. We can summarize:

LEMMA 3.34. *Let $m \in \mathbb{N}$ and $\lambda = \lfloor \log_2 m \rfloor + 1$. Then Algorithm* `yacobi` *produces an addition chain for m with $D = \lambda - 1 + S$ doublings and $A = R + S$ star steps where $S \in \mathbb{N}_0$ is the number of sequences generated in the algorithm and $R \in \mathbb{N}_0$ is the number of different sequences with last bit 1. We have $R \le S \le \nu_2(m) - 1$.*

Average case and examples. We fix some $k \in \mathbb{N}$. Then $\Omega = \{m \in \mathbb{N}\colon m < 2^k\}$ is a probability space. Interprete $\mathcal{D}$ as in Remark 3.33. For a randomly chosen element of Ω the tree is expected to be balanced (see Yacobi 1991). The question is: What is the number S of generated sequences on the average?

First we have to estimate the number of zeros (Step 3) that appear before a new sequence $\mathcal{S}$ starts (Step 4). In a random sequence both '1' and '0' occur with probability $\frac{1}{2}$. Therefore j times '0' in front of a new sequence has the probability $\frac{1}{2^{j+1}}$ for $j = 0, 1, \ldots$. We get the expected number of '0' in front of a new sequence with $\sum_{j=0}^{k-1} j 2^{-(j+1)} \le 1$.

Any node of the tree represents one sequence and we have 2^i nodes at depth i representing sequences of length $i + 1$. Together with the leading zero and not counting the root node, we get $\lambda - 1 = \sum_{i=1}^{h} (i + 2) \cdot 2^i = (h+1) \cdot 2^{h+1} - 2$, where h is the depth of the tree. We get $S = 2^{h+1} - 2$ nodes (without root) and $\lambda - 1 = (S + 2) \log_2 (S + 2) - 2$. According to Yacobi (1991) we have $S = \frac{\lambda}{\log_2 \lambda}(1 + o(1))$ on average. Because on average $R = \frac{1}{2} S$ we get the result:

LEMMA 3.35. *Let $m \in \mathbb{N}$ and $\mu = \log_2 m$. On the average Algorithm* `yacobi` *computes an addition chain for m with*

$$\begin{aligned} D_{ave} &= \lambda - 1 + S = \lfloor \mu \rfloor + \frac{\mu}{\log_2 \mu}(1 + o(1)) \text{ doublings and} \\ A_{ave} &= S + R = \frac{3}{2} \frac{\mu}{\log_2 \mu}(1 + o(1)) \text{ star steps.} \end{aligned}$$

Finally we give some examples:

EXAMPLE 3.36. *1.* $m = 2^k - 1$: *Then there are $h-1$ sequences $s_1, \ldots, s_{h-1}$ with length $|s_i| = i+1$ and therefore we have $\sum_{i=1}^{h-1} s_i = 2 + \cdots + h \leq \lambda - 1 = k-1$. But with $\lambda - 1 \geq \sum_{i=2}^{h} i = \frac{h}{2}(h-1) - 1$ we get $h \leq \frac{1}{2} + \frac{1}{2}\sqrt{1+8\lambda}$ and $R = S = h$. Therefore $A = 2h \leq 1 + \sqrt{1+8\lambda} = 1 + \sqrt{1+8k}$ and $D \leq \lfloor \log_2 m \rfloor + \frac{1}{2} + \frac{1}{2}\sqrt{1+8\lambda} = k - 1 + \frac{1}{2}(1 + \sqrt{1+8k})$.*

2. $m = 2^k$: *Then no sequence is generated and $R = S = 0$. It follows $A = 0$ and $D = k$.*

3. $(m)_2 = (1)_2(2)_2(3)_2 \ldots (2^k - 1)_2$: *The tree has depth $h = k - 1$ (not counting the root node) and $\lambda - 1 = \sum_{i=1}^{k} i \cdot 2^{i-1} - 1 = 2^k(k-1)$. Because there are $S = 2^{k-1+1} - 2$ nodes (without root) we have $R = 2^{k-1} - 1$ and $R+S = 3(2^{k-1}-1)$. Evaluating $k(\lambda)$ with $\lambda - 1 = 2^k(k-1) = 2(k-1)\cdot 2^{k-1}$ we have with $k' = k - 1$ and $\lambda' = \frac{\lambda - 1}{2}$: $k = k' + 1 \geq \frac{k'}{\log_2(k' + \log_2 k')} + \frac{\log_2 k'}{\log_2(k' + \log_2 k')} = \frac{\log_2(k'2^{k'})}{\log_2 \log_2(k'2^{k'})} = \frac{\log_2 \lambda'}{\log_2 \log_2 \lambda'}$. Then*

$$
\begin{aligned}
A &= S + R = 3(2^{k-1} - 1) \\
&= 3(\frac{\lambda - 1}{2(k-1)} - 1) \\
&< 3(\frac{\lambda'}{k'} - 1) \\
&< 3\left(\frac{\lambda'}{\frac{\log_2 \lambda'}{\log_2 \log_2 \lambda'} - 1} - 1\right) \\
&< 3\frac{\lambda' \log_2 \log_2 \lambda'}{\log_2 \lambda' - \log_2 \log_2 \lambda'} \text{ and} \\
D &= \lambda - 1 + S = 2^k(k-1) + 2^k - 2 \\
&= 2^k k - 2 \\
&= \lambda - 1 + 2\frac{\lambda - 1}{2(k-1)} - 1 \\
&< \lambda + 2\frac{\lambda'}{k'} - 2 \\
&\leq \lambda + 2\frac{\lambda' \log_2 \log_2 \lambda'}{\log_2 \lambda' - \log_2 \log_2 \lambda'} - 2.
\end{aligned}
$$

Worst case. The worst case for `yacobi` is given if the number of sequences S is as large as possible because $R \leq S$. But there can only be 2^{i-1} sequences of length i for $i = 1, 2, \ldots$. S gets smaller is there are leading zeros. The last sequence has to be a new sequence. But then the worst case is given by

$(m)_2 = (1)_2(2)_2(3)_2 \ldots (2^k - 1)_2$. We already have found an upper bound in this case in Example 3.36 which leads to the following corollary:

COROLLARY 3.37. *Let $m \in \mathbb{N}$ and $\lambda = \lfloor \log_2 m \rfloor + 1$ and $\lambda' = \frac{\lambda - 1}{2}$. Algorithm* `yacobi` *computes an addition chain for m of length at most*

$$\lambda + 5\frac{\lambda' \log_2 \log_2 \lambda'}{\log_2 \lambda'}(1 + \frac{\log_2 \lambda'}{\log_2 \lambda' - \log_2 \log_2 \lambda'}) \leq \lambda + \frac{5}{2}\frac{\lambda \log_2 \log_2 \lambda}{\log_2 \lambda - \log_2 \log_2 \lambda}(1 + o(1)).$$

PROOF. We have $L = A + D = S + R + \lambda - 1 + S < \lambda + \frac{5}{2}S$ because $S = 2R$. But

$$\begin{aligned} S &= 2(2^{k-1} - 1) = 2\frac{\lambda - 1}{2(k-1)} - 2 \\ &= 2\frac{\lambda'}{k'} - 2 \\ &< 2\frac{\lambda' \log_2 \log_2 \lambda'}{\log_2 \lambda' - \log_2 \log_2 \lambda'} - 2 \\ &< 2\frac{\lambda' \log_2 \log_2 \lambda'}{\log_2 \lambda'}(1 + \frac{\log_2 \lambda'}{\log_2 \lambda' - \log_2 \log_2 \lambda'}). \end{aligned}$$

Using the fact that $\log_2 \lambda' = \log_2 \frac{\lambda - 1}{2} < \log_2 \lambda - 1$ completes the proof. □

3.8. Addition chain algorithms with proper sets.

The basic idea. Algorithm `yacobi` concentrates on analyzing a given bitstring from left to right. But it could be helpful to skip the dependence between scan direction and (pre)computation. This idea can be realized using an algorithm that was developed by Tunstall (1968). The algorithm uses so called *'variable–to–fixed' length codes* and is given due to Bocharova & Kudryashov (1995). Therefore this section has two parts: the first one introduces the parsing algorithm for $(m)_2$ that was developed by Tunstall (1968). The second one uses this parsing of $(m)_2$ to create an addition chain for m. In the following we concentrate on the binary representation of m again.

Tunstall's parsing algorithm. We will give a modified version of Tunstall's algorithm. The original is reprinted in the work of Jelinek & Schneider (1972). Originally Tunstall's algorithm was developed to compute a so called *complete and proper* set (for definitions see Jelinek & Schneider 1972).

ALGORITHM 3.38. `tunstall`
Input: $m, r \in \mathbb{N}, r \geq 2$, *with* $(m)_2 = (m_{\lambda-1}, \ldots, m_0)_2$ *and* $\lambda = \lfloor \log_2 m \rfloor + 1$ *and* r *a selectable parameter.*
Output: $\mathcal{D} \subset \mathbb{N}$ *a set with the following properties:* $\#\mathcal{D} = r$, $1 \in \mathcal{D}$ *and* $0 \notin \mathcal{D}$. *If* $d \in \mathcal{D}$ *with* $(d)_2 = (d_{\lambda-1}, \ldots, d_1, d_0)$ *with* $\lambda > 1$ *then* $d' \in \mathcal{D}$ *with* $(d')_2 = (d_{\lambda'-1}, \ldots, d_1)$ *and* d' *is the most probable element in* $\mathcal{D} \setminus \{2d', 2d'+1\}$ *according to* $(m)_2$.

1. *Set* $d_0 = 1$ *and compute* $d' = d_0 + d_0$ *and* $d'' = d' + d'$. *Set* $\mathcal{D} = \{d_0, d', d''\}$ *and* $r' = 2$.
2. *While* $(r' < r)$ *repeat Steps 3–5.*
   3. *Let* $(m)_2 = (\tilde{m}_1, 0^{z_1}, \ldots, \tilde{m}_k, 0^{z_k})$ *with* $\tilde{m}_i$ *specified as follows: For* $1 \leq i < k$ *there exists* $d \in \mathcal{D}$ *with* $(d)_2 = \tilde{m}_i$ *and* $2d, 2d+1 \notin \mathcal{D}$. *For* $\tilde{m}_k$ *exists* $d \in \mathcal{D}$ *with* $(d)_2 = \tilde{m}_k$ *and if* $z_k > 0$ *then* $2d \notin \mathcal{D}$.
   4. *Let* $d \in \mathcal{D}$ *be the element of* $\mathcal{D}$ *for which* $(d)_2$ *appears most often in* $\tilde{m}_1, \ldots, \tilde{m}_k$. *If there are two or more elements of* $\mathcal{D}$ *that satisfy this condition then choose the one of maximal value.*
   5. *Compute* $d' = d + d$ *and* $d'' = d' + d_0$. *Set* $\mathcal{D} = \mathcal{D} \cup \{d', d''\}$. *Set* $r' = r' + 1$.
6. *Return* $\mathcal{D}$.

The algorithm above calculates parts of an addition chain for m. There are $r-1$ doublings and also $r-1$ star steps. Indeed the algorithm adds two new elements every round in Step 5 and there are $r-2$ rounds. Therefore at most $2(r-2)+3 = 2r-1$ elements are generated.

An example. We illustrate Algorithm `tunstall` by giving an example for $m = 5541, r = 3$ in Figure 3.3. Within the example we use a tree for $\mathcal{D}$.

Generating an addition chain from a given set $\mathcal{D}$. We are now ready to give an algorithm that computes an addition chain for m if the elements $d_i, i \in \mathcal{D}$, have already been computed.

ALGORITHM 3.39. `bocharova`
Input: $m \in \mathbb{N}$ *with* $m = (m_{\lambda-1}, \ldots, m_0)_2$ *and* $\lambda = \lfloor \log_2 m \rfloor + 1$ *and* $\mathcal{D}$ *as calculated in Algorithm* `tunstall`.
Output: $1 = a_0, \ldots, a_L = m$, *an addition chain for* m.

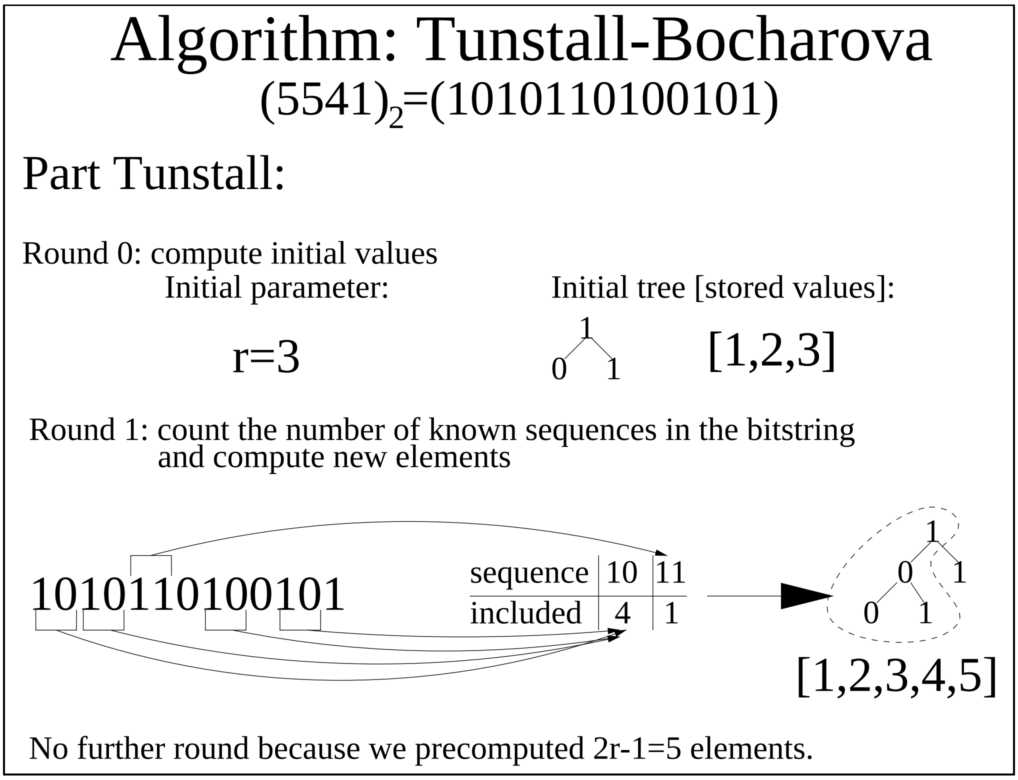

Figure 3.3: A schematic illustration of Algorithm `tunstall` on input $m = 5541$.

1. *Let* $\mathcal{S} = (m_{\lambda-1}, \ldots, m_{\lambda-s_1})$ *with* $s_1 = \max\{s' \colon \exists d \in \mathcal{D}$ *with* $(d)_2 = (m_\lambda, \ldots, m_{\lambda-s'+1})\}$*. Set* $a_0 = d$ *with* $d \in \mathcal{D}$ *and* $(d)_2 = \mathcal{S}$*. Set* $i = \lambda - s_1$ *and* $j = 1$*.*

2. *While* $i \geq 0$ *do*

   3. *If* $m_i = 0$ *then evaluate* $a_j = a_{j-1} + a_{j-1}$*. Set* $j = j+1$ *and* $i = i-1$*.*

   4. *else the next sequence* $\mathcal{S}$ *beginning with '1' has been detected: Let this sequence be* $\mathcal{S} = (m_i, \ldots, m_{i-s+1})$ *with* $s = \max\{s' \colon \exists d \in \mathcal{D}$ *with* $(d)_2 = (m_i, \ldots, m_{i-s'+1})\}$*. Do Steps 5–6.*

      5. *Compute* $a_{j+k} = a_{j+k-1} + a_{j+k-1}$ *for all* $k = 0, \ldots, s-1$*. Set* $j = j + s$ *and* $i = i - s + 1$*.*

      6. *[Comment: Calculate the next element of the addition chain.] Let* $d \in \mathcal{D}$ *with* $(d)_2 = S$*. Compute* $a_j = a_{j-1} + d$ *and set* $j = j + 1$ *and* $i = i - 1$*.*

7. *Return the addition chain built by concatenating* $a_0, \ldots, a_{j-1}$ *and* $\mathcal{D}$*.*

Continued example. We continue the example given in Figure 3.3. The second part computes an addition chain by using $\mathcal{D}$. The bitstring for m is parted into the sequences given by $\mathcal{D}$. This is illustrated in Figure 3.4.

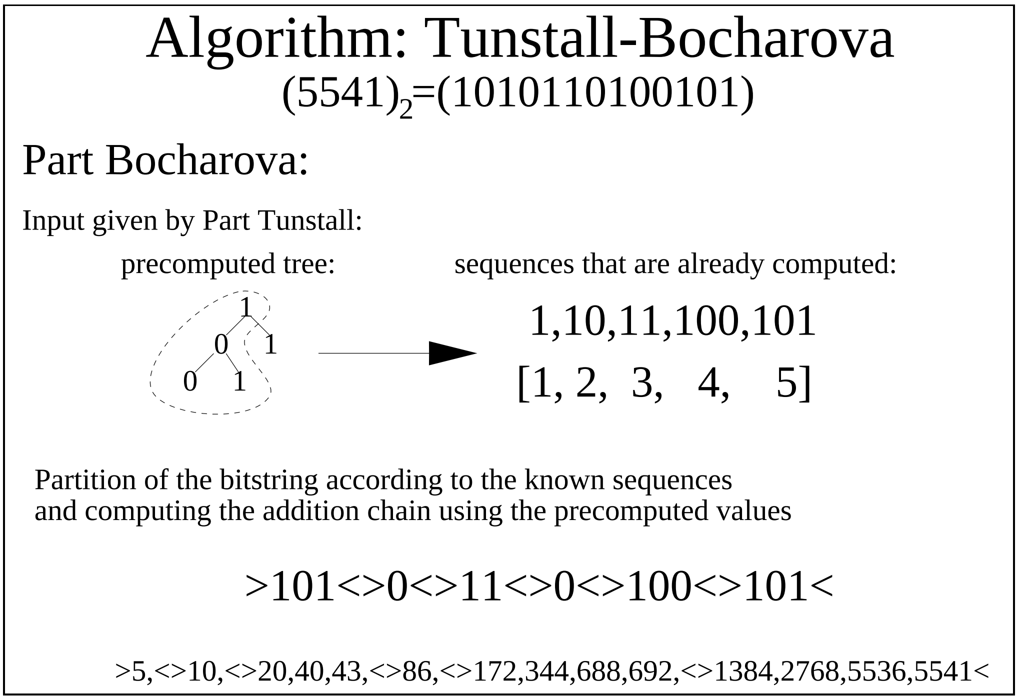

Figure 3.4: A schematic illustration of Algorithm `bocharova` on input $m = 5541$ and $\mathcal{D} = \{1, 2, 3, 4, 5\}$. The computation of $\mathcal{D}$ is illustrated in Figure 3.3.

Correctness.

LEMMA 3.40. *Algorithm* `bocharova` *calculates an addition chain for m correctly.*

PROOF. The proof is similar to the given proof of Algorithm `yacobi` when we use the fact that $\mathcal{D}$ can be ordered in a binary tree according to Remark 3.33. □

Number of steps. There are $\lambda - s_1$ doublings, where s_1 is the length of the first sequence in $(m)_2$, because we do a doubling for all m_i except for the bits of the first sequence. Let S be the number of sequences in $(m)_2$ according to the algorithm. We have $S \leq \nu_2(m)$ because every sequence starts with '1'. Then there are $S - 1$ additions with elements of $\mathcal{D}$ because we concatenate only the second to the last sequence with given elements of the addition chain.

Lemma 3.41. *Let $m \in \mathbb{N}$ and $\lambda = \lfloor \log_2 m \rfloor + 1$. Then Algorithm* `bocharova` *in connection with Algorithm* `tunstall` *produces an addition chain for m with $D = r + \lambda - 1 - s_1$ doublings and $A = r + S - 2$ star steps where $S \in \mathbb{N}_0$ is the number of sequences in $(m)_2$ and s_1 is the length of the first sequence of $(m)_2$. $r \in \mathbb{N}$ is a selectable parameter to determine the number of elements (pre)computed by Algorithm* `tunstall`. *We have $S \leq \nu_2(m)$ and $s_1 \leq \lambda$.*

Average Case. Because our results depend on the special form of $(m)_2$ we analyze the average case to be able to compare with `yacobi`.

Fix $k \in \mathbb{N}$ and let $\Omega = \{m \in \mathbb{N}\colon m < 2^k\}$ be a probability space. For an arbitrary chosen $m \in \Omega$ the probability for one bit in $(m)_2$ to be '1' is $\frac{1}{2}$. Let $\mathcal{D}$ and r as above and $\mu = \log_2 m$. Then — according to Bocharova & Kudryashov (1995) — we have

$$\begin{aligned} A_{ave} &= \frac{\mu}{2 + \frac{\log_2 r + \log_2 \frac{1}{2}}{1}} + r \\ &= \frac{\mu}{\log_2 r + 1} + r, \end{aligned}$$

and with $r = \lfloor \frac{\mu}{(\log_2 \mu)^2} \rfloor$ we get

$$\begin{aligned} A_{ave} &\leq \frac{\mu}{\log_2 (\frac{\mu}{(\log_2 \mu)^2}) + 1} + \frac{\mu}{(\log_2 \mu)^2} \\ &\leq \frac{\mu}{\log_2 \mu - 2 \log_2 \log_2 \mu} + \frac{\mu}{(\log_2 \mu)^2} \\ &= \frac{\mu}{\log_2 \mu}(1 + \frac{\log_2 \log_2 \mu}{\log_2 \mu - 2 \log_2 \log_2 \mu} + \frac{1}{\log_2 \mu}) \\ &= \frac{\mu}{\log_2 \mu}(1 + o(1)). \end{aligned}$$

With this choice of r we get the average number of doublings as

$$\begin{aligned} D_{ave} &= r + \lambda - 1 - s_1 \\ &< \lfloor \frac{\mu}{(\log_2 \mu)^2} \rfloor + \lfloor \mu \rfloor \\ &\leq \lfloor \mu \rfloor + \frac{\mu}{(\log_2 \mu)^2}. \end{aligned}$$

LEMMA 3.42. *Let $m \in \mathbb{N}$ and $\mu = \log_2 m$. On the average Algorithm* `bocharova-tunstall` *computes an addition chain for m with*

$$\begin{aligned} D_{ave} &< \lfloor \mu \rfloor + \frac{\mu}{(\log_2 \mu)^2} \text{ doubling steps and} \\ A_{ave} &= \frac{\mu}{\log_2 \mu}\left(1 + \frac{\log_2 \log_2 \mu}{\log_2 \mu - 2\log_2 \log_2 \mu} + \frac{1}{\log_2 \mu}\right) \\ &= \frac{\mu}{\log_2 \mu}(1 + o(1)) \text{ star steps.} \end{aligned}$$

Worst case. In the sequel we use the short form `bocharova` instead of `bocharova-tunstall`. To estimate the worst case for `bocharova` we first fix r. We have $D < r + \lambda - 1$. So we can concentrate on $A = r + S - 2$. To get an upper bound for A we have to estimate $S(r)$.
Algorithm `bocharova` tries to use the longest sequences that can be found in $\mathcal{D}$. There are $2r-1$ elements in $\mathcal{D}$. Let k be the length of the longest sequence. There are at most 2^{i-1} elements of length i in $\mathcal{D}$ for $i = 1, 2, \ldots$. Minimizing k we get the equation $\sum_{i=1}^{k} i \cdot 2^{i-1} = 2^k(k-1) + 1 = 2r - 1$ and with $r' = r - 1$ and $k' = k - 1$ we have $r' = 2^{k'}k'$. Then we have $k' \geq \frac{\log_2 r'}{\log_2 \log_2 r'}$ (see Example 3.36). We have $S \leq \lceil \frac{\lambda}{k'} \rceil$ and therefore $A = r + S - 2 \leq r + \frac{\lambda}{k'} \leq r + \lambda \frac{\log_2 \log_2 r'}{\log_2 r'}$.

COROLLARY 3.43. *Let $m, r \in \mathbb{N}$ and $\mu = \log_2 m$. Then Algorithm* `bocharova` *computes an addition chain for m of length at most*

$$\mu + \frac{\mu \log_2 \log_2 \mu}{\log_2 \mu}(2 + o(1)).$$

PROOF. Choose $r = \lfloor \frac{\mu}{(\log_2 \mu)^2} \rfloor$ again. We have $\lambda = \lfloor \log_2 m \rfloor + 1 < \mu + 1$ and

$$\begin{aligned} A &\leq r + \lambda \frac{\log_2 \log_2 (r-1)}{\log_2 (r-1)} \\ &\leq \frac{\mu}{(\log_2 \mu)^2} + (\mu + 1) \frac{\log_2 \log_2 \mu - \log_2(2 \log_2 \log_2 \mu)}{\log_2 \mu - 2\log_2 \log_2 \mu - 1} \\ &< \frac{\mu \log_2 \log_2 \mu}{\log_2 \mu}\left(\frac{1}{\log_2 \mu \log_2 \log_2 \mu} + \frac{\mu + 1}{\mu} \frac{\log_2 \mu}{\log_2 \mu - 2 \log_2 \log_2 \mu - 1}\right) \\ &< \frac{\mu \log_2 \log_2 \mu}{\log_2 \mu}(2 + o(1)) \text{ and} \\ D &< r + \lambda - 1 \\ &\leq \mu + \frac{\mu}{(\log_2 \mu)^2} \\ &= \mu + \frac{\mu \log_2 \log_2 \mu}{\log_2 \mu} \frac{1}{\log_2 \mu \log_2 \log_2 \mu}. \quad \square \end{aligned}$$

3.9. A new algorithm based on data compression.

Basic idea. The basic idea of this new algorithm is to detect the longest bitstring in front of the actual position that has already been calculated. This idea can also be found in Algorithm `yacobi`. But in this new algorithm we do not add a new bit to the known bitstring and store it. We store the concatenation of the bitstring already scanned and the detected bitstring. To find regularities within the bitstring we concatenate parts of the bitstring already stored and the detected bitstring and store it. To avoid unnecessary calculations we evaluate and add only those elements to the addition chain which are known to be reused. To realize this we need two scans of $(m)_2$.

The algorithm.

ALGORITHM 3.44. `lookahead`
Input: $m \in \mathbb{N}$ *with* $(m)_2 = (m_{\lambda-1}, \ldots, m_0)$ *and* $\lambda = \lfloor \log_2 m \rfloor + 1$.
Output: $1 = a_0, \ldots, a_L = m$, *an addition chain for* m.

Part A: First scan to find the bitstrings that are used more than once.

1. *Let* $(m)_2 = (m_{\lambda-1}, 0^z, 1\ldots)$. *Set* $\mathcal{S}_l = (m_{\lambda-1}, 0^z)$ *and* $\mathcal{S}_a = \mathcal{S}_l$. *Set* $\Sigma = \{m_{\lambda-1}\}$ *and* $\Sigma' = \Sigma$.
2. *While* $\mathcal{S}_a \neq m$ *repeat*
   3. *Let* $(m)_2 = (\mathcal{S}_a, \mathcal{S}_n, 0^z, 1\ldots)$ *with* $\mathcal{S}_n$ *specified as follows:* $\mathcal{S}_n \in \Sigma$ *and* $\mathcal{S}_n = \max\{\mathcal{S}' \in \Sigma \colon m = (\mathcal{S}_a, \mathcal{S}', 0^{z'}, 1\ldots)\}$.
   4. *Set* $\Sigma' = \Sigma' \cup \mathcal{S}_n$ *and* $\Sigma = \Sigma \cup \{(\mathcal{S}_a, \mathcal{S}_n), (\mathcal{S}_l, \mathcal{S}_n)\}$.
   5. *Set* $\mathcal{S}_a = (\mathcal{S}_a, \mathcal{S}_n, 0^z)$ *and* $\mathcal{S}_l = (\mathcal{S}_n, 0^z)$.
6. *[Comment: Evaluate the bitstrings of* Σ' *which are reused.] Set* $\mathcal{D} = \{1\} \cup \{d \in \mathbb{N} \colon \exists \mathcal{S}, \mathcal{S}' \in \{0,1\}^*$ *not the empty word with* $(\mathcal{S}, \mathcal{S}') \in \Sigma'$ *and* $(d)_2 = (\mathcal{S}, 0)$ *or* $(d)_2 = (\mathcal{S}, 1)\}$. *Compute all* $d \in \mathcal{D}$.

Part B: Second scan and evaluation of the addition chain

7. *Let* $\mathcal{S} = (m_{\lambda-1}, \ldots, m_{\lambda-s_1})$ *with* $s_1 = \max\{s' \colon \exists d \in \mathcal{D}$ *with* $(d)_2 = (m_\lambda, \ldots, m_{\lambda-s'+1})\}$. *Set* $a_0 = d$ *with* $d \in \mathcal{D}$ *and* $(d)_2 = \mathcal{S}$. *Set* $i = \lambda - s_1$ *and* $j = 1$.
8. *While* $i \geq 0$ *do*
   9. *If* $m_i = 0$ *then evaluate* $a_j = a_{j-1} + a_{j-1}$. *Set* $j = j + 1$ *and* $i = i - 1$.

10. *else the next sequence $\mathcal{S}$ beginning with '1' has been detected: Let this sequence be $\mathcal{S} = (m_i, \ldots, m_{i-s+1})$ of length $s = \max\{s'$: $\exists d \in \mathcal{D}$ with $(d)_2 = (m_i, \ldots, m_{i-s'+1})\}$. Do Steps 5–6.*
    11. *Compute $a_{j+k} = a_{j+k-1} + a_{j+k-1}$ for all $k = 0, \ldots, s-1$. Set $j = j+s$ and $i = i-s+1$.*
    12. *[Comment: Add a element to the addition chain.] Let $d \in \mathcal{D}$ with $(d)_2 = S$. Compute $a_j = a_{j-1} + d$ and set $j = j+1$ and $i = i-1$.*
13. *Return the addition chain built by concatenating $a_0, \ldots, a_{j-1}$ and $\mathcal{D}$.*

An example. The algorithm works as follows: in Part A only bitstrings are scanned to detect a new sequence. A sequence $\mathcal{S}_n$ is a bitstring that has already been stored in Σ followed by 0^z with $z \geq 0$. The algorithm then concatenates $\mathcal{S}_n$ and the previous sequence $\mathcal{S}_l$ and adds it to Σ. It also adds the whole bitstring $(\mathcal{S}_a, \mathcal{S}_n)$ to Σ. Σ' stores only those bitstrings that are used twice or more. Only those bitstrings are worth while calculating. At the end of Part A all bitsequences that have been found twice or more are evaluated (Step 6). This is the proper set $\mathcal{D}$ used in Part B. We illustrate Part A of the algorithm for $m = 5541$ in Figure 3.5.

Part B is just Algorithm 3.39. We therefore illustrate Part B of Algorithm `lookahead` in Figure 3.6 in the same way as Algorithm 3.39.

$\mathcal{D}$ can then be viewed as a binary tree again according to Remark 3.33.

Correctness.

LEMMA 3.45. *Algorithm* `lookahead` *computes an addition chain for $m \in \mathbb{N}$ correctly.*

PROOF. To show partial correctness we first have to show that $\mathcal{D} \neq \emptyset$, $1 \in \mathcal{D}$ and the elements of $\mathcal{D}$ form an addition chain.

We have $1 \in \mathcal{D}$ according to Step 6 and hence $\mathcal{D} \neq \emptyset$. The fact that the elements of $\mathcal{D}$ form an addition chain can be shown by induction over $\lambda = \lfloor \log_2 d \rfloor + 1$, $d \in \mathcal{D}$. We prove by induction that for any $d \in \mathcal{D}$ with $(d)_2 = (d_{\lambda-1}, \ldots, d_1, d_0)$ we can form an addition chain using only elements of $\mathcal{D}$ of bitlength at most $\lambda - 1$ and $(d')_2 = (d_{\lambda-1}, \ldots, d_1, 0), (d'')_2 = (d_{\lambda-1}, \ldots, d_1, 1)$ with $d', d'' \in \mathcal{D}$.
$\lambda = 1$: We have $1 \in \mathcal{D}$ and $0 \notin \mathcal{D}$ according to Step 6. But 1 is an addition chain for 1.
$\lambda \to \lambda + 1$: Let $d \in \mathcal{D}$ with $(d)_2 = (d_{(\lambda+1)-1}, \ldots, d_1, d_0)$ with $d_0 = \{0, 1\}$.

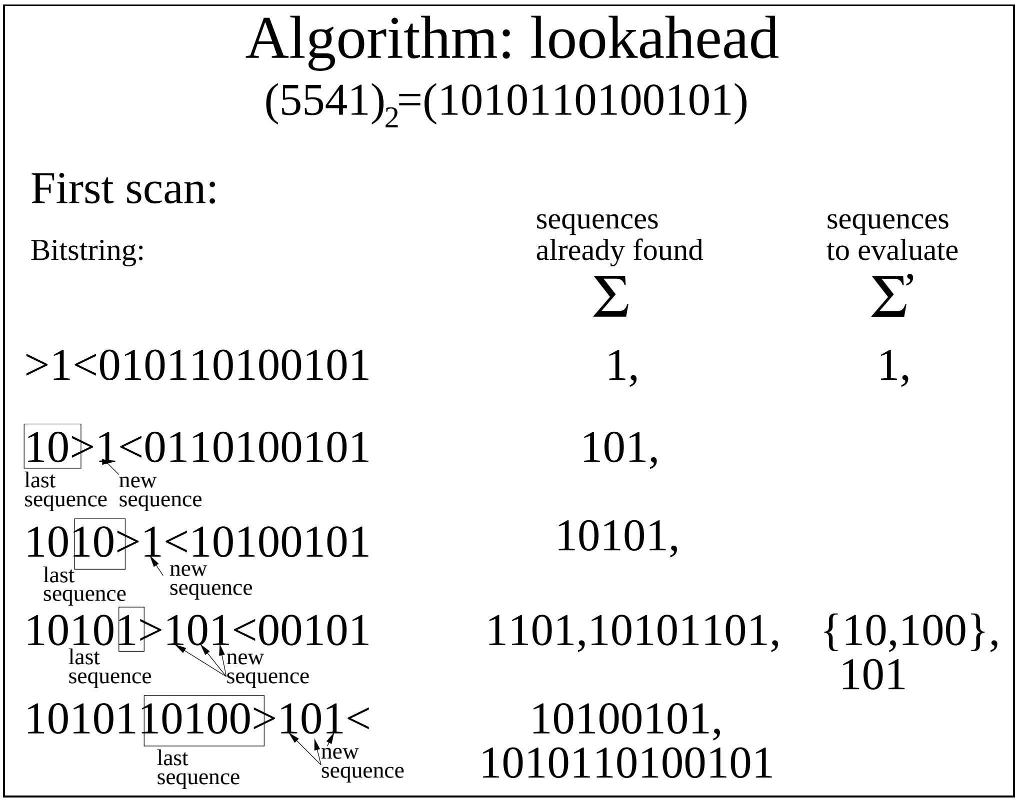

Figure 3.5: A schematic illustration of Part A of Algorithm `lookahead` on input $m = 5541$: all found bitstrings are stored in Σ, bitstrings that are found twice are stored in Σ'.

Then for $(d')_2 = (d_\lambda, \ldots, d_1)$ we have $d' \in \mathcal{D}$ according to the definition of $\mathcal{D}$ in Step 6. But $\lfloor \log_2 d' \rfloor + 1 = \lambda$. By induction hypothesis we can form an addition chain for d' using the elements of $\mathcal{D}$ with bitlength at most $\lambda - 1$ and $(d_\lambda, \ldots, d_2, 0), (d_\lambda, \ldots, d_2, 1)$. But then the following holds: If $d_0 = 0$ then $d' + d' = d$ and we get an addition chain for d. If $d_0 = 1$ then $a = d' + d'$ and $a + 1 = d$ and we get also an addition chain for d. Because $(a)_2 = (d_\lambda, \ldots, d_1, 0)$ we have $a \in \mathcal{D}$ according to Step 6. So in both cases we get an addition chain satisfying the requirements of the induction hypothesis.

Because $\#\mathcal{D} < \infty$ the elements of $\mathcal{D}$ form an addition chain.

Part B is just Algorithm 3.39 for which correctness has already been shown. Because the concatenation of two addition chains form also an addition chain partial correctness has been shown.

Therefore only termination of Part A is left to show: If $(m)_2 = (1, 0^z)$,

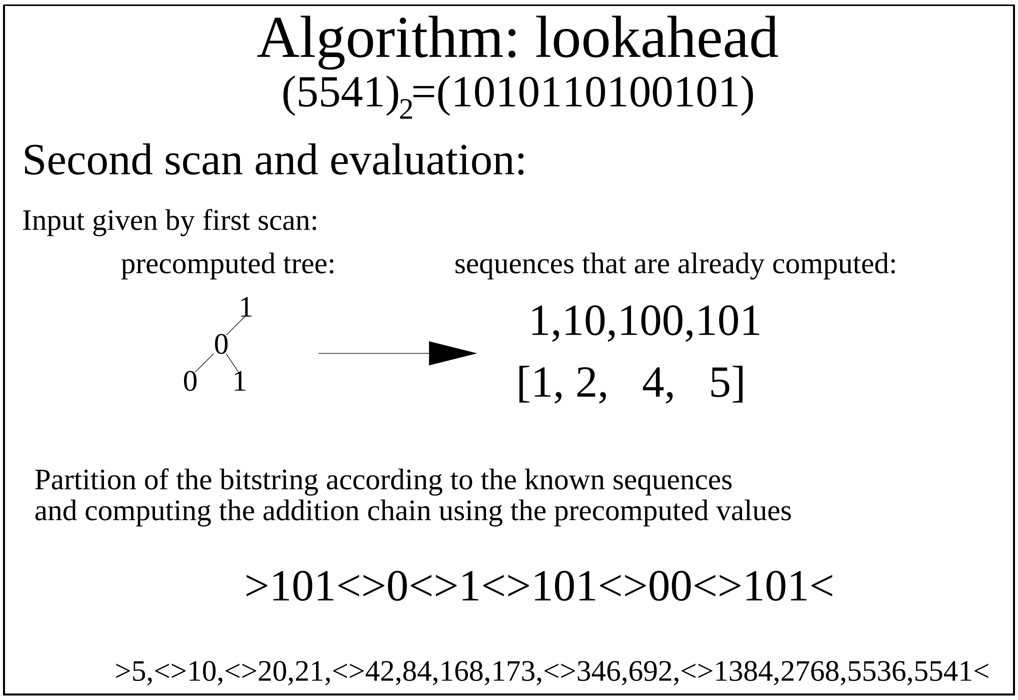

Figure 3.6: A schematikc illustration of Part B of Algorithm `lookahead` on input $m = 5541$ and $\mathcal{D}$ according to Σ'. The evaluation of Σ' is given in Figure 3.5.

termination is clear. In the other case all sequences $\mathcal{S}_n$ start with first bit '1' and $(1)_2 \in \Sigma$. Therefore any sequence contains at least one bit. But there are only λ bits and thus termination follows. $\square$

Number of steps. We denote the number of sequences by S and the length of the first sequence found in the second scan by s_1. If we have already computed all $d \in \mathcal{D}$ in Part A (Step 6) we have the following number of steps in Part B:

We have to scan through $(m)_2$ (Step 9+11) starting at position $\lambda - 1 - s_1$ (Step 7). Therefore $D_1 = \lambda - s_1$. Any sequence except for the first one means one star step (Step 12). Therefore $A_1 = S - 1$.

If we have to do A' star steps and D' doublings in Step 6 to evaluate all $d \in \mathcal{D}$, we can summarize:

Lemma 3.46. *Let $m \in \mathbb{N}$ and $\lambda = \lfloor \log_2 m \rfloor + 1$. Then Algorithm* `lookahead` *produces an addition chain for m with $D = \lambda - 1 - s_1 + D'$ doublings and $A = S - 1 + A'$ star steps where $S \in \mathbb{N}_0$ is the number of sequences in $(m)_2$ found in Part B and s_1 is the length of the first sequence of $(m)_2$ in Part B.*

$A', D' \in \mathbb{N}_0$ denote the number of star steps and doublings that are necessary to evaluate all elements of the proper set $\mathcal{D}$.

Average case. Fix $k \in \mathbb{N}$ and let $\Omega = \{m \in \mathbb{N}: m < 2^k\}$ be a probability space. For an arbitrary chosen $m \in \Omega$ the probability of a bit in $(m)_2$ to be '1' is $\frac{1}{2}$. Let $\mathcal{D}, m, S, s_1, A', D'$ as above.

We assume that the tree built in Step 6 is balanced. The last layer has only leaves that contain $d \in \mathcal{D}$ with $d \equiv 1 \bmod 2$. Let h denote the depth of the tree and k the number of nodes not counting the root node. Then we have

$$k = 2^h - 2 + \frac{2^h}{2} = 3 \cdot 2^{h-1} - 2. \tag{3.5}$$

Assume that any node of the tree represents one sequence according to the rules of Part A. Then we have 2^{i-1} nodes on depth i representing sequences of length $i+1$. As in the proof to Lemma 3.34 we assume a further zero after every sequence on the average. Because the tree only includes the sequences found twice it represents only one half of $(m)_2$. We get:

$$\begin{aligned} \frac{\lambda}{2} &= \sum_{i=1}^{h}(i+2)\cdot 2^{i-1} = \sum_{i=0}^{h-1}(i+2+1)\cdot 2^i \\ &= \sum_{i=0}^{h-1}(i+2)\cdot 2^i + \sum_{i=0}^{h-1} 2^i = h\cdot 2^h - 2 + 2^h - 1 \\ &= (h+1)\cdot 2^h - 3 > h \cdot 2^h - 3. \end{aligned}$$

We then have (cf. proof of Lemma 3.34)

$$2^h < \frac{\frac{\lambda}{2}}{\log_2 \lambda}(1+o(1)) = \frac{1}{2}\frac{\lambda}{\log_2 \lambda}(1+o(1)).$$

Because the tree is balanced we have $A' = D'$: every node has two sons or only the son that contains $d \in \mathcal{D}$ with $d \equiv 1 \bmod 2$. If a node has both sons we can evaluate both by using one doubling and one star step. In the other case we also have to compute one doubling and one star step.

But A' is exactly the number of nodes containing $d \in \mathcal{D}$ with $d \equiv 1 \bmod 2$. We have $\frac{2^h-2}{2} + \frac{2^h}{2} = 2^{h-1} - 1 + 2^{h-1} = 2^h - 1$ such nodes and therefore

$$D' = A' = 2^h - 1 < \frac{1}{2}\frac{\lambda}{\log_2 \lambda}(1+o(1)).$$

We now analyze Part B: Because the tree given in Step 6 is balanced we can assume that all sequences found in Step 10 except for the last one end at the layer h or $h-1$ of the tree. This is true because if we found a sequence $\mathcal{S}_n = (m_i, \ldots, m_{i-s+1})_2$ ending in layer $j < h-1$ and $i-s+1 \neq 0$ then we have $(\mathcal{S}_n, 0)_2$ and $(\mathcal{S}_n, 1)_2$ ending at layer $j+1$ because the tree is balanced. Therefore we have 2^{h-1} possible sequences of length h at layer $h-1$ and $\frac{2^h}{2}$ possible sequences of length $h+1$ at layer h. We expect the average length of a sequence to be $\frac{2^{h-1}h+2^{h-1}(h+1)}{2\cdot 2^{h-1}} = \frac{2^{h-1}}{2^h}(h+h+1) = \frac{2h+1}{2} = h+\frac{1}{2}$. Therefore we have

$$s_1 = h + \frac{1}{2} < \log_2 \frac{1}{2}\frac{\lambda}{\log_2 \lambda}(1+o(1)) < \log_2 \frac{\lambda}{\log_2 \lambda}$$

and $\lambda = S(h+\frac{1}{2})$ which means

$$S = \frac{\lambda}{h+\frac{1}{2}} < \frac{\lambda}{\log_2 \lambda - \log_2 \log_2 \lambda}.$$

We can summarize:

Lemma 3.47. *Let $m \in \mathbb{N}$ and $\mu = \log_2 m$. On the average Algorithm* `look-ahead` *computes an addition chain for m with*

$$\begin{aligned} D_{ave} &< \lfloor \mu \rfloor + \frac{\mu}{2\log_2 \mu}(1+o(1)) \text{ doubling steps and} \\ A_{ave} &< \frac{3}{2}\frac{\mu}{\log_2 \mu}(1+o(1)) \text{ star steps.} \end{aligned}$$

Proof. Using the results above we get:

$$\begin{aligned} D_{ave} &= \lambda - 1 - s_1 + D' < \lambda + D' \\ &< \lambda + \frac{1}{2}\frac{\lambda}{\log_2 \lambda}(1+o(1)) \\ A_{ave} &= S - 1 + A' \\ &< \frac{\lambda}{\log_2 \lambda - \log_2 \log_2 \lambda} + \frac{1}{2}\frac{\lambda}{\log_2 \lambda}(1+o(1)) \\ &= \frac{3}{2}\frac{\lambda}{\log_2 \lambda}(1+o(1)). \end{aligned}$$

Using $\lambda = \lfloor \mu \rfloor$ we have:

$$\begin{aligned} D_{ave} &< \mu + \frac{\mu}{2\log_2(\mu-1)}(1+o(1)) \\ A_{ave} &< \frac{3}{2}\frac{\mu}{\log_2(\mu-1)}(1+o(1)). \ \square \end{aligned}$$

Worst case. The problem to fix the worst case for Algorithm `lookahead` is that we haven't found a relation between the number of sequences S and the number of star steps A' in the worst case. In the worst case the sequences are as short as possible to get a big S. Simular to the arguments given for the worst case of `bocharova` we have to assume a complete binary tree. But then $S < \frac{\lambda}{\log_2 \lambda - \log_2 \log_2 \lambda}$ as in the average case. If we only have a look at A', the worst case can be found easily: For $m = 2^{\lambda+1} - 1$ and $\lambda = 2^k, k \in \mathbb{N}$ we have to do $\frac{1}{2}\lambda$ star steps computing $\mathcal{D}$ in Step 6 of the algorithm. Because we assume two different situations to find an upper bound for S and A' respectively, we only can give an unsharp upper bound.

COROLLARY 3.48. *Let $m \in \mathbb{N}$ and $\mu = \log_2 m$. An upper bound on the length of the addition chain for m computed by Algorithm* `lookahead` *is given by*

$$2\lambda + \frac{\lambda}{\log_2 \lambda - \log_2 \log_2 \lambda}.$$

PROOF. We have $S < \frac{\lambda}{\log_2 \lambda - \log_2 \log_2 \lambda}$ and $A' \le \frac{\lambda}{2}$. But then $D' \le \frac{\lambda}{2}$ according to Step 6 of the algorithm. We therefore get

$$\begin{aligned} A &= S - 1 + A' < \frac{\lambda}{\log_2 \lambda - \log_2 \log_2 \lambda} + \frac{\lambda}{2}, \\ D &= \lambda - 1 - s_1 + D' < \lambda + D' \le \frac{3}{2}\lambda. \end{aligned}$$

Because of $L = A + D$ we get the upper bound. □

REMARK 3.49. *Algorithm* `lookahead` *can probably be improved by defining $\mathcal{D} = \{d \in \mathbb{N}\colon (d)_2 \in \Sigma'\}$ in Step 6. But this has not been analyzed yet.*

3.10. Summarizing survey. The following tables show the results of this section. $m \in \mathbb{N}$ is the integer for which an addition chain has to be computed. We only consider the case $q = 2$ to facilitate comparison of all algorithms.

For the algorithms based on the idea of using the q^r-ary representation of m we give the worst case as an upper bound (Table 1). We use the same notations as above with input m and $\mu = \log_2 m$.

For the algorithms based on data compression we concentrate on the average case (Table 2).

Algorithm (reference)	`binary` (Alg. 3.13)	`brauer` (Alg. 3.17)	`bgmw` (Alg. 3.25)
#steps L	$\lfloor\mu\rfloor + \nu_2(m) - 1$	$\nu_{2^r}(m) + 2^r - 3$ $+r\lfloor\frac{\mu}{r}\rfloor - (r - \lfloor\log_2\lfloor\frac{m}{2^{r\lfloor\frac{\mu}{r}\rfloor}}\rfloor\rfloor)$	$(r+1)\lfloor\frac{\mu}{r}\rfloor + 2^r - 2$
#doublings D	$\lfloor\mu\rfloor$	$r\lfloor\frac{\mu}{r}\rfloor - (r - \lfloor\log_2\lfloor\frac{m}{2^{r\lfloor\frac{\mu}{r}\rfloor}}\rfloor\rfloor)$	$r\lfloor\frac{\mu}{r}\rfloor$
#further steps A	$\nu_2(m) - 1$	$\nu_{2^r}(m) + 2^r - 3$	$\lfloor\frac{\mu}{r}\rfloor + 2^r - 2$
Upper bounds			
Parameter r		$\lfloor\frac{1}{2}\log_2\mu\rfloor + 1$	$\lfloor\log_2\mu - 2\log_2\log_2\mu\rfloor + 1$
L_{worst}	$\leq 2\mu$	$\leq \mu + 2\frac{\mu}{\log_2\mu}(1+o(1))$	$\leq \mu + \frac{\mu}{\log_2\mu}(1+o(1))$
D_{worst}	$= \lfloor\mu\rfloor$	$\leq \mu$	$\leq \mu$
A_{worst}	$= \lceil\mu\rceil - 1$	$\leq 2\frac{\mu}{\log_2\mu}(1 + \frac{\log_2\mu}{\sqrt{\mu}})$	$< \frac{\mu}{\log_2\mu}(1$ $+2\frac{\log_2\log_2\mu}{\log_2\mu - 2\log_2\log_2\mu} + \frac{2}{\log_2\mu})$

Description: $\mu = \log_2 m$

Table 1: Theoretical comparison between the classical addition chain algorithms.

Algorithm (reference)	`yacobi` (Alg. 3.31)	`bocharova` (Alg. 3.39)	`lookahead` (Alg. 3.44)
#steps L	$\lfloor\mu\rfloor + 2S + R$	$\lfloor\mu\rfloor + 2r - S - s_1 - 2$	$\lfloor\mu\rfloor - s_1 + D'$ $+S - 1 + A'$
#doublings D	$\lfloor\mu\rfloor + S$	$r + \lfloor\mu\rfloor - s_1$	$\lfloor\mu\rfloor - s_1 + D'$
#further steps A	$R + S$	$r + S - 2$	$S - 1 + A'$
Average case			
Parameter r		$\lfloor\frac{\mu}{(\log_2\mu)^2}\rfloor$	
$L_{ave} <$	$\lfloor\mu\rfloor + \frac{5}{2}\frac{\mu}{\log_2\mu}(1+o(1))$	$\lfloor\mu\rfloor + \frac{\mu}{(\log_2\mu)}(1+o(1))$	$\lfloor\mu\rfloor + 2\frac{\mu}{\log_2\mu}(1+o(1))$
$D_{ave} <$	$\lfloor\mu\rfloor + \frac{\mu}{\log_2\mu}(1+o(1))$	$\lfloor\mu\rfloor + \frac{\mu}{(\log_2\mu)^2}$	$\lfloor\mu\rfloor + \frac{1}{2}\frac{\mu}{\log_2\mu}(1+o(1))$
$A_{ave} <$	$\frac{3}{2}\frac{\mu}{\log_2\mu}(1+o(1))$	$\frac{\mu}{\log_2\mu}(1+o(1))$	$\frac{3}{2}\frac{\mu}{\log_2\mu}(1+o(1))$
Upper bounds			
Parameter r		$\lfloor\frac{\mu}{(\log_2\mu)^2}\rfloor$	
$L_{worst} <$	$\mu + \frac{5}{2}\frac{\mu\log_2\log_2\mu}{\log_2\mu - \log_2\log_2\mu}(1+o(1))$	$\mu + 2\frac{\mu\log_2\log_2\mu}{\log_2\mu}(1+o(1))$	$2\mu + \frac{\mu}{\log_2\mu}(1+o(1))$
$D_{worst} \leq$	$\mu + \frac{\mu\log_2\log_2\mu}{\log_2\mu - \log_2\log_2\mu}(1+o(1))$	$\mu + \frac{\mu}{(\log_2\mu)^2}$	$\frac{3}{2}(\mu+1)$
$A_{worst} <$	$\frac{3}{2}\frac{\mu\log_2\log_2\mu}{\log_2\mu - \log_2\log_2\mu}(1+o(1))$	$2\frac{\mu\log_2\log_2\mu}{\log_2\mu}(1+o(1))$	$\frac{1}{2}\mu + \frac{\mu}{\log_2\mu}(1+o(1))$

Description: S: #sequences, R: #sequences with last bit '1', s_1: length of first sequence, $\mu = \log_2 m$, A', D': #star steps/doublings in Part A

Table 2: Theoretical comparision between addition chain algorithms based on data compression.

4. Fast exponentiation

4.1. The relation between addition chains and exponentiation.

A homomorphism. We briefly mentioned the relation between addition chains and exponentiation already: because exponents are additive we concentrated on addition chains so far. We will now transfer the results of the previous section to exponentiation.

DEFINITION 4.1. *Let G be any multiplicative group and $b \in G$. We define a map*

$$\begin{array}{rccc} \varphi\colon & \mathbb{N}_0 & \to & G \\ & i & \mapsto & b^i \end{array}$$

Then φ defines a homomorphism between commutative monoids since $\varphi(0) = b^0 = 1$ and $\varphi(i+j) = b^{i+j} = b^i \cdot b^j = \varphi(i) \cdot \varphi(j)$ for all $i, j \in \mathbb{N}_0$.

REMARK 4.2. 1. *Let $1 = a_0, \ldots, a_L = e$ be an addition chain of length L for $e \in \mathbb{N}$. Then $b = \varphi(a_0), \ldots, \varphi(a_L) = b^e$ is a prescription how to compute b^e with L multiplications and squarings, respectively.*

2. *A doubling step i becomes a squaring under φ because $b^{a_i} = \varphi(a_i) = \varphi(a_j + a_j) = \varphi(a_j)\varphi(a_j) = b^{a_j} \cdot b^{a_j} = (b^{a_j})^2$ with $0 \leq j < i$.*

3. *If we have a q-generalized addition chain then a q-step i becomes $b^{a_i} = \varphi(a_i) = \varphi(q \cdot a_j) = b^{q \cdot a_j} = (b^{a_j})^q$ which means that raising to the qth power is only one operation when computing b^e.*

Because of the definition of φ we can easily transform the algorithms for addition chains by computing $\varphi(a_i)$ instead of a_i for any element a_i and $0 \leq i \leq L$ of the addition chain.

An algorithm. These ideas can also be expressed in an algorithmic way:

ALGORITHM 4.3. `addition chain to exponentiation`
Input: $1 = a_0, \ldots, a_L = e$ *an addition chain for $e \in \mathbb{N}$ and an element $b \in G$ where G is a multiplicative group.*
Output: $b^e \in G$.

1. *Set $b_0 = b^{a_0} = b$.*

2. *For $i = 1$ to L do*

3. *Let $0 \leq j, k < i$ be indices with $a_i = a_j + a_k$ according to the given addition chain for e. Compute $b_i = b^{a_i} = b^{a_j + a_k} = b^{a_j} \cdot b^{a_k} = b_j \cdot b_k$.*

4. *Return b_L.*

In fact, this is actually a 'compiler' that transforms addition chains for e into algorithms for computing b^e from b, for any b in any group G.

Because we have an explicit function φ it is clear how to get an exponentiation algorithm if an addition chain is given. But what about the other way round? Downey *et al.* (1981) write that it is an "observation that computations involving multiplication and a single variable x are *isomorphic* to computations involving addition and the integer 1."

But this "isomorphism" does not have an inverse in general way. If G is finite, we can compute $b^{\#G} = 1$ due to Lagrange's Theorem without any multiplication nor squaring. But an addition chain for $\#G$ has $\Omega(\log(\#G))$ elements.

Memory requirements. Another question has not been answered yet: how many elements of the addition chain are used to generate not only the immediately following but also further elements of the addition chain? This can be reformulated for exponentiation algorithms: How many powers of b evaluated during the computation have to be stored if we are only interested in evaluating $b^e \in G$? We will pay attention to the demand of storage when transferring the results of the addition chain heuristics to exponentiation.

4.2. Results transferred from addition chains. According to the practical part of this Diplomarbeit we concentrate on $G = \mathbb{F}_{q^n}^\times$ with $n \in \mathbb{N}$. We can assume without loss of generality that $0 < e < \#G = \#\mathbb{F}_{q^n}^\times = q^n - 1$ for a given exponent $e \in \mathbb{N}_0$.

Binary method.

COROLLARY 4.4. *For given $e \in \mathbb{N}$ and $b \in \mathbb{F}_{q^n}^\times$ we can compute $b^e \in \mathbb{F}_{q^n}$ with $D = \lfloor \log_2 e \rfloor$ squarings and $A = \nu_2(e) - 1$ further multiplications according to Algorithm* `binary`.

We only have to store the input b and one intermediate result.

PROOF. The number of multiplications and squarings follows directly from Lemma 3.14. The demand on storage is clear because in Algorithm 3.13 we only have to do doublings — which means that we have to square the intermediate result — and star steps with d_0 as second summand — which means that we have to multiply the intermediate result with b. □

COROLLARY 4.5. *For any* $e \in \mathbb{N}, b \in \mathbb{F}_{q^n}$ *we can compute* $b^e \in \mathbb{F}_{q^n}$ *with* $A < n \log_2 q$ *multiplications and* $D < n \log_2 q$ *squarings.*

PROOF. Use Corollary 3.15 and $e < q^n$. □

q^r-ary method.

COROLLARY 4.6. *For given* $e, r \in \mathbb{N}$ *and* $b \in \mathbb{F}_{q^n}^\times$ *we can compute* $b^e \in \mathbb{F}_{q^n}^\times$ *with at most* $Q = r - 1 + r\lfloor \log_{q^r} e \rfloor$ *qth powers and* $A = q^r - q^{r-1} - 1 + \lfloor \log_{q^r} e \rfloor$ *further multiplications according to the* q^r*-ary method.*

We have to store $q^r - 1$ *elements of* $\mathbb{F}_{q^n}^\times$ *and one intermediate result.*

PROOF. A and Q can be found in Lemma 3.22. The q^r-ary method uses only q-steps and star steps with second summand $d_i \in \{1, \ldots, q^r - 1\}$. This can be easily derived from Algorithm 3.17 generalized for $q \in \mathbb{N}$. □

COROLLARY 4.7. *For any* $e \in \mathbb{N}$ *and* $b \in \mathbb{F}_{q^n}^\times$ *we can compute* $b^e \in \mathbb{F}_{q^n}^\times$ *with* $A \leq q\frac{n}{\log_q n}(1 + o(1))$ *multiplications and* $Q \leq n + \frac{1}{q}\log_q n$ *qth powers. This can be done storing* $q\sqrt[q]{n}$ *elements of* $\mathbb{F}_{q^n}^\times$ *and one intermediate result.*

PROOF. Choose $r = \lfloor \frac{1}{q}\log_q n \rfloor + 1$. Then we can estimate A and Q as in Corollary 3.23.

And finally we have to store $q^r - 1 = q^{\lfloor \frac{1}{q}\log_q n \rfloor + 1} - 1 < q\sqrt[q]{n}$ elements of $\mathbb{F}_{q^n}^\times$. □

The algorithm of Brickell *et al.*

COROLLARY 4.8. *For given* $e, r \in \mathbb{N}$ *and* $b \in \mathbb{F}_{q^n}^\times$ *we can compute* $b^e \in \mathbb{F}_{q^n}^\times$ *with at most* $Q = r\lfloor \log_{q^r} e \rfloor$ *qth powers and* $A = q^r + \lfloor \log_{q^r} e \rfloor - 2$ *further multiplications according to Algorithm* `bgmw`.

We have to store $\lfloor \log_{q^r} e \rfloor + 1$ *elements of* $\mathbb{F}_{q^n}^\times$ *and two intermediate results.*

PROOF. Lemma 3.27 gives A and Q. The demand of storage can be easily seen from Algorithm 3.25. □

COROLLARY 4.9. *For any* $e \in \mathbb{N}, b \in \mathbb{F}_{q^n}^\times$ *we can compute* $b^e \in \mathbb{F}_{q^n}^\times$ *with* $A = \frac{n}{\log_q n}(1 + o(1))$ *multiplications and* $Q < n$ *qth powers. This can be done storing at most* $\frac{n}{\log_q n}(1 + o(1))$ *elements of* $\mathbb{F}_{q^n}^\times$.

PROOF. With $r = \lfloor \log_q n - 2\log_q \log_q n \rfloor + 1$ we can estimate A as in the proof of Corollary 3.28. With $e < q^n$ we have $Q < \frac{r}{r} \log_q q^n = n$. Finally we have to store $\log_{q^r} e$ elements of $\mathbb{F}_{q^r}$. With r as above we get the upper bound of $\frac{n}{\log_q - 2\log_q \log_q n} = \frac{n}{\log_q n}(1 + o(1))$. □

This corollary was first proven for $q = 2$ by Stinson (1990) and Agnew *et al.* (1988); von zur Gathen (1991) has shown it for all finite fields.

Yacobi's algorithm.

COROLLARY 4.10. *For given $e \in \mathbb{N}$ and $b \in \mathbb{F}_{q^n}^\times$ we can compute $b^e \in \mathbb{F}_{q^n}^\times$ with $D = \lfloor \log_2 e \rfloor + S$ squarings and $A = R + S$ further multiplications according to Algorithm* `yacobi`. *S denotes the number of sequences generated in the algorithm and R is the number of different sequences with last bit 1.*

We have to store at most S elements of $\mathbb{F}_{q^n}^\times$ and the intermediate result.

PROOF. A and D are given by Lemma 3.34. Algorithm 3.31 shows that only $d_k, k \in \mathcal{D}$ have to be stored. But $\#\mathcal{D} = S$. □

COROLLARY 4.11. *The expected number of elements of $\mathbb{F}_{q^n}^\times$ to store is $\frac{\log_2 e}{\log_2 \log_2 e}$ $(1 + o(1))$.*

PROOF. We have to store S elements of $\mathbb{F}_{q^N}^\times$. According to Lemma 3.35 we expect $S = \frac{\log_2 e}{\log_2 \log_2 e}(1 + o(1))$. □

COROLLARY 4.12. *Let $e \in \mathbb{N}$ and $b \in \mathbb{F}_{q^n}^\times$. Let $\lambda = \lfloor \log_2 e \rfloor + 1$. Then $b^e \in \mathbb{F}_{q^n}^\times$ can be computed in at most $A \leq \frac{3}{2} \frac{\lambda \log_2 \log_2 \lambda}{\log_2 \lambda - \log_2 \log_2 \lambda}(1 + o(1))$ multiplications and $Q \leq \lambda + \frac{\lambda \log_2 \log_2 \lambda}{\log_2 \lambda - \log_2 \log_2 \lambda}(1 + o(1))$ qth powers. This can be done storing at most $\frac{\lambda \log_2 \log_2 \lambda}{\log_2 \lambda - \log_2 \log_2 \lambda}(1 + o(1))$ elements of $\mathbb{F}_{q^n}^\times$.*

PROOF. This follows directly from Corollary 3.37 and the fact that we have to store at most S elements of $\mathbb{F}_{q^n}^\times$. □

Bocharova's idea.

COROLLARY 4.13. *For given $e, r \in \mathbb{N}$ and $b \in \mathbb{F}_{q^n}^\times$ we can compute $b^e \in \mathbb{F}_{q^n}^\times$ with $D = \lfloor \log_2 e \rfloor + r - s_1$ squarings and $A = r + S - 2$ further multiplications according to Algorithm* `bocharova`. *S is the number of sequences generated in the algorithm and s_1 is the length of the first detected sequence of $(e)_2$.*

We have to store $2r - 1$ elements of $\mathbb{F}_{q^n}^\times$ and the intermediate result.

PROOF. Lemma 3.41 shows the correctness of A and D. The number of elements that have to be stored are given by the output of Algorithm 3.38. □

COROLLARY 4.14. *Let $\eta = n \log_2 q$. For any $e \in \mathbb{N}$, $b \in \mathbb{F}_{q^n}^{\times}$ we can compute $b^e \in \mathbb{F}_{q^n}^{\times}$ with $A < 2\frac{\eta \log_2 \log_2 \eta}{\log_2 \eta}(1 + o(1))$ and $D \leq \eta(1 + \frac{1}{(\log_2 \eta)^2})$. This can be done storing at most $2\frac{\eta}{(\log_2 \eta)^2} - 1$ elements.*

PROOF. With $r = \lfloor \frac{n \log_2 q}{(\log_2(n \log_2 q))^2} \rfloor = \lfloor \frac{\eta}{(\log_2 \eta)^2} \rfloor$ we can estimate A and Q as in the proof of Corollary 3.43. We have to store $2r - 1 = 2\lfloor \frac{n \log_2 q}{(\log_2(n \log_2 q))^2} \rfloor - 1$ elements. □

The new algorithm.

COROLLARY 4.15. *For given $e \in \mathbb{N}$ and $b \in \mathbb{F}_{q^n}^{\times}$ we can compute $b^e \in \mathbb{F}_{q^n}^{\times}$ with $D = \lfloor \log_2 e \rfloor - s_1 + D'$ squarings and $A = S - 1 + A'$ further multiplications according to Algorithm* `lookahead`. *$S \in \mathbb{N}_0$ is the number of sequences in $(e)_2$ found in Part B and s_1 is the length of the first sequence of $(e)_2$ in Part B. $A', D' \in \mathbb{N}_0$ denote the number of star steps and doublings that are necessary to evaluate all elements of the proper set $\mathcal{D}$.*

We have to store $\#\mathcal{D}$ elements of $\mathbb{F}_{q^n}^{\times}$ and the intermediate result.

PROOF. Lemma 3.46 proves the correctness of A and D. The elements to store can be seen in Algorithm 3.44: We have to store one value d_k for any $k \in \mathcal{D}$. □

COROLLARY 4.16. *The expected number of elements of $\mathbb{F}_{q^n}^{\times}$ to store is $< \frac{3}{4}\frac{\log_2 e}{\log_2 \log_2 e}(1 + o(1))$.*

PROOF. Remember the tree we used to analyze the average case of Algorithm 3.44. Then $\#\mathcal{D}$ is just the number of nodes in this tree. But in Equation (3.5) we have shown that $k = 3 \cdot 2^{h-1} - 2$ where h is the depth of this tree and $2^h < \frac{\lambda}{2 \log_2 \lambda}(1+o(1))$ with $\lambda = \lfloor \log_2 e \rfloor + 1$. Then we get $k < \frac{3}{2}\frac{\lambda}{2 \log_2 \lambda}(1+o(1)) = \frac{3}{4}\frac{\log_2}{\log_2 \log_2 e}(1 + o(1))$. □

COROLLARY 4.17. *Let $e \in \mathbb{N}$ and $b \in \mathbb{F}_{q^n}^{\times}$. Let $\lambda = \lfloor \log_2 e \rfloor + 1$. We can compute $b^e \in \mathbb{F}_{q^n}^{\times}$ with $A \leq \frac{\lambda}{\log_2 \lambda - \log_2 \log_2 \lambda} + \frac{\lambda}{2}$ multiplications and $Q \leq \frac{3}{2}\lambda$ qth powers.*

PROOF. Can be seen directly from Corollary 3.48. □

Algorithm	#elements to store	expected demand	worst case	parameter r/ remarks
`binary`	1	1	1	store b
q-`ary`	$q^r - 1$		$q\sqrt[q]{n}$	$\lfloor \frac{1}{q} \log_q n \rfloor + 1$
`bgmw`	$\lfloor \log_{q^r} e \rfloor + 1$		$\frac{n}{\log_q n}(1 + o(1))$	$\lfloor \log_q n - 2 \log_q \log_q n \rfloor + 1$
`yacobi`	S	$\frac{\log_2 e}{\log_2 \log_2 e}(1 + o(1))$		
`bocharova`	$2r - 1$		$2\lfloor \frac{n \log_2 q}{(\log_2 n \log_2 q)^2} \rfloor - 1$	$\lfloor \frac{n \log_2 q}{(\log_2 n \log_2 q)^2} \rfloor$
`lookahead`	$\#\mathcal{D}$	$< \frac{3}{4} \frac{\log_2 e}{\log_2 \log_2 e}(1 + o(1))$		

Description: S: #sequences, $\mathcal{D}$: precomputed set

Table 3: Memory requirements for exponentiation algorithms.

A summarizing table. In Table 3 we finally summarize the demand on storage for $\mathbb{F}_{q^n}^{\times}$.

5. Inversion in $\mathbb{F}_{q^n}$

We now apply the results about addition chains to an interesting problem over finite fields: inversion. In the literature "two different methods for finding the inverse algorithmically are well–known"(Brunner *et al.* 1993, p. 1010): the first is based on Fermat's Little Theorem and uses exponentiation with a very special exponent. The second one is based on the Extended Euclidean Algorithm. In the following the method using exponentiation is shown in detail. Then it is compared to the Euclidean method.

5.1. Inversion based on Fermat's Little Theorem.

The main idea. From Fermat's Little Theorem we have $\alpha^{q^n-1} \equiv 1 \bmod q^n$ for $q \in \mathbb{N}$ prime, $\alpha \in \mathbb{F}_{q^n} \setminus \{0\} = \mathbb{F}_{q^n}^\times$ and $n \in \mathbb{N}$. We therefore can calculate the inverse of $\alpha \in \mathbb{F}_{q^n}^\times$ as $\alpha^{-1} = 1 \cdot \alpha^{-1} = \alpha^{q^n-1} \cdot \alpha^{-1} = \alpha^{q^n-2}$. But we have

$$q^n - 2 = q^n - q + q - 2 = (q^{n-1} - 1)q + (q - 2) \tag{5.1}$$

and $(q^{n-1} - 1)_q = (q - 1, \ldots, q - 1)$ is a very special exponent.

REMARK 5.1. *The fact that $b^{\#G} = 1$ for any $b \in G, b \neq 0$ and G any finite group is known as Lagrange's theorem. But in the special case given above we have a consequence of Fermat's Little Theorem. Because Fermat found his theorem first, we denote this method to invert as* Fermat's method, *which is common in the literature (cf. e.g. Brunner* et al. *1993).*

The basic algorithm. The following algorithm reduces the problem of calculating the inverse of an element $\alpha \in \mathbb{F}_{q^n}^\times$ to addition chains. It uses the idea of Equation (5.1).

THEOREM 5.2. *Let $\alpha \in \mathbb{F}_{q^n}^\times, q \in \mathbb{N}$ prime, and an addition chain for $n - 1$ of length L_1 and, if $q > 2$ an addition chain for $q - 2$ of length L_2 be given. Then we can evaluate $\alpha^{-1} \in \mathbb{F}_{q^n}$ with*

1. *$L_1 + L_2 + 2$ multiplications in $\mathbb{F}_{q^n}$ if $q > 2$, and*
2. *L_1 multiplications in $\mathbb{F}_{2^n}$ if $q = 2$.*

Let $b_{j_i} + b_{k_i} = b_i$ for $0 \leq j_i \leq k_i < i$ according to the first addition chain. Then we have to compute $1 + \sum_{i=1}^{L_1} b_{j_i}$ qth powers in $\mathbb{F}_{q^n}$.

PROOF. We prove this by giving an algorithm. For reasons of comfort this algorithm is divided into three parts which are analyzed separately.

ALGORITHM 5.3. `inverse`
Input: $\alpha \in \mathbb{F}_{q^n}^{\times}$ *with* $q \in \mathbb{N}$ *prim,* $n \in \mathbb{N}$ *and two addition chains:* $1 = b_0, \ldots, b_{L_1} = n-1$ *for* $n-1$ *of length* L_1 *and* $1 = a_0, \ldots, a_{L_2} = q-2$ *for* $q-2$ *of length* L_2.
Output: $\alpha^{-1} \in \mathbb{F}_{q^n}$.

Part A: Calculating $y = \alpha^{q-2} \in \mathbb{F}_{q^n}$.

1. *Set* $P[a_0] = \alpha$.
2. *For* $1 \leq i \leq L_2$ *compute* $P[a_i] = P[a_j] \cdot P[a_k]$, *where* $j, k \in \mathbb{N}_0, 0 \leq j \leq k < i$ *with* $a_i = a_j + a_k$ *according to the given addition chain for* $q-2$. *[Comment: the following invariant holds:* $P[a_i] = \alpha^{a_i}$.*]*
3. *Set* $y = P[a_{L_2}]$.

Part B: Calculating $x = \alpha^{q^{n-1}-1} \in \mathbb{F}_{q^n}$.

4. *Compute* $N[b_0] = \alpha \cdot y$.
5. *For* $1 \leq i \leq L_1$ *do [Comment: the invariant is given by* $N[b_i] = \alpha^{q^{b_i}-1}$ *for all* $0 \leq i \leq L_1$.*]*
   6. *Let* $j, k \in \mathbb{N}_0, 0 \leq j \leq k < i$ *with* $b_i = b_j + b_k$.
   7. *Compute* $x = N[b_k]^{q^{b_j}}$.
   8. *Compute* $N[b_i] = x \cdot N[b_j]$.
9. *Set* $x = N[b_{L_1}]$.

Part C: Calculating $x^q y = \alpha^{-1} \in \mathbb{F}_{q^n}$.

10. *Return* $x^q \cdot y$.

LEMMA 5.4. *Part A of the algorithm computes* $y = \alpha^{q-2} \in \mathbb{F}_{q^n}$ *and needs* L_2 *multiplications in* $\mathbb{F}_{q^n}$.

PROOF. We assume $P[a_i] = \alpha^{a_i}$ for all $0 \leq i \leq L_2$ as an invariant. In Step 1 the initial step is done by calculating $P[a_0] = P[1] = \alpha^1 = \alpha^{a_0}$, because $a_0 = 1$. Therefore the invariant holds before entering the loop in Step 2. In Step 2 the main work is done: let $P[a_j] = \alpha^{a_j}$ be already calculated for

$j, k < i$. Then $P[a_i] = P[a_j] \cdot P[a_k] = \alpha^{a_j} \cdot \alpha^{a_k} = \alpha^{a_j + a_k} = \alpha^{a_i}$ because of the choice of j, k. After the loop $P[a_{L_2}] = \alpha^{a_{L_2}} = \alpha^{q-2}$ has been evaluated and therefore $y = \alpha^{q-2} \in \mathbb{F}_{q^n}$ in Step 3. Since there is one multiplication for any $i \in \{1, \ldots, L_2\}$ in Step 2, Part A needs L_2 multiplications at all. $\square$

LEMMA 5.5. *Part B of the algorithm computes $x = \alpha^{q^{n-1}-1} \in \mathbb{F}_{q^n}$. It needs $1 + L_1$ multiplications in $\mathbb{F}_{q^n}$.*

PROOF. We first prove the invariant $N[b_i] = \alpha^{q^{b_i}-1}$ for all $0 \le i \le L_1$: again in Step 4 initial work is done by calculating $N[b_0] = N[1] = \alpha \cdot \alpha^{q-2} = \alpha^{q-1} = \alpha^{q^{b_0}-1}$. Therefore the invariant $N[b_i] = \alpha^{q^{b_i}-1}$ holds before entering the loop in Step 5. By induction hypothesis we may assume that $N[b_j] = \alpha^{q^{b_j}-1}$ for all $j, k < i$. Then $x = N[b_j]^{q^{b_k}} = \alpha^{(q^{b_j}-1)q^{b_k}} = \alpha^{q^{b_j+b_k}-q^{b_k}}$ and $N[b_i] = \alpha^{q^{b_j+b_k}-q^{b_k}} \cdot N[b_k] = \alpha^{q^{b_i}-q^{b_k}+q^{b_k}-1} = \alpha^{q^{b_i}-1}$ and the invariant holds after running through the loop for i. Therefore in Step 6 the algorithm returns $N[b_{L_1}] = N[n-1] = \alpha^{q^{n-1}-1}$. The number of multiplications can directly be seen from Step 4 and Step 8: because there are L_1 rounds of Step 5 we get $1 + L_1$ multiplications. $\square$

LEMMA 5.6. *Part C of the algorithm computes $x^q y = \alpha^{-1} \in \mathbb{F}_{q^n}$ and needs one multiplication in $\mathbb{F}_{q^n}$.*

PROOF. Because $x^q = (\alpha^{q^{n-1}-1})^q = \alpha^{q^n - q}$ and $y = \alpha^{q-2}$ the algorithm returns $\alpha^{q^n-q} \cdot \alpha^{q-2} = \alpha^{q^n-2} = \alpha^{-1}$. This last part can be calculated using one further multiplication. $\square$

Concatenating the three parts we have built an algorithm that calculates $\alpha^{-1} \in \mathbb{F}_{q^n}$ for given $\alpha \in \mathbb{F}_{q^n}$ if two addition chains for $n-1$ and $q-2$ (if $q > 2$) have been given. If $q = 2$, then $\alpha^{q-2} = 1$ and therefore we can skip Part A of the algorithm and also the multiplications in Step 4 (Part B) and Step 10 (Part C). Therefore we get the result: if $q = 2$ we need $L_1 + 1$ multiplications in $\mathbb{F}_{2^n}$. If $q > 2$, there are $L_1 + 1 + L_2 + 1$ multiplications in $\mathbb{F}_{q^n}$. The number of qth powers that have to be evaluated can be directly seen from Part B (Step 7) and Part C (Step 10). $\square$

A corollary. We are now ready to formulate the summarizing theorem:

COROLLARY 5.7. *Let $\alpha \in \mathbb{F}_{q^n}^{\times}$, $q \ge 2$ prime. Then the inverse of α in $\mathbb{F}_{q^n}$ can be evaluated using*

1. $\log_2(n-1)(1+\frac{2}{\log_2\log_2(n-1)}+\frac{2}{\sqrt{\log_2(n-1)}}) = \log_2(n-1)(1+o(1))$ *multiplications in* $\mathbb{F}_{2^n}$ *if* $q=2$, *or*

2. $\log_2(n-1)(1+\frac{2}{\log_2\log_2(n-1)}+\frac{2}{\sqrt{\log_2(n-1)}}) + \log_2(q-2)(1+\frac{2}{\log_2\log_2(q-2)}+\frac{2}{\sqrt{\log_2(q-2)}}) + 2 = (\log_2(n-1)(q-2))(1+o(1))$ *multiplications in* $\mathbb{F}_{q^n}$ *if* $q \neq 2$.

The computation needs $n-1$ *further* q*th powers.*

PROOF. Use Lemma 5.2 with the addition chains generated by the algorithm `brauer` (Alg. 3.17) to prove the number of multiplications. The number of qth powers can be seen from Step 7 and Step 10: there are $1+\sum_{i=1}^{L_1} b_{j_i}$ qth powers. We note that the algorithm `brauer` computes an addition chain where all steps including the doublings can be regarded as star steps. Then the following Lemma 5.8 gives the number of qth powers because $1+b_{L_1}-1=b_{L_1}=n-1$.

LEMMA 5.8. *Let* $1=b_0,\ldots,b_{L_1}=n-1$ *be an addition chain only containing star steps. Let* $j_i \in \mathbb{N}$ *with* $0 \leq j_i < i$ *and* $b_i = b_{j_i}+b_{i-1}$ *for all* $1 \leq i \leq L_1$ *according to the addition chain. Then we have for all* $1 \leq l \leq L_1$*:*

$$\sum_{i=1}^{l} b_{j_i} = b_l - 1$$

PROOF. (by induction on l)

$$l=1 \quad : \quad 1+\sum_{i=1}^{1} b_{j_i} = 1+b_{j_1} = 1+b_0 = b_1 = b_l \text{ because } b_0=1, b_1=2.$$

$$l \to l+1 \quad : \quad 1+\sum_{i=1}^{l+1} b_{j_i} = (1+\sum_{i=1}^{l} b_{j_i}) + b_{j_{l+1}} = b_l + b_{j_{l+1}} = b_{l+1}. \ \square$$

Remarks. The algorithm given above is a generalization of an idea of Itoh & Tsujii (1988). They describe an algorithm that is just Algorithm 5.3 for $q=2$ and an addition chain generated by a variation of the binary method (first do all doublings, then compute the star steps). When using the binary method to generate an addition chain for $n-1$ we get the following corollary which is just Theorem 2 in the paper of Itoh & Tsujii (1988):

Corollary 5.9. *Let* $1 = b_0, \ldots, b_{L_1} = n-1$ *an addition chain according to Algorithm* `binary`*. Let* α *be a non–zero element in* $\mathbb{F}_{2^n}$*. Then, there exists an algorithm for computing* α^{-1}*, which requires*

$\lfloor \log_2(n-1) \rfloor + \nu_2(n-1) - 1 \leq 2\lfloor \log_2(n-1) \rfloor$ *multiplications and*
$n-1$ *squarings in* $\mathbb{F}_{2^n}$*.*

Proof. Algorithm `binary` (Algorithm 3.13) generates an addition chain of length $L_1 = \lfloor \log_2(n-1) \rfloor + \nu_2(n-1) - 1$ (Corollary 3.21). Using Algorithm 5.3 with this addition chain we proved the claim for the number of multiplications. To prove the number of squarings we need Lemma 5.8 again because Algorithm `binary` computes an addition chain where all steps including the doublings can be regarded as star steps. According to this lemma we have $b_{L_1} - 1$ squarings in Part B. Together with a further squaring in Part C (Step 10) we get a total number of $b_{L_1} - 1 + 1 = b_{L_1} = n-1$ squarings. □

In Corollary 5.7 we have already shown that Algorithm 5.3 is better than the algorithm of Itoh & Tsujii (1988) in the worst case. We finally give an example to illustrate this:

Example 5.10. *Let* $n = 61$ *and* $\alpha \in \mathbb{F}_{2^{61}}^{\times}$*. Then we can calculate* α^{-1} *using the algorithm of Itoh & Tsujii (1988) with 8 multiplications and 60 squarings in* $\mathbb{F}_{2^{61}}$*. The evaluations are just the same as running Algorithm 5.3 with the addition chain* $1, 2, 4, 8, 16, 32, 48, 56, 60$ *for* $n - 1 = 60$*. But Algorithm* `bgmw` *(Algorithm 3.25) calculates an addition chain for* $n-1 = 60$ *of length only 7:* $1, 2, 4, 8, 16, 20, 40, 60$*. We therefore can evaluate* α^{-1} *using Algorithm 5.3 with 7 multiplications and* $\sum_{i=1}^{7} b_{j_i} = 1 + 2 + 4 + 8 + 4 + 20 + 20 = 59$ *squarings.*

5.2. Calculating the Inverse with Euclid.

The basic idea. Because $\mathbb{F}_{q^n} \cong \mathbb{F}_q[x]/(f)$, where f is an irreducible polynomial of degree n over $\mathbb{F}_q$, any $\alpha \in \mathbb{F}_{q^n}^{\times}$ can be identified with a polynomial $\alpha \in \mathbb{F}_q[x]$ of degree less than n. With the Extended Euclidean Algorithm $s, t \in \mathbb{F}_q[x]$ can be found with $s\alpha + tf = \gcd(\alpha, f)$. Because f is irreducible and $0 \leq \deg \alpha < \deg f$ $\gcd(\alpha, f) = 1$ and therefore $s\alpha \equiv 1 \bmod f$, we have $s \bmod f = \alpha^{-1} \in \mathbb{F}_q[x]/(f) \cong \mathbb{F}_{q^n}$.

Using the classical Extended Euclidean Algorithm we have to calculate all remainders although we are only interested in s and t. But "Euclid's algorithm requires $\frac{n^2-1}{2}$ operations irrespective of the efficiency of multiplication in the [given] domain"(Moenck 1973, p. 143) because of the output size. We therefore have to find a modified algorithm to evaluate s and t.

The basic idea to speed up the evaluation of s, t is to avoid evaluating unnecessary remainders: Let ℓ be the length of the Euclidean scheme for α, f and

$$\begin{aligned} a_{i+1} &= a_{i-1} - a_i q_i \\ s_{i+1} &= s_{i-1} - s_i q_i \\ t_{i+1} &= t_{i-1} - t_i q_i \end{aligned}$$

be the ith step $(1 \le i \le \ell)$ of the Euclidean scheme for α, f. Then we have to calculate $s = s_\ell, t = t_\ell$. But this can be done by evaluating only the quotients q_i:

Remark 5.11. *Let* $Q_i = \begin{pmatrix} 0 & 1 \\ 1 & q_i \end{pmatrix}$, $R_j = \prod_{i=1}^{j} Q_i$ *and* $R_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ *for* $1 \le i, j \le \ell$. *Then* $R_j = \begin{pmatrix} s_j & t_j \\ s_{j+1} & t_{j+1} \end{pmatrix}$, $1 \le j \le \ell$.

Proof. Because $s_0 = 1, s_1 = 0, t_0 = 0, t_1 = 1$, the remark is clear for $j = 0$. So let $j > 0$:

$$\begin{aligned} R_j &= \prod_{i=1}^{j} Q_i = \prod_{i=1}^{j-1} Q_i \cdot Q_j \\ &= \begin{pmatrix} s_{j-1} & t_{j-1} \\ s_j & t_j \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & q_j \end{pmatrix} \\ &= \begin{pmatrix} s_j & t_j \\ s_{j-1} + s_j q_j & t_{j-1} + t_j q_j \end{pmatrix} \\ &= \begin{pmatrix} s_j & t_j \\ s_{j+1} & t_{j+1} \end{pmatrix}. \ \square \end{aligned}$$

But $\begin{pmatrix} \gcd(\alpha, f) \\ * \end{pmatrix} = \begin{pmatrix} s_\ell \alpha + t_\ell f \\ * \end{pmatrix} = \begin{pmatrix} s_\ell & t_\ell \\ * & * \end{pmatrix} \begin{pmatrix} \alpha \\ f \end{pmatrix} = R_\ell \begin{pmatrix} \alpha \\ f \end{pmatrix}$.

The Fast Euclidean Algorithm. A divide–and–conquer algorithm for integers based on this idea was developed by Knuth (1981) and Schönhage (1971). Moenck (1973) has generalized it to any Euclidean domain. A presentation of the so called *Fast Euclidean Algorithm* can be found in the article of Strassen (1983). We concentrate on the results here. In order to abstract from the underlying multiplication algorithm, we introduce the following function (cf. von zur Gathen & Gerhard 1995).

DEFINITION 5.12. *Let R be a ring. A function* $\mathsf{M}\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ *is called* a multiplication time *for $R[x]$ if polynomials in $R[x]$ of degree less than n can be multiplied using $O(\mathsf{M}(n))$ operations in R.*

THEOREM 5.13. *The* gcd *of two univariate polynomials over a finite field $\mathbb{F}_{q^n}$ can be computed in $O(\mathsf{M}(n)\log n)$ operations in $\mathbb{F}_q$ where $\mathsf{M}(n)$ denotes the number of operations in $\mathbb{F}_q$ to multiply two elements of $\mathbb{F}_{q^n}$. It is assumed that $\mathsf{M}(n) \geq n$ and $\mathsf{M}(2n) \geq 2\mathsf{M}(n)$.*

PROOF. See Moenck (1973). □

With the argumentation above we get

COROLLARY 5.14. *For given $\alpha \in \mathbb{F}_{q^n}^{\times}$ the inverse $\alpha^{-1} \in \mathbb{F}_{q^n}^{\times}$ can be calculated with $O(\mathsf{M}(n)\log n)$ operations in $\mathbb{F}_q$ where $\mathsf{M}(n)$ denotes the number of operations in $\mathbb{F}_q$ to multiply two elements of $\mathbb{F}_{q^n}$.*

5.3. Comparison. We have introduced two methods to invert $\alpha \in \mathbb{F}_{q^n}^{\times}$. The method based on Fermat needs $O(\mathsf{M}(n))\log(n)(1 + o(1))$ operations in $\mathbb{F}_q$ if raising to the qth power is for free. This assumption can be made using a normal basis representation of $\mathbb{F}_{q^n}$ (see Section8.1). Euclid's algorithm uses $O(\mathsf{M}(n)\log(n))$ operations in $\mathbb{F}_q$ as well and works on a power basis representation of $\mathbb{F}_{q^n}$. (We deal with the topic of representation of finite fields in the next section.) But we found no reference about the constant hidden behind the O-notation in literature. It can be estimated that the constant is greater than the one for Fermat's method due to the results of an implementation of the Fast Euclidean Algorithm given in citegatger96a where the constant is about 3.

6. Finite fields

6.1. Introduction. Up to now we examined ways to reduce the number of multiplications which are needed for the computation of $b^e \in G$ for given $b \in G, e \in \mathbb{N}$, and G an arbitrary multiplicative group. The second point to deal with is to speed up the time needed for a single multiplication or raising to a determined power, respectively.

We concentrate on the special case of finite fields in the following, i.e. $G = \mathbb{F}_{q^n}^\times$ where $q = p^t$ with $p = \mathrm{char}\mathbb{F}_q$ is a prime, $t \in \mathbb{N}$, and $n \in \mathbb{N}$. The results of the previous parts distinguished between multiplication on the one hand and squaring and raising to the qth power on the other hand. We will continue this separation when discussing how to speed up basic arithmetic operations in $\mathbb{F}_{q^n}$.

6.2. Definitions. We recall some definitions that are needed in the sequel:

DEFINITION 6.1. *Let $f \in \mathbb{F}_q[x]$ with $\deg f = n$ and $f = \sum_{0 \leq i \leq n} f_i x^i$. Then a field extension $\mathbb{E}$ of $\mathbb{F}_q$ is called* a splitting field *of f over $\mathbb{F}_q$ if*

1. *there exist elements $\theta_1, \ldots, \theta_n \in \mathbb{E}$ such that $f(x) = f_n \prod_{1 \leq i \leq n}(x - \theta_i)$ and*
2. *$\mathbb{E} = \mathbb{F}_q(\theta_1, \ldots, \theta_n)$.*

$\theta_1, \ldots, \theta_n$ are called the roots *of f.*

DEFINITION 6.2. *Let $\mathbb{F}_{q^n}$ be an extension of $\mathbb{F}_q$ and let $\alpha \in \mathbb{F}_{q^n}$. Then the elements $\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{n-1}}$ are called the* conjugates *of α with respect to $\mathbb{F}_q$.*

DEFINITION 6.3. *Let $\mathbb{E}$ be the splitting field of $x^n - 1$ over $\mathbb{F}_q$ and $\gcd(n, q) = 1$. Then the roots $\zeta_1, \ldots, \zeta_n$ of $x^n - 1$ are called the* nth roots of unity *over $\mathbb{F}_q$.*

RESULT 6.4. *The set of all nth roots of unity over $\mathbb{F}_q\}$ is a cyclic subgroup of the splitting field of $x^n - 1$ over $\mathbb{F}_q$ with respect to multiplication.*

PROOF. Cf. Lidl & Niederreiter (1983), Theorem 2.42. □

DEFINITION 6.5. *Let ζ be an nth root of unity over $\mathbb{F}_q$. If ζ generates a multiplicative subgroup of order n in the splitting field of $x^n - 1 \in \mathbb{F}_q[x]$ then ζ is called* a primitive nth root of unity *over $\mathbb{F}_q$.*

NOTATION 6.6. *Let G be a group, and $g_1, \ldots, g_n \in G$. A subgroup $U < G$ generated by $g_1, \ldots, g_n$ is written $U = \langle g_1, \ldots, g_n \rangle$.*

DEFINITION 6.7. *Let q a prime power, and $r \in \mathbb{N}$ with $\gcd(q, r) = 1$. Let $\zeta \in \mathbb{F}_{q^r}$ be a primitive rth root of unity in $\mathbb{F}_{q^r}$. Then the polynomial*

$$\Phi_r(x) = \prod_{\substack{1 \le i \le r \\ \gcd(i,r)=1}} (x - \zeta^i) \in \mathbb{F}_q[x]$$

is the rth cyclotomic polynomial *over $\mathbb{F}_q$.*

6.3. The representation of finite fields. One crucial point when examining basic arithmetic operations in $\mathbb{F}_{q^n}$ in detail is the representation of the elements of a finite field.

We can regard $\mathbb{F}_{q^n}$ as a vector space of dimension n over $\mathbb{F}_q$. Thus $\mathbb{F}_{q^n}$ can be identified with $\mathbb{F}_q^n$. If $\alpha_0, \ldots, \alpha_{n-1} \in \mathbb{F}_{q^n}$ form a basis of $\mathbb{F}_q^n$ over $\mathbb{F}_q$, $\alpha \in \mathbb{F}_{q^n}$ can be uniquely written as $\alpha = \sum_{0 \le i < n} a_i \alpha_i =: (a_0, \ldots, a_{n-1})$ (see Menezes *et al.* 1993).

There are three special kinds of bases commonly used to implement efficient arithmetic in finite fields:

1. Remember the following:

   THEOREM 6.8. *Let f be an irreducible polynomial in $\mathbb{F}_q[x]$ of degree n. Then the splitting field of f over $\mathbb{F}_q$ is given by $\mathbb{F}_{q^n}$.*

   PROOF. See Lidl & Niederreiter (1983), Corollary 2.15. □

   Because of this theorem we have $\mathbb{F}_{q^n} \cong \mathbb{F}_q[x]/(f)$ and any $\alpha \in \mathbb{F}_{q^n}$ can be represented by a polynomial of degree at most $n-1$ over $\mathbb{F}_q$. So arithmetic here means polynomial arithmetic in $\mathbb{F}_q[x]$ modulo f. We call this a *polynomial representation* of $\mathbb{F}_{q^n}$. If $\alpha = x \bmod f$ in $\mathbb{F}_q[x]/(f)$, then $\mathcal{B} = (1, \alpha, \ldots, \alpha^{n-1})$ is a basis for $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$.

2. DEFINITION 6.9. *A* normal basis *$\mathcal{N} = (\alpha_0, \ldots, \alpha_{n-1})$ of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ is a basis with*

   $$\alpha_0, \alpha_1 = \alpha_0^q, \ldots, \alpha_{n-1} = \alpha_0^{q^{n-1}}.$$

   In this case, $\alpha_0 \in \mathbb{F}_{q^n}$ is called a normal basis generator *or* a normal element *over $\mathbb{F}_q$.*

   This is called a *normal basis representation* of $\mathbb{F}_{q^n}$.

3. Let $\zeta \in \mathbb{F}_{q^n}$ be primitive. Then we can represent $\alpha \in \mathbb{F}_{q^n} \setminus \{0\}$ by $\log_\zeta \alpha \in \mathbb{N}$, with $0 \leq \log_\zeta \alpha \leq q^n - 2$. This can be used to implement arithmetic efficiently in small finite fields, with the help of exp– and log–tables stored in main memory. We do not discuss this *primitive element representation* of $\mathbb{F}_{q^n}$ in the sequel.

So we have two possible ways to represent the elements of $\mathbb{F}_{q^n}$. We examine the differences of these representations for the three basic arithmetic operations we are mainly interested in: addition, multiplication, and exponentiation, in particular raising to the qth power.

7. Polynomial representation

7.1. Irreducible polynomials. If $\mathbb{F}_q[x]$ is the polynomial ring in one variable over $\mathbb{F}_q$ and $f \in \mathbb{F}_q[x]$ is a monic irreducible polynomial of degree n, then $\mathbb{F}_{q^n} \cong \mathbb{F}_q[x]/(f)$ (see Theorem 6.8) where $\mathbb{F}_q[x]/(f)$ is the *residue class ring.*

THEOREM 7.1. *Let $f \in \mathbb{F}_q[x]$ be irreducible and monic and θ a root of f with $f(\theta) = 0$. Then $\mathbb{F}_q/(f) \cong \mathbb{F}_q(\theta)$ and $\mathcal{B} = (1, \theta, \ldots, \theta^{n-1})$ is a basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$.*

PROOF. Cf. Lidl & Niederreiter (1983), Theorem 1.86. □

So every $g \in \mathbb{F}_{q^n}$ can be represented by a polynomial of degree at most $n - 1$ in $\mathbb{F}_q[x]$ and polynomial arithmetic can be used. This is the *polynomial representation* of a finite field.

Addition. Addition is component–wise: Let $g, h \in \mathbb{F}_q[x]/(f)$. Then we have $g = \sum_{0\leq i<n} g_i x^i =: (g_0, \ldots, g_{n-1})$ and $h = \sum_{0\leq i<n} h_i x^i =: (h_0, \ldots, h_{n-1})$ and $g + h = \sum_{0\leq i<n}(g_i + h_i)x^i = (g_0 + h_0, \ldots, g_{n-1} + h_{n-1})$. We therefore need n additions in $\mathbb{F}_q$ to do one addition in $\mathbb{F}_q[x]/(f)$.

Multiplication. Let $g = \sum_{0\leq i<n} g_i x^i, h = \sum_{0\leq i<m} h_i x^i \in \mathbb{F}_q[x]$ be two polynomials of degree less than n and m, respectively. The 'classical method' to multiply $g, h \in \mathbb{F}_q[x]/(f)$ proceeds in two steps (cf. Brunner *et al.* 1993):

1. $gh = (\sum_{0\leq i<n} g_i x^i)(\sum_{0\leq i<m} h_i x^i) = \sum_{0\leq i<mn-1}(\sum_{j+k=i} g_j + h_k)x^i \in \mathbb{F}_q[x]$. This can be done using $\Theta(mn)$ additions in $\mathbb{F}_q$.
2. Calculate $gh = uf + v$ with $u, v \in \mathbb{F}_q[x]$ and $v = 0$ or $0 \leq \deg v < \deg f$. Then $v = gh \in \mathbb{F}_q[x]/(f)$.

This way to multiply two polynomials in $\mathbb{F}_q[x]/(f)$ needs $O(n^2)$ operations in $\mathbb{F}_q$. We now introduce faster algorithms to multiply two polynomials $g, h \in \mathbb{F}_q[x]$. We assume $\deg g = \deg h$ which is the worst choice.

7.2. Fast multiplication for polynomials.

The algorithm of Karatsuba & Ofman. The first algorithm beating the asymptotical bound of $O(n^2)$ was given by Karatsuba & Ofman (1962). It is based on the divide–and–conquer strategy.

Let $g, h \in \mathbb{F}_q[x]$ as before with $n = m = 2^t, t \in \mathbb{N}_0$. Then we divide g, h into two polynomials of degree at most $\frac{m}{2} - 1$ each:

$$\begin{aligned} g &= \sum_{0 \le i < \frac{m}{2}} g_i x^i + x^{\frac{m}{2}} \sum_{0 \le i < \frac{m}{2}} g_{i+\frac{m}{2}} x^i = G_1 + x^{\frac{m}{2}} G_2, \\ h &= \sum_{0 \le i < \frac{m}{2}} h_i x^i + x^{\frac{m}{2}} \sum_{0 \le i < \frac{m}{2}} h_{i+\frac{m}{2}} x^i = H_1 + x^{\frac{m}{2}} H_2. \end{aligned}$$

Then $\deg G_1, \deg G_2, \deg H_1, \deg H_2 < \frac{m}{2}$ and we have

$$\begin{aligned} g \cdot h &= (G_1 + x^{\frac{m}{2}} G_2)(H_1 + x^{\frac{m}{2}} H_2) \\ &= G_1 H_1 + (G_1 H_2 + G_2 H_1) x^{\frac{m}{2}} + G_2 H_2 x^m \\ &= G_1 H_1 + ((G_1 + G_2)(H_1 + H_2) - G_1 H_1 - G_2 H_2) x^{\frac{m}{2}} + G_2 H_2 x^m \end{aligned}$$

So the problem of multiplying two polynomials of degree $< m$ is reduced to *three* multiplications of polynomials of degree $< \frac{m}{2}$ and $4m$ additions in $\mathbb{F}_q$.

LEMMA 7.2. *Two polynomials in $\mathbb{F}_q[x]$ of degree less than $m = 2^t, t \in \mathbb{N}_0$, can be multiplied with $O(m^{\log_2 3})$ operations in $\mathbb{F}_q$.*

PROOF. (cf. von zur Gathen & Gerhard 1995) Let $T(m)$ denote the number of operations in $\mathbb{F}_q$ to multiply two polynomials of degree less than m. If $T(1) = 1$ and $T(2^t) \le 3T(2^{t-1}) + c2^t$ with some constant c for $t > 0$, then $T(2^t) \le (1 + 2c)3^t - 2c \cdot 2^t$ for $t \ge 0$. This can be shown by induction on t.

But Karatsuba & Ofman's algorithm satisfies exactly the conditions on T and therefore needs $\le (1 + 2c)m^{\log_2 3} - 2cm = O(m^{\log_2 3})$ operations in $\mathbb{F}_q$. $\square$

Fast multiplication using the Fast Fourier Transformation. We usually represent a polynomial $f = \sum_{0 \le i < m} f_i x^i \in \mathbb{F}_q[x]$ by its *coefficient list* $(f_0, \ldots, f_{m-1}) \in \mathbb{F}_q$. Another way is given by the *value representation* (see von zur Gathen & Gerhard 1995): f is given by $f(u_i) \in \mathbb{F}_q(u_0, \ldots, u_{m-1})$ for $u_0, \ldots, u_{m-1} \in \mathbb{F}_q(u_0, \ldots, u_{m-1})$. If $g, h \in \mathbb{F}_q[x]$ are represented by values with $g(u_0), \ldots, g(u_{2m-1}), h(u_0), \ldots, h(u_{2m-1})$ then multiplication is quite easy: $(g \cdot h)(u_i) = g(u_i) \cdot h(u_i)$ for all $0 \le i < 2m$. So two polynomials of degree less than m represented by $2m$ values can be multiplied using $2m$ multiplications in $\mathbb{F}_q(u_0, \ldots, u_{2m-1})$. Hence, one possibility to speed up polynomial multiplication is to concentrate on a fast transformation between coefficient representation and value representation. This idea was first used for fast multiplication by Schönhage & Strassen (1971). Our exposition follows von zur

Gathen & Gerhard (1995) and Aho *et al.* (1974).

Let ζ be a primitive mth root of unity in a field extension of $\mathbb{F}_q$ and $f = \sum_{0\le i<m} f_i x^i \in \mathbb{F}_q[x]$.

DEFINITION 7.3. *The map*

$$\begin{array}{rrcl} \mathrm{DFT}_\zeta\colon & \mathbb{F}_q^m & \to & \mathbb{F}_q(\zeta)^m \\ & (f_0,\ldots,f_{m-1}) & \mapsto & (f(1), f(\zeta),\ldots,f(\zeta^{m-1})) \end{array}$$

which evaluates a polynomial at the powers of ζ is called the Discrete Fourier Transformation (DFT).

DFT_ζ is the transformation we search for because the following holds:

LEMMA 7.4. *For polynomials $g, h \in \mathbb{F}_q[x]$ of degree less than $\frac{m}{2}$ we have*

$$\mathrm{DFT}_\zeta(g *_m h) = \mathrm{DFT}_\zeta(g)\cdot\mathrm{DFT}_\zeta(h)$$

where $\cdot$ denotes the pointwise multiplication of vectors and $*_m$ denotes the multiplication of two polynomials modulo (x^m-1).

PROOF. We have $g*_m h \equiv gh \bmod (x^m-1)$ and $\deg(gh) < 2\frac{m}{2} = m$. Therefore $g*_m h = gh + s(x^m-1)$ with $s\in\mathbb{F}_q[x]$ and $(g*_m h)(\zeta^i) = g(\zeta^i)h(\zeta^i) + s(\zeta^i)(\zeta^{im}-1) = g(\zeta^i)h(\zeta^i)$ for $0\le i<m$. □

NOTATION 7.5. *Let $V_\zeta = (\zeta^{ij})_{0\le i,j<m} \in \mathbb{F}_q(\zeta)^{m\times m}$. V_ζ is called the* Vandermonde matrix.

LEMMA 7.6. *The inverse of $V_\zeta \in \mathbb{F}_q(\zeta)^{m\times m}$ exists and is given by*

$$V_\zeta^{-1} = (\frac{1}{m}\zeta^{-ij})_{0\le i,j<m} = \frac{1}{m}V_{\zeta^{-1}}.$$

PROOF. Cf. Aho *et al.* (1974), Lemma 7.1. □

Using the fact that

$$\begin{aligned} \mathrm{DFT}_\zeta(f) &= ({}^t(f(1), f(\zeta),\ldots,f(\zeta^{m-1}))) \\ &= ({}^t(\sum_{0\le i<m} f_i 1^i, \sum_{0\le i<m} f_i\zeta^i,\ldots,\sum_{0\le i<m} f_i(\zeta^{m-1})^i)) \\ &= V_\zeta({}^t(f_0,\ldots,f_{m-1})) \end{aligned}$$

we obtain the following corollary.

Corollary 7.7. $\mathrm{DFT}_\zeta^{-1} = \frac{1}{m}\mathrm{DFT}_{\zeta^{-1}}$.

Proof. Let $f \in \mathbb{F}_q[x]$. Then we have

$$\begin{aligned} & \mathrm{DFT}_\zeta(\frac{1}{m}\mathrm{DFT}_{\zeta^{-1}})(f) \\ = \; & \frac{1}{m}V_\zeta V_{\zeta^{-1}}({}^t(f_0,\ldots,f_{m-1})) \\ = \; & E_m({}^t(f_0,\ldots,f_{m-1})) = f \end{aligned}$$

and so $\frac{1}{m}\mathrm{DFT}_{\zeta^{-1}} = \mathrm{DFT}_\zeta^{-1}$. □

Therefore the inverse DFT can be calculated quickly if we find an algorithm to compute the DFT quickly.

If we evaluate DFT_ζ using V_ζ directly then we need $O(m^2)$ operations in $\mathbb{F}_q(\zeta)$ for arbitrary m (cf. Aho *et al.* 1974). But we can do better for $m = 2^t$, $t \in \mathbb{N}_0$. Let

$$f \;=\; \sum_{0\le j<\frac{m}{2}} f_{2j}(x^2)^j + x\sum_{0\le j<\frac{m}{2}} f_{2j+1}(x^2)^j = F_1(x^2) + xF_2(x^2) \qquad (7.1)$$

with $F_1, F_2 \in \mathbb{F}_q[x]$ and $\deg F_1, \deg F_2 < \frac{m}{2}$. But because ζ is a primitive mth root of unity we have

$$\zeta^m = 1 \text{ and } \zeta^{\frac{m}{2}+j} = \zeta^{\frac{m}{2}}\zeta^j = -\zeta^j \text{ for } 0 \le j < \frac{m}{2}$$

and

$$f(\zeta^{j+\frac{m}{2}}) = F_1(\zeta^{2j}\zeta^m) + \zeta^{j+\frac{m}{2}}F_2(\zeta^{2j}\zeta^m) = F_1(\zeta^{2j}) - \zeta^j F_2(\zeta^{2j}). \qquad (7.2)$$

To evaluate $\mathrm{DFT}_\zeta(f)$ where $\deg f < m$ and is ζ a primitive mth root of unity we have to evaluate $\mathrm{DFT}_{\zeta^2}(F_i)$ with $\deg F_i < \frac{m}{2}$ and ζ^2 a primitive $\frac{m}{2}$th root of unity for $i = 1, 2$. Using this idea recursively we get an algorithm known as the *Fast Fourier Transformation*. This algorithm is due to Schönhage & Strassen (1971). We describe it according to von zur Gathen & Gerhard (1995).

Algorithm 7.8. FFT
Input: $m = 2^t, t \in \mathbb{N}_0, f = \sum_{0\le i<m} f_i x^i \in \mathbb{F}_q[x]$, *and the powers* $\zeta, \zeta^2, \ldots, \zeta^{m-1}$ *of a primitive* m*th root of unity* ζ *in a field extension of* $\mathbb{F}_q$.
Output: $DFT_\zeta(f) = (f(1), f(\zeta), \ldots, f(\zeta^{m-1})) \in \mathbb{F}_q(\zeta)$.

1. *If* $m = 1$ *then return* (f_0).
2. *Let* $f = F_1(x^2) + xF_2(x^2)$ *with* $F_1, F_2 \in \mathbb{F}_q[x], \deg F_1, \deg F_2 < \frac{m}{2}$.
3. *Recursively compute* $(G_i)_{0 \le i < \frac{m}{2}} = \mathtt{FFT}(\frac{m}{2}, F_1, \zeta^2, \zeta^4, \ldots, \zeta^m)$ *and* $(H_i)_{0 \le i < \frac{m}{2}} = \mathtt{FFT}(\frac{m}{2}, F_2, \zeta^2, \zeta^4, \ldots, \zeta^m)$.
4. *Compute* $F^{(i)} = G_i + \zeta^i H_i$ *and* $F^{(i+\frac{m}{2})} = G_i - \zeta^i H_i$ *for all* $0 \le i < \frac{m}{2}$.
5. *Return* $(F^{(0)}, \ldots, F^{(m-1)})$.

LEMMA 7.9. *Algorithm* `FFT` *computes* DFT_ζ *as specified. It uses* $O(m \log m)$ *operations in* $\mathbb{F}_q(\zeta)$ *for* $m = 2^t, t \in \mathbb{N}_0$.

PROOF.

1. Correctness (by induction on t): For $t = 0$ correctness is clear. Assume now that Algorithm `FFT` works correctly for $m = 2^t$ and let ζ be a primitive $2m$th root of unity. Then ζ^2 is a primitive mth root of unity and we have $G_i = F_1(\zeta^{2i})$ and $H_i = F_2(\zeta^{2i})$ for all $0 \le i < m$ by induction hypothesis. For $0 \le i < m$ we have $F^{(i)} = F_1(\zeta^{2i}) + \zeta^i F_2(\zeta^{2i}) \overset{(7.1)}{=} f(\zeta^i)$ and $F^{(i+m)} = F_1(\zeta^{2(i+m)}) - \zeta^i F_2(\zeta^{2i}) \overset{(7.2)}{=} f(\zeta^{i+m})$. Thus `FFT` works correctly.
2. Number of operations: Let $T(m)$ denote the number of operations in $\mathbb{F}_q(\zeta)$ for Algorithm `FFT` on input m. Then Steps 1+2 need no operations, Step 3 needs $2T(\frac{m}{2})$ and Step 4 $\frac{m}{2}$ multiplications and m additions (by first evaluating $\zeta^i H_i$). We have $T(1) = 0$ and thus the recursion for $m = 2^t$:

$$\begin{aligned} T(2^t) &= 2^1 T(2^{t-1}) + \frac{3}{2}m \\ &= 2^2 T(2^{t-2}) + 2^1 \cdot \frac{3}{2}\frac{m}{2^1} + \frac{3}{2}m = 2^2 T(2^{t-2}) + 2\frac{3}{2}m \\ &= \ldots \\ &= 2^t T(1) + \frac{3}{2}mt = \frac{3}{2}m \log_2 m \in O(m \log m) \end{aligned}$$

We are now ready to give an algorithm for fast polynomial multiplication:

ALGORITHM 7.10. `polynomial multiplication using FFT`
Input: $g, h \in \mathbb{F}_q[x], \deg g, \deg h < m = 2^t, t \in \mathbb{N}_0$, *and* ζ *a primitive* $2m$*th root of unity in a field extension of* $\mathbb{F}_q$.
Output: $gh \in \mathbb{F}_q[x]$.

1. *Compute* $\zeta^2, \ldots, \zeta^{2m-1} \in \mathbb{F}_q(\zeta)$.
2. *Compute* $G = \mathrm{DFT}_\zeta(g)$ *and* $H = \mathrm{DFT}_\zeta(h) \in \mathbb{F}_q(\zeta)$.
3. *Compute* $F = G \cdot H \in \mathbb{F}_q(\zeta)$.
4. *Return* $DFT_\zeta^{-1}(F) = \frac{1}{2m}\mathrm{DFT}_{\zeta^{-1}}(F)$

LEMMA 7.11. *Algorithm* `polynomial multiplication using FFT` *computes* $gh \in \mathbb{F}_q[x]$ *and uses at most* $9m\log_2(2m) + 6m - 2$ *operations in* $\mathbb{F}_q(\zeta)$.

PROOF. Correctness follows directly from Lemma 7.4. The cost is given by at most $2m-2$ multiplications in Step 1, $2\frac{3}{2}(2m)\log_2(2m)$ operations in Step 2, $2m$ multiplications in Step 3 and $\frac{3}{2}(2m)\log_2(2m) + 2m$ operations in Step 4. □

COROLLARY 7.12. *Let* $m = 2^t, t \in \mathbb{N}_0$ *and* q *odd. In a field extension of* $\mathbb{F}_q$ *containing a primitive* $2m$*th root of unity, two arbitrary polynomials of degree less than* m *can be multiplied using* $O(m \log m)$ *operations in this extension of* $\mathbb{F}_q$.

Fast multiplication over arbitrary finite fields. The given algorithm for fast polynomial multiplication works only if there exists a primitive 2^tth root of unity in a field extension of $\mathbb{F}_q$. If $q = 2$ there exists no such root because $1 + 1 = 0$. But the idea can be generalized to all finite fields.

THEOREM 7.13 (SCHÖNHAGE 1977). *Let* $m = 3^t$ *and* ζ *be a primitive* $3m$*th root of unity in a field extension of* $\mathbb{F}_q$. *Two arbitrary polynomials of degree less than* m *can be multiplied with* $O(m \log m \log\log m)$ *operations in* $\mathbb{F}_q(\zeta)$.

PROOF. See Schönhage (1977). □

THEOREM 7.14 (CANTOR 1989). *Two polynomials of degree less than* m *over* $\mathbb{F}_q[x]$ *can be multiplied using* $O(m(\log m)^3)$ *operations in* $\mathbb{F}_q$.

PROOF. See Cantor (1989). This multiplication algorithm uses an analogue to the Fast Fourier Transformation for additive subgroups. □

THEOREM 7.15 (CANTOR & KALTOFEN 1991). *The product of two polynomials of degree less than* m *with coefficients in* $\mathbb{F}_q$ *can be computed using* $O(m \log m \log\log m)$ *additions/subtractions and* $O(m \log m)$ *multiplications in* $\mathbb{F}_q$.

PROOF. See Cantor & Kaltofen (1991). □

7.3. Modular composition. We have concentrated on fast polynomial multiplication so far. But to speed up exponentiation we also have to concern with algorithms for raising to a determined power. Shoup (1994) suggests an algorithm using modular composition based on fast matrix multiplication.

The 'classical' algorithm for matrix multiplication. Let $m, n, k \in \mathbb{N}$ and $A = (a_{ij}) \in \mathbb{F}_q^{m\times n}, B = (b_{ij}) \in \mathbb{F}_q^{n\times k}$ be two matrices. Then we can compute $AB = C = (c_{ij}) \in \mathbb{F}_q^{m\times k}$ according to the definition:

$$c_{ij} = \sum_{1\le s\le n} a_{is}b_{sj} \text{ for all } 1 \le i \le m, 1 \le j \le k.$$

We need n multiplications and $n-1$ additions in $\mathbb{F}_q$ for each i and j and hence a total number of $m(2n-1)k$ operations in $\mathbb{F}_q$.

We can concentrate on square matrices $A, B \in \mathbb{F}_q^{n\times n}$ in the following because padding by zeros reduces the general problem to this special case, as follows:

ALGORITHM 7.16. `matrix to square`
Input: $m, n, k \in \mathbb{N}$ *and* $A = (a_{ij}) \in \mathbb{F}_q^{m\times n}, B = (b_{ij}) \in \mathbb{F}_q^{n\times k}$.
Output: $C = (c_{ij}) = AB \in \mathbb{F}_q^{m\times k}$.

1. *Let* $s, t \in \mathbb{N}$ *with* $(s-1)n < m \le sn$ *and* $(t-1)n < k \le tn$.
2. *Define* $a'_{ij} = \begin{cases} a_{ij} & 1 \le i \le m \text{ and } 1 \le j \le n \\ 0 & \text{else} \end{cases}$
   and $b'_{ij} = \begin{cases} b_{ij} & 1 \le i \le n \text{ and } 1 \le j \le k \\ 0 & \text{else} \end{cases}$.
3. *For all* $i' \in \{1,\ldots,s\}$ *and* $j' \in \{1,\ldots,t\}$ *define* $A_{i'}, B_{j'}, C_{i'j'} \in \mathbb{F}_q^{n\times n}$ *by*
$$\begin{aligned} A_{i'} &:= (a'_{ij})_{(i'-1)n<i\le i'n, 1<j\le n}, \\ B_{j'} &:= (b'_{ij})_{1<i\le n, (j'-1)n<j\le j'n}, \\ C_{i'j'} &:= (c'_{ij})_{(i'-1)n<i\le i'n, (j'-1)n<j\le j'n}. \end{aligned}$$
4. *Compute* $C_{i'j'} = A_{i'}B_{j'}$ *for all* $i' \in \{1,\ldots,s\}$ *and* $j' \in \{1,\ldots,t\}$.
5. *Return* $C = (c'_{ij})_{1\le i\le m, 1\le j\le k}$.

DEFINITION 7.17. *Let R be a ring. A real number $\omega \in \mathbb{R}_{>0}$ is called* a matrix multiplication exponent *if matrices in $R^{n\times n}$ can be multiplied using $O(n^\omega)$ operations in R.*

LEMMA 7.18. *The algorithm* `matrix to square` *works correctly. The algorithm uses*

$$\left\lceil \frac{m}{n} \right\rceil \left\lceil \frac{k}{n} \right\rceil O(n^\omega)$$

operations in $\mathbb{F}_q$.

PROOF. Correctness is clear. The number of operations can be seen directly connecting Step 1 and Step 4. □

Strassen's algorithm. Strassen (1969) uses a divide–and–conquer algorithm to multiply two square matrices. For convenience we assume $m = 2^t, t \in \mathbb{N}_0$.

THEOREM 7.19 (STRASSEN 1969). *Two square matrices* $A, B \in \mathbb{F}_q^{m \times m}$ *with* $m = 2^t, t \in \mathbb{N}_0$ *can be multiplied with* $O(m^\omega)$ *operations in* $\mathbb{F}_q$ *with* $\omega = \log_2 7 \approx 2.80735492$.

To proof this theorem we need the following lemma:

LEMMA 7.20. *If* $T(1) = 1$ *and* $T(2^t) \le 7T(2^{t-1}) + c(2^t)^2$ *with some constant* c *for* $t > 0$, *then* $T(2^t) \le (1 + \frac{4}{3}c)7^t - \frac{4}{3}c(2^t)^2$.

PROOF. (by induction on t) $t = 0$ is clear. For $t > 0$ we have

$$\begin{aligned}
T(2^t) &\le 7T(2^{t-1}) + c(2^t)^2 \\
&\le 7((1 + \frac{4}{3}c)7^{t-1} - \frac{4}{3}c(2^{t-1})^2) + c(2^t)^2 \\
&= (1 + \frac{4}{3}c)7^t - \frac{7 \cdot 4}{3 \cdot 4}c2^{2t} + c2^{2t} \\
&= (1 + \frac{4}{3}c)7^t - \frac{4}{3}c(2^t)^2. \ \square
\end{aligned}$$

We can now prove Theorem 7.19.
PROOF. Let $A, B, C \in \mathbb{F}_q^{2^t \times 2^t}$ and write

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

where $A_{ij}, B_{ij}, C_{ij} \in M(2^{t-1}; \mathbb{F}_q)$ for $i, j = 1, 2$. Then compute

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\
M_2 &= (A_{21} + A_{22})B_{11},
\end{aligned}$$

$$\begin{aligned}
M_3 &= A_{11}(B_{12} - B_{22}),\\
M_4 &= A_{22}(B_{21} - B_{11}),\\
M_5 &= (A_{11} + A_{12})B_{22},\\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}),\\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}),\\
\text{and } C_{11} &= M_1 + M_4 - M_5 + M_7,\\
C_{12} &= M_2 + M_4,\\
C_{21} &= M_3 + M_5,\\
C_{22} &= M_1 + M_3 - M_2 + M_6.
\end{aligned}$$

It can be easily shown that we have indeed $C = AB$. We use 7 multiplications and 18 additions of matrices in $\mathbb{F}_q^{2^{t-1}\times 2^{t-1}}$. One addition can be done using $(2^{t-1})^2$ additions in $\mathbb{F}_q$. For $m = 2^0 = 1$ we have only one multiplication in $\mathbb{F}_q$. If $T(m)$ denotes the number of operations in $\mathbb{F}_q$ to multiply two matrices in $\mathbb{F}_q^{m\times m}$ we get the recursion

$$T(1) = 1 \text{ and } T(2^t) = 7T(2^{t-1}) + c(2^{t-1})^2 \text{ for } t > 0.$$

According to Lemma 7.20 we have $T(m) \le (1 + \frac{4}{3}c)n^{\log_2 7} - \frac{4}{3}cn^2 \in O(n^{\log_2 7})$. □

A slightly better version of Strassen's algorithm is given by Winograd (1971). This version uses only 7 multiplications and 15 additions. The identities can be found in Aho *et al.* (1974).

Strassen's result for ω has been improved by other authors. An overview about the early history of fast matrix multiplication is given by Pan (1984). The currently best known value for ω is $\omega < 2.376$ (cf. Coppersmith & Winograd 1990). "Because of the hidden constants involved, however, none of the algorithms found after Strassen's is of much practical use."(Brassard & Bratley 1988, p. 243)

Exponentiation and modular composition. A basic tool in the algorithm of Shoup (1994) is the calculation of modular compositions as introduced in the *iterated frobenius* algorithm of von zur Gathen & Shoup (1992). Let $f, g, h \in \mathbb{F}_q[x]$ with $\deg f = n$ and $\deg g, \deg h < n$. The *modular composition* of g and h is given by $g(h) \bmod f$.

Let $f, g \in \mathbb{F}_q$ and $R = \mathbb{F}_q[x]/(f)$. In the following we have to distinguish between the image of g in R and the polynomial in $\mathbb{F}_q[x]$ obtained as the remainder on dividing g by f. This leads to the following notation (cf. von zur Gathen & Shoup 1992):

NOTATION 7.21. *Let f, g, R as above.*

1. *The* image *of g in R is denoted by $(g \bmod f)$, and the* remainder on dividing *g by f is denoted by $(g$ rem $f)$.*
2. *For $\alpha \in R$, the* canonical representative *of α is the unique polynomial $a \in \mathbb{F}_q[x]$ of degree less than n such that $(a \bmod f) = \alpha$.*

LEMMA 7.22. *Let $f, g, h \in \mathbb{F}_q[x]$, $r \in \mathbb{N}$ with $h = x^{q^r}$ rem f. Then $g^{q^r} \equiv g(h) \bmod f$.*

PROOF. Let without loss of generality g be reduced modulo f and $\deg g < n = \deg f$, that is $g = \sum_{0 \leq i < n} g_i x^i$, $g_0, \ldots, g_{n-1} \in \mathbb{F}_q$. Then $g(h) = \sum_{0 \leq i < n} g_i h^i \equiv \sum_{0 \leq i < n} g_i (x^{q^r})^i = \sum_{0 \leq i < n} g_i (x^i)^{q^r} = (\sum_{0 \leq i < n} g_i x^i)^{q^r} \equiv g^{q^r} \bmod f$. □

Hence we can use modular composition to raise to the q^rth power in $\mathbb{F}_q[x]/(f)$ for any $r \in \mathbb{N}$.

A fast algorithm for modular composition. Because we have $\deg f = n$ and we can assume $\deg g, \deg h < n$, we have to evaluate the first n coefficients of $g(h)$ rem f. Let $g = \sum_{0 \leq i < n} g_i x^i, h = \sum_{0 \leq i < n} h_i x^i$ with $g_i = 0$ for $\deg g < i < n$ and $h_i = 0$ for $\deg h < i < n$. Let $k = \lceil \sqrt{n}\, \rceil$. Then $l = \lceil \frac{n}{k} \rceil \leq \left\lceil \frac{n}{\sqrt{n}} \right\rceil = \lceil \sqrt{n}\, \rceil = k$. Then

$$\begin{aligned} g &= \sum_{0 \leq i < n} g_i x^i = \sum_{0 \leq j < l} (x^k)^j \sum_{0 \leq i < k} g_{i+kj} x^i \\ &= \sum_{0 \leq j < k} (x^k)^j G_j \text{ with } G_j = \sum_{0 \leq i < k} g_{i+kj} x^i. \end{aligned} \tag{7.3}$$

Brent & Kung (1978) used this grouping of g as the basic idea of their modular composition algorithm.

ALGORITHM 7.23. `modular composition`
Input: $f, g, h \in \mathbb{F}_q[x]$ *with* $n = \deg f$ *and* $\deg g, \deg h < n$.
Output: $g(h)$ *rem* f.

1. *Let* $k = \lceil \sqrt{n}\, \rceil$ *and* $G_i = \sum_{0 \leq j < k} g_{ik+j} x^j$ *for* $0 \leq i < k$.

2. *Set* $H^{(1)} = h$. *Compute* $H^{(i)} = hH^{(i-1)}$ *rem* f *for all* $2 \leq i \leq k$.

3. *Set* $P^{(1)} = H^{(k)}$. *Compute* $P^{(i)} = H^{(k)}P^{(i-1)}$ *rem* f *for all* $2 \leq i < k$.

4. *Compute* $G^{(i)} = G_i(h) = \sum_{0 \leq j < k} g_{j+ki} H^{(j)}$ *for* $0 \leq i < k$.

5. *Compute* $R = \sum_{0 \leq i < k} G^{(i)} P^{(i)}$ *rem* f.

6. *Return* R.

Theorem 7.24 (Brent & Kung 1978). *The algorithm* `modular composition` *works correctly. It computes* $g(h)$ *rem* f *using* $O(\sqrt{n}\mathsf{M}(n) + \sqrt{n}^{\,\omega+1})$ *operations in* $\mathbb{F}_q$.

Proof. Correctness can be seen directly with Equation (7.3) noting that $H^{(i)} = h^i$ rem f, and $P^{(i)} = h^{ki}$ rem f. The number of operations in $\mathbb{F}_q$ can be seen as follows: There are no operations in Step 1. Steps 2 can be done with $k-1$ multiplications modulo f. Step 3 can be done with $k-2$ multiplications modulo f. Step 5 uses k multiplications modulo f. Steps 2, 3 and 5 jointly need $O((3k-3)\mathsf{M}(n))$ operations in $\mathbb{F}_q$.

Step 4 can be computed using fast matrix multiplication because if for all $0 \leq i < k$ we have $H^{(i)} = \sum_{0 \leq j < n} H_j^{(i)} x^j$ then $G_i(h) = \sum_{0 \leq j_1 < k} g_{ki+j_1} H^{(j_1)} = \sum_{0 \leq j_2 < n} (\sum_{0 \leq j_1 < k} g_{j_1+ki} H_{j_2}^{(j_1)}) x^{j_2}$ rem f. Let $A = (a_{ij}) \in \mathbb{F}_q^{n \times k}$, $B = (b_{ij}) \in \mathbb{F}_q^{k \times k}$ with

$$
\begin{aligned}
a_{ij} &= g_{ik+j} \text{ for all } 0 \leq i < n, 0 \leq j < k \text{ and} \\
b_{ij} &= H_i^{(j)} \text{ for all } 0 \leq i, j < k.
\end{aligned}
$$

Then $(AB)_{ij} = \sum_{0 \leq s < k} g_{s+ik} H_s^{(j)}$.

This can be done with $\lceil \frac{n}{k} \rceil O(k^\omega)$ operations in $\mathbb{F}_q$ according to Lemma 7.18. Step 4 needs $k+1$ further calculations modulo f using $O(k\mathsf{M}(n))$ operations.

We therefore get a total number of $O(k\mathsf{M}(n) + \lceil \frac{n}{k} \rceil k^\omega)$ operations with $k = O(\sqrt{n})$ which completes the proof. □

Remark 7.25. *Modular composition can be done with*

1. $O(n^{\frac{5}{2}})$ *operations using classical arithmetic, i.e.* $\mathsf{M}(n) = O(n^2)$ *and* $\omega = 3$.

2. $O(\sqrt{n}(n^{\log_2 3} + \sqrt{n}^{\,\log_2 7})) = O(n^{\frac{1}{2} + \log_2 3})$ *operations using the algorithms of Karatsuba & Ofman and Strassen, i.e.* $\mathsf{M}(n) = O(n^{\log_2 3})$ *and* $\omega = \log_2 7$.

3. *$O(\sqrt{n}(n \log n \log\log n + \sqrt{n}^{\,\omega})) = O(n^{1.668})$ operations with $\omega < 2.376$ using the results of Schönhage & Strassen (1971), Schönhage (1977) and Cantor & Kaltofen (1991) for $\mathsf{M}(n)$ and Coppersmith & Winograd (1990) for ω, i.e. $\mathsf{M}(n) = O(n \log n \log\log n)$ and $\omega < 2.376$.*

Another model for counting operations. To compare algorithms more exactly we have to evaluate the constant hidden behind the 'O'-notation. But it is often difficult to compute the constant. For our purposes we count only *block operations.*

NOTATION 7.26. *We use the following block operations:*

$\mathsf{M}(n)$	*the number of operations in $\mathbb{F}_q$ to multiply two polynomials in $\mathbb{F}_q[x]$ of degree less than n.*
$\mathsf{S}(n)$	*the number of operations in $\mathbb{F}_q$* ∘ *to add n elements of $\mathbb{F}_q$ to n elements of $\mathbb{F}_q$ or* ∘ *to sum up n elements of $\mathbb{F}_q$ or* ∘ *to multiply n elements of $\mathbb{F}_q$ with one element of $\mathbb{F}_q$.*

We can estimate the cost for one multiplication of two polynomials modulo a fixed polynomial f of degree n with $3\mathsf{M}(n)+\mathsf{S}(n)$ ignoring the precomputation of the inverse of the reverse of f modulo x^n (cf. von zur Gathen & Gerhard 1995). We can assume $\mathsf{S}(kn) = k\mathsf{S}(n)$ and $k\mathsf{M}(n) \le \mathsf{M}(kn) \le k^2\mathsf{M}(n)$ for $k \in \mathbb{N}$. A cyclic shift of coefficients is assumed to be free.

COROLLARY 7.27. *Modular composition can be done using at most*

$$9\sqrt{n}\mathsf{M}(n)(1+o(1)) + 3\sqrt{n}\mathsf{S}(n)(1+o(1)) + \lceil\sqrt{n}\,\rceil O(\sqrt{n}^{\,\omega})$$

operations in $\mathbb{F}_q$. If classical matrix multiplication is used we have $\omega = 3$ and

$$\lceil\sqrt{n}\,\rceil O(n^{\frac{3}{2}}) = 2n\mathsf{S}(n)(1+o(1)).$$

PROOF. Let $k = \lceil\sqrt{n}\,\rceil$. We have $3k-3$ modular multiplications regarding Step 2, 3 and 5. Hence we have $3(k-1)(3\mathsf{M}(n)+\mathsf{S}(n))$ operations in $\mathbb{F}_q$. Step 5 uses $(k-1)\mathsf{S}(n)$ additional operations. The number of operations used in Step 4 can be seen from the proof of Theorem 7.24 for arbitrary ω. For classical matrix multiplication we have $\omega = 3$ using $2\mathsf{S}(n)$ operations for each entry of the resulting matrix. We therefore have $\lceil\frac{n}{k}\rceil \cdot 2k^2\mathsf{S}(k) \le \lceil\sqrt{n}\,\rceil \cdot 2\lceil\sqrt{n}\,\rceil^2\mathsf{S}(\lceil\sqrt{n}\,\rceil) \le 2\lceil\sqrt{n}\,\rceil^2\mathsf{S}(\lceil\sqrt{n}\,\rceil^2) = 2n\mathsf{S}(n)(1+o(1))$. □

7.4. Shoup's algorithm. We are now ready to give an algorithm for exponentiation using modular composition due to Shoup (1994). Shoup invented the algorithm for the special case $q = 2$. We give a generalization for all $q \in \mathbb{N}$ with q a prime power.

ALGORITHM 7.28. `exponentiation with composition`
Input: $f, b \in \mathbb{F}_q[x]$ *with* $\deg b < \deg f = n$, $e \in \mathbb{N}$ *with* $0 < e < q^n$ *and a parameter* $r \in \mathbb{N}$.
Output: $y = b^e$ *rem* f.

PART 1: Precomputation

1. *Let* $(e)_{q^r} = (e_{\lambda-1}, \ldots, e_0)$ *be the* q^r*-ary representation of* e *with* $\lambda = \lfloor \log_{q^r} e \rfloor + 1$ *and* $0 \leq e_i < q^r$ *for all* $0 \leq i < \lambda$.
2. *(Pre)Compute and store all values* b^{e_i} *rem* f *for* $0 \leq i < \lambda$.

PART 2: Horner's rule

3. *Compute* $h = x^{q^r}$ *rem* f.
4. *Let* $y = b^{e_{\lambda-1}}$ *rem* f. *For* $i = \lambda - 2$ *downto* 0 *do*
    5. *Compute* $y = y(h)$ *rem* f *by Algorithm* `modular composition` *according to Brent & Kung (1978).*
    6. *Compute* $y = yb^{e_i}$ *rem* f *using precomputed values.*
7. *Return* y.

LEMMA 7.29. *Algorithm* `exponentiation with composition` *works as specified.*

PROOF. We have $y(h) \equiv y^{q^r} \bmod f$ in Step 5 by modular composition according to Lemma 7.22. Then the algorithm above is just the q^r-ary method: Step 6 is Horner's rule (cf. Equation (3.3)) because after round i we have $y = b^{(\cdots(e_{\lambda-1}q^r + e_{\lambda-2})q^r + \cdots + e_{i+1})q^r + e_i}$. The $b^{e_i}, 0 \leq i < \lambda$ used in Step 6 are precomputed in Step 2. $\square$

LEMMA 7.30. *Algorithm* `exponentiation with composition` *can be done with* $O(\mathsf{M}(n)(\frac{n\sqrt{n}}{r} + r) + \frac{n\sqrt{n}}{r}\sqrt{n}^{\,\omega})$ *operations in* $\mathbb{F}_q$. *We have to store* $\frac{r}{\log_q r}(1 + o(1)) + \lfloor \frac{n}{r} \rfloor$ *elements of* $\mathbb{F}_{q^n}$

PROOF. The first part can be done using Algorithm `bgmw` (Algorithm 3.25) and Corollary 3.29. We have $e_i < q^r$ for all $0 \leq i < \lambda$. Denote the chooseable parameter in Algorithm `bgmw` with $r' \in \mathbb{N}$. Then we can compute $b^{e_0} \bmod f, \ldots, b^{e_{\lambda-1}} \bmod f$ with $Q = r'\lfloor \log_{q^{r'}} q^r \rfloor$ qth powers and $A \leq \lambda(q^{r'} + \lfloor \log_{q^{r'}} q^r \rfloor - 2)$ multiplications. Because raising to the qth power can be done with less than $2\lfloor \log_2 q \rfloor$ multiplications we have at most $A + 2\lfloor \log_2 q \rfloor Q$ multiplications for PART 1 (Steps 1–2).

If we choose $r' = \lfloor \log_q r - 2\log_q \log_q r \rfloor + 1$ according to Corollary 4.9 we get $A < 2\frac{r}{\log_q r}$ and $Q < r$. Hence we have at most $A + 2(\log_2 q)Q < 2(\frac{r}{\log_q r} + r\log_2 q)$ multiplications modulo f in the PART 1.

PART 2 (Steps 3–6) uses $r \log_2 q$ multiplications modulo f if we compute x^{q^r} rem f with Algorithm `binary` (Algorithm 3.13) in Step 4. Step 5 can be done with $\lambda - 1$ multiplications modulo f. In Step 6 we have $\lambda - 1$ modular compositions modulo f.

Using the result of Theorem 7.24 PART 2 of the algorithm can be done using $r(\log_2 q)O(\mathsf{M}(n)) + (\lambda - 1)(O(\mathsf{M}(n)) + O(\sqrt{n}\mathsf{M}(n) + \sqrt{n}^{1+\omega})) = O(\lambda\sqrt{n}(\mathsf{M}(n) + \sqrt{n}^{\ \omega}))$ operations in $\mathbb{F}_q$.

The whole algorithm can be done with

$$\begin{aligned} & O((\frac{r}{\log r} + r)\mathsf{M}(n)) + O(\lambda\sqrt{n}(\mathsf{M}(n) + \sqrt{n}^{\ \omega})) \\ = \ & O(\mathsf{M}(n)(\lambda\sqrt{n} + \frac{r}{\log r} + r) + (\sqrt{n}^{\ \omega})(\lambda\sqrt{n})) \end{aligned}$$

Because $\lambda = \lfloor \log_{q^r} e \rfloor + 1 = \lfloor \frac{1}{r}\log_q e \rfloor + 1 \leq \lfloor \frac{1}{r}\log_q q^n \rfloor + 1 = \lfloor \frac{n}{r} \rfloor + 1$ we have

$$\begin{aligned} & O(\mathsf{M}(n)(\lambda\sqrt{n} + \frac{r}{\log r} + r) + (\sqrt{n}^{\ \omega})(\lambda\sqrt{n})) \\ = \ & O(\mathsf{M}(n)(\frac{n\sqrt{n}}{r} + r + \frac{r}{\log r}) + (\sqrt{n}^{\ \omega})\frac{n\sqrt{n}}{r}) \\ = \ & O(\mathsf{M}(n)(\frac{n\sqrt{n}}{r} + r) + \frac{n^{\frac{\omega+3}{2}}}{r}). \ \square \end{aligned}$$

For PART 1 we have to store at most $\frac{r}{\log_q r}(1 + o(1))$ elements of $\mathbb{F}_{q^n}$ according to Corollary 4.9. The output of PART 1 has to be stored for PART 2. As can be seen in Step 2, the output of PART 1 contains $\lambda = \lfloor \log_{q^r} e \rfloor + 1 \leq \lfloor \frac{1}{r}\log_q q^n \rfloor = \lfloor \frac{n}{r} \rfloor$ elements of $\mathbb{F}_{q^r}$. $\square$

COROLLARY 7.31 (SHOUP 1994). *Let $b \in \mathbb{F}_{q^n}$ and $0 < e < q^n$. Then b^e can be evaluated with*

$$O(\mathsf{M}(n)\frac{n}{\log n} + \sqrt{n}^{\ \omega+1}\log n)$$

operations in $\mathbb{F}_q$. *Using fast polynomial arithmetic we have*

$$O(n^2 \log\log n)$$

operations in $\mathbb{F}_q$. *We have to store* $O(\frac{n}{\log n})$ *elements of* $\mathbb{F}_{q^n}$.

Proof. Let $f \in \mathbb{F}_q[x]$ with $\deg f = n$ be irreducible. Then $\mathbb{F}_{q^n} \cong \mathbb{F}_q[x]/(f)$ and we can use polynomial arithmetic.

Select $r = \lceil \frac{n}{\log n} \rceil$ as input for Algorithm `exponentiation with composition`. According to Lemma 7.30 b^e rem f can be computed using

$$\begin{aligned}
& O(\mathsf{M}(n)(\frac{n\sqrt{n}}{r} + r) + (\sqrt{n}^{\ \omega})\frac{n\sqrt{n}}{r}) \\
= \ & O(\mathsf{M}(n)(\frac{n\sqrt{n}}{\frac{n}{\log n}} + \frac{n}{\log n}) + (\sqrt{n}^{\ \omega})\frac{n\sqrt{n}}{\frac{n}{\log n}}) \\
= \ & O(\mathsf{M}(n)(\sqrt{n}\log n + \frac{n}{\log n}) + \sqrt{n}^{\ \omega+1}\log n) \\
= \ & O(\mathsf{M}(n)\frac{n}{\log n} + n^{\frac{\omega+1}{2}}\log n).
\end{aligned}$$

We can use fast multiplication with $\mathsf{M}(n) = O(n \log n \log\log n)$ according to Theorem 7.15. Fast matrix multiplication can be done with $\omega = \log_2 7$ using Theorem 7.19 which leads to the number of operations. The demand of storage can be easily seen from Lemma 7.30 because $\frac{r}{\log_q r} = O(\frac{\frac{n}{\log n}}{\log \frac{n}{\log n}}) = O(\frac{n}{(\log n)^2})$ and $O(\frac{n}{r}) = O(\log n) \in O(\frac{n}{(\log n)^2})$. □

Corollary 7.32. *Algorithm* `exponentiation with composition` *computes* $b^e \in \mathbb{F}_{q^n}$ *for* $b \in \mathbb{F}_{q^n}, e \in \mathbb{N}$ *using at most*

$$\begin{aligned}
& (9(\log_2 q)^2\frac{n}{\log_2 n} + \frac{9}{\log_2 q}\sqrt{n}\log_2 n)\mathsf{M}(n)(1+o(1)) \\
+ \ & (3(\log_2 q)^2\frac{n}{\log_2 n} + \frac{2}{\log_2 q}n\log_2 n)\mathsf{S}(n)(1+o(1))
\end{aligned}$$

operations in $\mathbb{F}_q$.

Proof. We only have to translate the proof of Lemma 7.30 using block operations and $r = \left\lceil \frac{n}{\log_q n} \right\rceil$.

Step 2 can be done with at most $\frac{r}{\log_2 r}(1 + o(1)) + 2r\log_2 q \leq 2r\log_2 q(1 +$

$o(1)) = 2\frac{n}{\log_q n}\log_2 q(1+o(1))$ modular multiplications. Step 3 uses $r\log_2 q = \frac{n}{\log_q n}\log_2 q(1+o(1))$ further modular multiplications.

Step 5 and 6 are repeated $\lambda - 1 = \lfloor \log_{q^r} e \rfloor + 1 - 1 \leq \lfloor \frac{1}{r}\log_q q^n \rfloor \leq \log_q n$ times. Step 6 uses one modular multiplication. According to Corollary 7.27 we have $9\sqrt{n}\mathsf{M}(n)(1+o(1)) + 2n\mathsf{S}(n)(1+o(1))$ operations in $\mathbb{F}_q$ for one modular composition with classical matrix multiplication.

Counting all operations of Steps 1–6 and estimating $3\mathsf{M}(n) + \mathsf{S}(n)$ for one modular multiplication, we have $(9(\log_2 q)^2 \frac{n}{\log_2 n} + \frac{9}{\log_2 q}\sqrt{n}\log_2 n)\mathsf{M}(n)(1+o(1))$ operations for multiplying and $(3(\log_2 q)^2 \frac{n}{\log_2 n} + \frac{2}{\log_2 q} n \log_2 n)\mathsf{S}(n)(1+o(1))$ further operations in $\mathbb{F}_q$. $\square$

Number of operations. We summarize the results of this section in the following theorem:

THEOREM 7.33. *Let $q, n \in \mathbb{N}$. Then the following holds using the polynomial representation for $\mathbb{F}_{q^n}$:*

1. *Addition of two elements can be done with n additions in $\mathbb{F}_q$.*
2. *Multiplication of two elements can be done with $O(n\log(n)\log\log(n))$ operations in $\mathbb{F}_q$.*
3. *Exponentiation of an element can be done with $O(n^2\log\log n)$ operations in $\mathbb{F}_q$ with an algorithm which needs to store $O(\frac{n}{(\log n)^2})$ elements of $\mathbb{F}_{q^n}$.*

PROOF.

1. Clear.
2. Theorem 7.15.
3. The number of operations in $\mathbb{F}_q$ is given by Corollary 7.31. The demand on storage can also be seen in Corollary 7.31.

8. Normal bases

8.1. Definition and basic arithmetic operations.

Definition and existence. We examine a normal basis representation in the following:

Recall Definition 6.9: A *normal basis* $\mathcal{N} = (\alpha_0, \ldots, \alpha_{n-1})$ of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ is a basis with

$$\alpha_0, \alpha_1 = \alpha_0^q, \ldots, \alpha_{n-1} = \alpha_0^{q^{n-1}}.$$

This is called a *normal basis representation* of $\mathbb{F}_{q^n}$.

FACT 8.1 (NORMAL BASIS THEOREM). *For any prime power q and $n \geq 1$, $\mathbb{F}_{q^n}$ has a normal basis over $\mathbb{F}_q$.*

PROOF. See e.g. the proof of Theorem 2.35 or Theorem 3.73 in Lidl & Niederreiter (1983). □

The elements of the normal basis determined by α are just the *conjugates* of α. For discussing the algebraic operations using a normal basis we recall the Frobenius automorphism:

DEFINITION 8.2. *Let $\mathbb{F}_{q^n}$ be a finite field. Then the map*

$$\begin{array}{rccc} \sigma\colon & \mathbb{F}_{q^n} & \to & \mathbb{F}_{q^n} \\ & \alpha & \mapsto & \alpha^q \end{array}$$

is called the Frobenius automorphism *of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$.*

REMARK 8.3. *The inverse map is given by $\alpha \mapsto \alpha^{q^{n-1}}$ and the following hold:*

1. *$\forall \alpha, \beta \in \mathbb{F}_{q^n}$: $\sigma(\alpha + \beta) = \sigma(\alpha) + \sigma(\beta)$,*

2. *$\forall \alpha, \beta \in \mathbb{F}_{q^n}$: $\sigma(\alpha\beta) = \sigma(\alpha)\sigma(\beta)$,*

3. *$\forall a \in \mathbb{F}_q$: $\sigma(a) = a$.*

Therefore σ is indeed an automorphism.

Addition. Let $\mathcal{N} = (\alpha_0, \ldots, \alpha_{n-1})$ be a normal basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ and $\beta, \gamma \in \mathbb{F}_{q^n}$ with $(\beta)_\mathcal{N} = (\sum_{0\le i<n} b_i\alpha_i)_\mathcal{N} = (b_0, \ldots, b_{n-1}), (\gamma)_\mathcal{N} = (\sum_{0\le i<n} c_i\alpha_i)_\mathcal{N} = (c_0, \ldots, c_{n-1})$. Then addition is component–wise as it is for any basis representation of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ and we have $(\beta+\gamma)_\mathcal{N} = (\sum_{0\le i<n}(b_i + c_i)\alpha_i)_\mathcal{N} = (b_0 + c_0, \ldots, b_{n-1} + c_{n-1})$. Therefore one addition in $\mathbb{F}_{q^n}$ needs n additions in $\mathbb{F}_q$.

Raising to the qth power. We know that the Frobenius automorphism is a linear operator on $\mathbb{F}_{q^n}$, as a $\mathbb{F}_q$-vector space. Therefore we have for an arbitrary $\beta = \sum_{0\le i<n} b_i\alpha_i \in \mathbb{F}_{q^n}$ with $(\beta)_\mathcal{N} = (b_0, \ldots, b_{n-1})$ that

$$\beta^q = \sigma(\beta) = \sigma(\sum_{0\le i<n} b_i\alpha_i) = \sum_{0\le i<n} b_i\sigma(\alpha_i) = \sum_{0\le i<n} b_i\alpha_{i+1}.$$

Thus $(\beta^q)_\mathcal{N} = (b_{n-1}, b_0, \ldots, b_{n-2})$. This is just a cyclic shift of the coordinates of β. It is therefore customary to neglect the cost of raising to the qth power (cf. Agnew *et al.* 1988, Stinson 1990, Jungnickel 1993, von zur Gathen 1991) because no arithmetic operation in $\mathbb{F}_q$ has to be done.

Multiplication. Unfortunately, multiplication is more difficult and expensive. To illustrate this (see e.g., Menezes *et al.* 1993, Chapter 5) let $(\delta)_\mathcal{N} = (d_0, \ldots, d_{n-1}) = (\beta\cdot\gamma)_\mathcal{N} \in \mathbb{F}_{q^n}$. Then, expressing the d_k's in terms of b_i's and c_j's, we have

$$\delta = \sum_{0\le k<n} d_k\alpha_k = (\sum_{0\le i<n} b_i\alpha_i)(\sum_{0\le j<n} c_j\alpha_j) = \sum_{0\le i,j<n} b_ic_j\alpha_i\alpha_j.$$

We define the *multiplication tensor* $T_k = (t_{ij}^{(k)})_{0\le i,j<n} \in \mathbb{F}_q^{n\times n}$ by

$$\alpha_i\alpha_j = \sum_{0\le k<n} t_{ij}^{(k)}\alpha_k. \tag{8.1}$$

Then we get

$$\sum_{0\le i,j<n} b_ic_jt_{ij}^{(k)} = d_k = \beta\cdot T_k\cdot\gamma^T \text{ for all } 0\le k<n. \tag{8.2}$$

This method works, in fact, for an arbitrary basis $\mathcal{B} = (\alpha_0, \ldots, \alpha_{n-1})$, and stores n matrices $T_0, \ldots, T_{n-1} \in \mathbb{F}_q^{n\times n}$, i.e. n^3 elements of $\mathbb{F}_q$. One multiplication in $\mathbb{F}_{q^n}$ then requires $2n\cdot n^2 = 2n^3$ multiplications in $\mathbb{F}_q$, plus $O(n^3)$ additions.

A substantial simplification is possible when $\mathcal{N} = (\alpha_0, \ldots, \alpha_{n-1})$ is a normal basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$. Raising both sides of Equation (8.1) to the q^{-l}th power we have

$$\begin{aligned}\sum_{0\le k<n} t^{(k)}_{i-l,j-l}\alpha_k &= \alpha_{i-l}\alpha_{j-l} = \sigma^{-l}(\alpha_i\alpha_j) = \sigma^{-l}(\sum_{0\le k<n} t^{(k)}_{ij}\alpha_k)\\ &= \sum_{0\le k<n} t^{(k)}_{ij}\sigma^{-l}(\alpha_k) = \sum_{0\le k<n} t^{(k)}_{ij}\alpha_{k-l}\\ &= \sum_{0\le k<n} t^{(k+l)}_{ij}\alpha_k\end{aligned}$$

$$\Rightarrow t^{(0)}_{i-l,j-l} = t^{(l)}_{ij} \text{ for all } l \in \{0, \ldots, n-1\}.$$

Consequently, we only have to store one matrix $T_0 \in \mathbb{F}_q^{n\times n}$ and can generate $T_k, 0 \le k < n$, by simple shifts. Writing $T_\mathcal{N} = (t_{ij})_{0\le i,j<n} \in \mathbb{F}_q^{n\times n}$, we have

$$t^{(k)}_{ij} = t^{(0)}_{i-k,j-k} = t^{(k-j)}_{i-k+k-j,j-k+k-j} = t^{(k-j)}_{i-j,0} = t_{i-j,k-j} \text{ for all } 0 \le i,j,k < n. \quad (8.3)$$

Massey & Omura's algorithm for multiplication. Using these results we have the following algorithm for multiplication of two elements in a normal basis representation of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$:

ALGORITHM 8.4. `Massey-Omura multiplier`
Input: $\beta, \gamma \in \mathbb{F}_{q^n}$ *with* $(\beta)_\mathcal{N} = (b_0, \ldots, b_{n-1}), (\gamma)_\mathcal{N} = (c_0, \ldots, c_{n-1})$ *and* $T_\mathcal{N} = (t_{ij})_{0\le i,j<n} \in \mathbb{F}_q^{n\times n}$ *for a normal basis* $\mathcal{N}$.
Output: $(\delta)_\mathcal{N} = (\beta \cdot \gamma)_\mathcal{N} = (d_0, \ldots, d_{n-1})$.

1. *For* $k = 0$ *to* $n-1$ *do*
    2. *Set* $d_k = 0$.
    3. *For all* $0 \le i, j < n$ *do*
        4. *Set* $0 \le z, s < n$ *with* $z \equiv (i-j) \bmod n$ *and* $s \equiv (k-j) \bmod n$.
        5. *If* $t_{zs} \ne 0$ *then compute* $d_k = d_k + b_i \cdot t_{zs} \cdot c_j$.
6. *Return* $(d_0, \ldots, d_{n-1})$.

LEMMA 8.5. *The algorithm* `Massey-Omura multiplier` *works as specified.*

PROOF. Correctness of the algorithm is clear because of Equation (8.2) and Equation (8.3). □

As it can be seen in Step 5 the number of non–zero entries in $T_{\mathcal{N}}$ determines the number of multiplications in $\mathbb{F}_q$ for one multiplication in the given normal basis representation of $\mathbb{F}_{q^n}$.

For $q = 2$ this algorithm can be directly used to construct hardware devices performing multiplication in $\mathbb{F}_{2^n}$ (see the example given in Jungnickel 1993, Chapter 3). This technique was first proposed by Massey & Omura (1981).

Density of $T_{\mathcal{N}}$.

NOTATION 8.6. *$T_{\mathcal{N}}$ is called the* multiplication table *of the normal basis $\mathcal{N}$, and the number of non–zero entries in $T_{\mathcal{N}}$ is the* density *$c_{\mathcal{N}}$ of $\mathcal{N}$.*

Ash *et al.* (1989) call $c_{\mathcal{N}}$ the 'complexity' of $\mathcal{N}$, but since this incorrectly suggests a connection to the usual notion of 'complexity', in this case of the complexity of multiplication in $\mathbb{F}_{q^n}$, we prefer the above terminology (cf. Schlink 1996b).

LEMMA 8.7. *Multiplying two elements of $\mathbb{F}_{q^n}$ given in a normal basis representation can be done with $2nc_{\mathcal{N}}$ multiplications in $\mathbb{F}_q$. Additionally, $c_{\mathcal{N}}$ elements of $\mathbb{F}_q$ have to be stored.*

PROOF. This is clear because of Algorithm `Massey-Omura multiplier`. □

Obviously $c_{\mathcal{N}} \leq n^2$. There are n^2 entries in $T_{\mathcal{N}}$ and every entry has q possible values with $q-1$ non–zero values. If we assume a binomial distribution for the number of non–zero entries in $T_{\mathcal{N}}$ we expect a density $E(c_{\mathcal{N}}) = \frac{q-1}{q}n^2$. But of course, the entries of $T_{\mathcal{N}}$ are not independent uniform random elements of $\mathbb{F}_q$. It depends on the chosen normal basis $\mathcal{N}$. For the topic of a randomly chosen normal basis see von zur Gathen & Giesbrecht (1990). A lower bound for $c_{\mathcal{N}}$ is given by the following theorem:

THEOREM 8.8. *If $\mathcal{N}$ is a normal basis for $\mathbb{F}_{q^n}$ then $c_{\mathcal{N}} \geq 2n-1$.*

PROOF. See Mullin *et al.* (1989), Theorem 2.1. □

DEFINITION 8.9. *A normal basis $\mathcal{N}$ with density $c_{\mathcal{N}} = 2n-1$ is called* optimal.

We therefore have a new task in our goal to speed up multiplication:

PROBLEM 8.10. *For which $q, n \in \mathbb{N}$ exists an optimal normal basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$?*

8.2. Normal bases generated by Gauß periods.

Gauß periods. To find 'good' normal bases, i.e. normal bases $\mathcal{N}$ for $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ with low density $c_{\mathcal{N}}$, we introduce Gauß periods.

DEFINITION 8.11. *Let $n, k \in \mathbb{N}$ such that $r = nk + 1$ is a prime. Let $\mathcal{K} < \mathbb{Z}_r^\times$ be the unique subgroup of $\mathbb{Z}_r^\times$ of order k, and let ζ be a primitive rth root of unity in $\mathbb{F}_{q^{nk}}$. Then*

$$\alpha = \sum_{a \in \mathcal{K}} \zeta^a$$

is called a Gauß period *of type (n, k) over $\mathbb{F}_q$.*

We have the following theorem based on Gauß periods:

THEOREM 8.12. *Let $r = nk + 1$ be a prime not dividing q, e the order of q modulo r, $\mathcal{K}$ be the unique subgroup of order k of the multiplicative group of $\mathbb{Z}_r$, and ζ be a primitive rth root of unity in $\mathbb{F}_{q^r}$. Then the Gauß period*

$$\alpha = \sum_{a \in \mathcal{K}} \zeta^a$$

generates a normal basis $\mathcal{N} = (\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}})$ of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ if and only if $\gcd(\frac{nk}{e}, n) = 1$.

PROOF. See Gao *et al.* (1995a), Theorem 2.1. Further proofs can be found in Menezes *et al.* (1993), Theorem 5.5 and Geiselmann (1994), Theorem 2.19. In the special case $q = 2$ a proof was given by Ash *et al.* (1989), Theorem 2.2. □

Determination of all optimal normal bases. This construction was first used by Ash *et al.* (1989) for $q = 2$. But only a reviewers comment cited in the paper mentioned the connection to Gauß periods. Mullin *et al.* (1989) showed that for $k \in \{1, 2\}$ one obtains a optimal normal basis.

COROLLARY 8.13 (MULLIN *et al.* 1989). *Suppose $n+1$ is a prime and $\mathbb{Z}_{n+1}^\times = \langle q \rangle$ with $q = p^t$, where p is a prime, $t \in \mathbb{N}$. Then $\mathcal{N} = \{\zeta \in \mathbb{F}_{q^n} : \zeta^{n+1} - 1 = 0$ and $\zeta \neq 1\}$ forms an optimal normal basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$.*

COROLLARY 8.14 (MULLIN *et al.* 1989). *Let $2n + 1$ be a prime and assume that either*

1. $\langle 2 \rangle = \mathbb{Z}_{2n+1}^{\times}$, *or*

2. $2n+1 \equiv 3 \bmod 4$ *and* $\langle 2 \rangle = \{a \in \mathbb{Z}_{2n+1} \colon \exists x \in \mathbb{Z}_{2n+1} : x^2 \equiv a \bmod 2n+1\}$.

Then there exists a primitive $(2n+1)$*st root of unity* $\zeta \in \mathbb{F}_{2^{2n}}$ *and* $\mathcal{N} = (\zeta + \zeta^{-1}, \ldots, \zeta^n + \zeta^{-n})$ *is an optimal normal basis of* $\mathbb{F}_{2^n}$ *over* $\mathbb{F}_2$.

Gao & Lenstra (1992) proved that these are the only optimal normal bases.

EXAMPLE 8.15. 1. *Let* $q = 2$ *and* $n = 24$. *Then* $\mathbb{F}_{2^{24}}$ *has no optimal normal basis over* $\mathbb{F}_2$ *because neither* $n + 1 = 25$ *nor* $2n + 1 = 49$ *are prime.*

2. *Let* $q = 2^{12}$ *and* $n = 2$. *We have* $2 \cdot 2 + 1 = 5$ *is a prime and* $\langle 2 \rangle = \{2, 4, 3, 1\} = \mathbb{Z}_5^{\times}$. *Therefore* $\mathbb{F}_{2^{24}}$ *has an optimal normal basis over* $\mathbb{F}_{2^{12}}$.

Therefore, there are finite fields $\mathbb{F}_{q^n}$ for which no optimal normal basis exists.

Density of normal bases generated by Gauß periods.

FACT 8.16. *Let* $\mathcal{N}$ *be a normal basis constructed according to Theorem 8.12. Then*

$$c_{\mathcal{N}} \le (n-1)k + n.$$

PROOF. See e.g. Geiselmann (1994) or Menezes *et al.* (1993). □

There are further results for special values of $q = p^t$ with p a prime, $t \in \mathbb{N}$.

FACT 8.17. *Let* $\mathcal{N}$ *be a normal basis constructed according to Theorem 8.12 with density* $c_{\mathcal{N}}$ *and let* $p = \mathrm{char}\mathbb{F}_q$.

1. *If* $p|k$, *then we have* $c_{\mathcal{N}} \le kn - 1$.

2. *If* $q = 2$, *then we have*

   - $nk - k + 1 - (k-2)^2 \le c_{\mathcal{N}} \le nk - k + 1$ *for* $k \equiv 0 \bmod 2$,
   - $n(k+1) - 2k + 1 - (k^2 - k + 2) \le c_{\mathcal{N}} \le n(k+1) - 2k + 1$ *for* $k \equiv 1 \bmod 2$.

PROOF.

1. See Menezes *et al.* (1993), Theorem 5.5.

2. For the upper bounds cf. Beth *et al.* (1991), Corollary 8. The lower bounds are given in Ash *et al.* (1989), Theorem 2.3.

The above theorem gives a new parameter k in the estimation of the density $c_{\mathcal{N}}$. To construct 'good' normal bases we therefore have to examine if there exists k small enough for given $q, n \in \mathbb{N}$.

We have to check for given q, n

1. the existence of a k satisfying the assumption of Theorem 8.12,
2. an upper bound on the smallest such k,
3. the density $c_{\mathcal{N}}$ of the corresponding normal basis $\mathcal{N}$.

Existence of k. The question whether there exists a k for given q, n and which upper bound can be given leads to the following definition (see Schlink 1996b):

DEFINITION 8.18. *A pair (n, k) is called* a Gauß pair *if and only if the Gauß period of type (n, k) is a normal element in $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$. Define*

$$\kappa'_q(n) = \begin{cases} \inf k & : \quad (n,k) \textit{ is a special Gauß pair, if such a } k \textit{ exists,} \\ \infty & : \quad \textit{if no such } k \textit{ exists.} \end{cases}$$

FACT 8.19. *Let $q = p^t$, p a prime, $t \in \mathbb{N}$ with the notations above. Then $\kappa'_q(n) < \infty$ if and only if the following conditions hold*

1. $\gcd(n, t) = 1$ *and*
2. (a) $2p \not| n$, *if* $p \equiv 1 \bmod 4$,

   (b) $4p \not| n$, *if* $p \equiv 2, 3 \bmod 4$.

PROOF. See Wassermann (1993), Satz: 3.3.4. □

In Ash *et al.* (1989) we can find the hint of a reviewer that for $q = 2$ we have $\kappa'_q(n) = \infty$ if $8|n$. This is caused in the fact that 2 is a quadratic residue modulo r if $8|(r-1)$.

Upper bounds on $\kappa'_q(n)$. If $\kappa'_q(n) < \infty$, i.e. there exists a k for given q, n satisfying the conditions of Theorem 8.12, we are interested in an upper bound on $\kappa'_q(n)$ to have bounds on $c_{\mathcal{N}}$ that only depend on q, n.

- Schlink (1996b) searched $\kappa'_q(n)$ experimentally and considered that $\kappa'_q(n)$ is, if finite, fairly small. Indeed the computational results lead to the conjecture that $\sqrt{n}$ is an upper bound for $\kappa'_q(n)$.
- Geiselmann (1994) did also empirical examinations for $n \leq 2 \cdot 10^4$ and $q \leq 32$. He states that $k \ll n$ can be assumed for cryptographically interesting n and q. In Beth *et al.* (1991) we find the assumption that $k \in O(n)$.
- Ash *et al.* (1989) listed k for some Mersenne primes $2^n - 1$. This confirms the conjecture that $k \in O(n)$.

Finally we point to an exhausted table in Gao *et al.* (1995b) not only on $\kappa'_q(n)$ but on Gauß periods of type (n, k).

A bound for $\kappa'_q(n)$ proven so far needs the *Extended Riemann Hypothesis* (ERH):

FACT 8.20. *Let $q = p^t$ a prime power and $n \in \mathbb{N}$. If n and q satisfy the conditions of Fact 8.19 the following holds, assuming the ERH:*

$$\kappa'_q(n) \in O(n^3 \log^2(np)).$$

PROOF. Cf. Bach & Shallit (1989). □

8.3. Construction of the multiplication table $T_{\mathcal{N}}$.

Basic ideas. If we want to multiply two elements $\beta, \gamma \in \mathbb{F}_{q^n}$ given in the normal basis representation according to Theorem 8.12 we can use Algorithm `Massey-Omura multiplier` (Algorithm 8.4). Then we have to compute the multiplication table $T_{\mathcal{N}} = (t_{ij})_{0 \leq i,j < n}$ of a normal basis $\mathcal{N}$ generated by Gauß periods.

One way to do so is to transfer $\alpha\alpha_i = \sum_{0 \leq j < n} t_{ij}\alpha_j$ into a special polynomial representation first and then to compute all $t_{ij}, 0 \leq i, j < n$.

But we can compute $T_{\mathcal{N}}$ also directly. The algorithm was first given by Wassermann (1993) and independently by Beth *et al.* (1991) and Geiselmann (1994). It is explicitly based upon the fact that the normal element $\alpha = \sum_{a \in \mathcal{K}} \zeta^a$ is generated by a Gauß period.

The constructive lemma. We have $\alpha\alpha_i = \sum_{0\leq j<n} t_{ij}\alpha^{q^j} = \sum_{0\leq j<n} t_{ij}\alpha_j$. To compute $T_{\mathcal{N}} = (t_{ij})_{0\leq i,j<n}$ we just need to know the normal basis representation of $\alpha\alpha_i$ for all $0 \leq i < n$.

NOTATION 8.21. *Let $\mathcal{K}_i = \{aq^i : a \in \mathcal{K}\} =: \mathcal{K}q^i$ for $0 \leq i < n$ where $\mathcal{K} < \mathbb{Z}_r^\times$ is the unique subgroup of order k. Let $0 \leq i_0 < n$ with $-1 \in \mathcal{K}_{i_0}$. If k is even then $i_0 = 0$, and if k is odd then $i_0 = \frac{n}{2}$. For $0 \leq i < n$ let*

$$\delta_{i,i_0} = \begin{cases} 0 & \text{if } i \neq i_0, \\ 1 & \text{if } i = i_0, \end{cases}$$

be the Kronecker symbol.

We have $\alpha_i = \alpha^{q^i} = (\sum_{a\in\mathcal{K}} \zeta^a)^{q^i} = \sum_{a\in\mathcal{K}} \zeta^{aq^i} = \sum_{b\in\mathcal{K}_i} \zeta^b$ for $0 \leq i < n$. We therefore have (cf. Gao *et al.* 1995a)

$$\begin{aligned} \alpha\alpha_i &= (\sum_{a\in\mathcal{K}} \zeta^a)(\sum_{b'\in\mathcal{K}_i} \zeta^{b'}) = (\sum_{a\in\mathcal{K}} \zeta^a)(\sum_{b\in\mathcal{K}} \zeta^{bq^i}) \\ &= \sum_{a,b\in\mathcal{K}} \zeta^{a+bq^i} = \sum_{a,b\in\mathcal{K}} \zeta^{a(1+bq^i)} \\ &= \sum_{b\in\mathcal{K}} \sum_{a\in\mathcal{K}} \zeta^{a(1+bq^i)} \end{aligned} \tag{8.4}$$

For each $b \in \mathcal{K}$, either $1 + bq^i \equiv 0 \bmod r$, or $1 + bq^i \in \mathcal{K}_j$ for a uniquely determined $j \in \{0, \ldots, n-1\}$. If $1 + bq^i \equiv 0 \bmod r$, then $i = i_0$ and $\sum_{a\in\mathcal{K}} \zeta^{a(1+bq^i)} = \sum_{a\in\mathcal{K}} \zeta^0 = \#\mathcal{K} \equiv k \bmod \Phi_r$. If $1 + bq^i \in \mathcal{K}_j$, then $i \neq i_0$ and $\sum_{a\in\mathcal{K}} \zeta^{a(1+bq^i)} \equiv \sum_{a\in\mathcal{K}} \zeta^{aq^j} \equiv \sum_{a\in\mathcal{K}_j} \zeta^a = \alpha_j \bmod \Phi_r$. Φ_r denotes the rth cyclotomic polynomial over $\mathbb{F}_q$ as defined in Definition 6.7.

We summarize this in the following lemma:

LEMMA 8.22. *Let $T_{\mathcal{N}} = (t_{ij})_{0\leq i,j<n}$ be the multiplication table corresponding to a normal basis $\mathcal{N}$ generated by Gauß period according to Theorem 8.12, and let $t'_{ij} = \#((1 + \mathcal{K}_i) \cap \mathcal{K}_j)$ for all $0 \leq i, j < n$ be the so–called* cyclotomic numbers. *Let i_0 and δ_{i,i_0} be as before. Then we have for $0 \leq i < n$ that*

$$t_{ij} = t'_{ij} - k\delta_{i,i_0}.$$

PROOF. We have $\alpha\alpha_i = \sum_{0\le j<n} t_{ij}\alpha_j$ and

$$\begin{aligned}
\alpha\alpha_i &= \sum_{b\in\mathcal{K}}\sum_{a\in\mathcal{K}} \zeta^{a(1+bq^i)} \\
&= \sum_{\substack{a,b\in\mathcal{K}\\ 1+bq^i\equiv 0 \bmod r}} \zeta^{a(1+bq^i)} + \sum_{\substack{a,b\in\mathcal{K}\\ 1+bq^i\in\mathcal{K}_j}} \zeta^{a(1+bq^i)} \\
&= k\delta_{i,i_0} + \sum_{a,b\in\mathcal{K},1+bq^i\in\mathcal{K}_j} \zeta^{a(1+bq^i)} \\
&= k\delta_{i,i_0} + \sum_{0\le j<n} t'_{ij}\alpha_j.
\end{aligned}$$

Since ζ is a primitive rth root of unity and $\frac{x^r-1}{x-1} = \sum_{0\le i\le nk} x^i$, we have $0 = \sum_{0\le i\le nk} \zeta^i$ and $-1 = \sum_{1\le i\le nk} \zeta^i = \sum_{0\le j<n} \alpha_j$. Therefore we have

$$k = \sum_{0\le j<n} (-k)\alpha_j, \tag{8.5}$$

and $\sum_{0\le j<n} t_{ij}\alpha_j = \sum_{0\le j<n}(t'_{ij} - k\delta_{i,i_0})\alpha_j$ which proves the claim. □

Our proof is based on Gao *et al.* (1995a), where a generalization of Lemma 8.22 can be found (cf. their Theorem 2.3).

An algorithm. We can now formulate an algorithm to compute the multiplication table $T_\mathcal{N}$ of a normal basis $\mathcal{N}$ generated by Gauß period (cf. Wassermann 1993, Algorithm 3.2.1).

ALGORITHM 8.23. `multiplication table`
Input: $q, k, n \in \mathbb{N}$ *such that the conditions of Theorem 8.12 are satisfied:* $r = nk+1$ *is a prime,* $r \nmid q$, $\gcd(\frac{nk}{\mathrm{ord}_{r(q)}}, n) = 1$.
Output: $T_\mathcal{N} \in \mathbb{F}_q^{n\times n}$, *the multiplication table for the normal basis* $\mathcal{N}$ *given by Theorem 8.12.*

1. *Let* $\mathcal{K} < \mathbb{Z}_r^\times$ *be the unique subgroup of order* k. *Compute an element* u *of order* k *in* $\mathbb{Z}_r^\times$ *and* $\mathcal{K} = \{u^i : 0 \le i < k\}$. *Compute* q^j *and* $\mathcal{K}_j = \mathcal{K}q^j$ *for all* $0 \le j < n$.
2. *Initialize* $T_\mathcal{N} = (t_{ij})_{0\le i,j<n} = 0$.
3. *If* k *is even then set* $i_0 = 0$ *else set* $i_0 = \frac{n}{2}$.

4. *For $i = 0$ to $n-1$ do*

   5. *If $i = i_0$ then set $t_{ij} = t_{ij} - k$ for all $0 \le j < n$.*
   6. *For all $b \in \mathcal{K}$ do*

      7. *If $1+bq^i \not\equiv 0 \bmod r$ then let $j \in \{0,\ldots,n-1\}$ with $1+bq^i \in \mathcal{K}_j$, and set $t_{ij} = t_{ij} + 1$.*

8. *Return $T_{\mathcal{N}}$.*

LEMMA 8.24. *Algorithm* `multiplication table` *computes $T_{\mathcal{N}}$ correctly.*

PROOF. This is clear because of Lemma 8.22. □

LEMMA 8.25. *Algorithm* `multiplication table` *computes $T_{\mathcal{N}}$ with at most $3nk + n - k - 2$ multiplications and nk additions in $\mathbb{Z}_r$ and $n(k+1)$ additions in $\mathbb{F}_q$. At most $n(k+1) - 2$ elements of $\mathbb{Z}_r^\times$ have to be stored.*

PROOF. In Step 1 an element $u \in \mathbb{Z}_r^\times$ of order k has to be found to compute $\mathcal{K} = \mathcal{K}_0 = \{u^i : 0 \le i < k\}$. This can be done with $\le nk$ multiplications in $\mathbb{Z}_r^\times$. $(n-1)k$ further multiplications are needed to compute $\mathcal{K}_j$ for $1 \le j < n$. Finally there are $n-2$ multiplications to compute $q^j, 2 \le j < n$. Therefore Step 1 needs $\le 2nk + n - k - 2$ multiplications in $\mathbb{Z}_r^\times$. In Steps 2+3 there are no operations to count. In Step 5 there are n additions in $\mathbb{F}_q$ because i_0 is uniquely determined in $\{0,\ldots,n-1\}$. In Steps 6+7 we have k multiplications and k additions in $\mathbb{Z}_r$ and k additions in $\mathbb{F}_q$. The total number of operations in Steps 4–7 is therefore nk multiplications and nk additions in $\mathbb{Z}_r$ and $nk+n$ additions in $\mathbb{F}_q$.

The demand on storage can be seen as follows: We have to store $\mathcal{K}_j \subset \mathbb{Z}_r^\times, 0 \le j < n$ with k elements each and $q^2, \ldots, q^{n-1} \in \mathbb{Z}_r^\times$. Therefore the algorithm has to store $nk + n - 2 = n(k+1) - 2$ elements of $\mathbb{Z}_r^\times$. □

Number of operations. We summarize the results of this section in the following theorem:

THEOREM 8.26. *Let $q, n, k \in \mathbb{N}$ satisfy the conditions of Theorem 8.12. Then using the normal basis representation for $\mathbb{F}_{q^n}$ the following hold:*

1. *The addition of two elements in $\mathbb{F}_{q^n}$ can be done with n additions in $\mathbb{F}_q$.*
2. *The multiplication of two elements in $\mathbb{F}_{q^n}$ can be done with $O(n^2k)$ operations in $\mathbb{F}_q$.*

3. *The exponentiation of an element in $\mathbb{F}_{q^n}^{\times}$ can be done with $O(\frac{n^3 k}{\log n})$ operations in $\mathbb{F}_q$. $O(\frac{n}{\log n})$ elements of $\mathbb{F}_{q^n}$ and $c_{\mathcal{N}}$ elements of $\mathbb{F}_q$ have to be stored.*

PROOF.

1. Clear.
2. Lemma 8.7 in connection with Result 8.16.
3. According to Algorithm `bgmw` (Algorithm 3.25) we need $\frac{n}{\log_q n}(1 + o(1))$ multiplications in $\mathbb{F}_{q^n}$ (Corollary 4.9) because raising to the qth power is just a cyclic shift. This method stores $\frac{n}{\log_q n}(1+o(1))$ elements of $\mathbb{F}_{q^n}$. The number of elements of $\mathbb{F}_q$ to store is due to the number $c_{\mathcal{N}}$ of non–zero entries in the multiplication table $T_{\mathcal{N}}$.

COROLLARY 8.27. *The exponentiation of an element in $\mathbb{F}_{q^n}^{\times}$ can be done with $2\log_2 q \frac{nc_{\mathcal{N}}+n^2}{\log_2 n}\mathsf{S}(n)(1+o(1))$ operations in $\mathbb{F}_q$. $\mathsf{S}(n)$ is used as given in Notation 7.26.*

PROOF. We examine a modified version of Algorithm `Massey-Omura multiplier` (Algorithm 8.4) according to the idea of Jungnickel (1993), Chapter 3: If we use T_0 instead of $T_{\mathcal{N}}$ (T_0 can be computed given $T_{\mathcal{N}}$ without any operations in $\mathbb{F}_q$, cf. Equation 8.3 and Beth *et al.* 1991), then we can compute $d_k = \beta T_k({}^t\gamma)$ by computing $\beta^{q^k} T_0({}^t\gamma)^{q^k}$. Set ${}^tC := (\gamma^{q^0}, \gamma^{q^1}, \ldots, \gamma^{q^{n-1}})$. If $t_{ij}^{(0)} \neq 0$ then one multiplication of row j in C with the scalar $t_{ij}^{(0)} \in \mathbb{F}_q$ and one addition with the previous result according to row j has to be done. Hence we can compute ${}^t\gamma_k' = T_0({}^t\gamma)^{q^k}$ for all $0 \leq k < n$ with $2c_{\mathcal{N}}\mathsf{S}(n)$ operations in $\mathbb{F}_q$. But then we can compute $d_k = \beta^{q^k}({}^t\gamma_k')$ with $2\mathsf{S}(n)$ further operations for $k \in \{0, \ldots, n-1\}$. So we have a total number of $2(c_{\mathcal{N}} + n)\mathsf{S}(n)$ operations for Algorithm `Massey-Omura multiplier`. According to Corollary 4.9 we can compute an exponentiation using Algorithm `bgmw` (Algorithm 3.25) with at most $\frac{n}{\log_q n}(1 + o(1))$ multiplications which completes the proof. $\square$

9. Using fast multiplication within normal basis representation

9.1. The basic idea. Gao *et al.* (1995a) suggest a way to connect fast multiplication (using polynomial basis representation) and free raising to the qth power in $\mathbb{F}_{q^n}$ (using normal basis representation). Their idea is based on normal bases generated by Gauß periods of type (n, k) according to Theorem 8.12 and the fact that $\xi = x \bmod \Phi_r \in \mathbb{F}_q[x]/(\Phi_r)$ is a primitive rth root of unity. Φ_r is the rth cyclotomic polynomial as defined in Definition 6.7.

9.2. The residue class ring $\mathbb{F}_q[x]/(\Phi_r)$.

The rth cyclotomic polynomial over $\mathbb{F}_q$. We now introduce a special residue class ring in $\mathbb{F}_q[x]$ for $r \in \mathbb{N}$. We regard cyclotomic polynomials Φ_r over $\mathbb{F}_q$. An introduction on cyclotomic polynomials for arbitrary $r \in \mathbb{N}$ is given in Lidl & Niederreiter (1983), Chapter 2.4. We concentrate on the special case when $r = nk + 1$ is a prime for $n, k \in \mathbb{N}$. Then 1 and r are the only divisors of r and therefore the following hold:

LEMMA 9.1. *Let $r = nk + 1$ be a prime and $\mathbb{F}_q$ be a field with $\gcd(q, r) = 1$. Then:*

1. $\frac{x^r-1}{x-1} = \Phi_r(x) = \sum_{0 \le i \le nk} x^i$ *is monic of degree nk,*
2. $\Phi_r(x)$ *is irreducible in $\mathbb{F}_q[x]$ if and only if $\mathrm{ord}_r(q) = nk$.*

PROOF.

1. $\gcd(q, r) = 1$ implies that char $\mathbb{F}_q \not| r$. According to Lidl & Niederreiter (1983), Theorem 2.45, we have $x^r - 1 = \prod_{d|r} \Phi_d(x) = \Phi_1(x)\Phi_r(x) = (x-1)\Phi_r(x)$.
2. This follows directly from Lidl & Niederreiter (1983), Theorem 2.47.

Two polynomial bases for $\mathbb{F}_q[x]/(\Phi_r)$. The following is due to Gao *et al.* (1995a):

REMARK 9.2. *Let $\mathcal{R} = \mathbb{F}_q[x]/(\Phi_r)$ and $\xi = x \bmod \Phi_r \in \mathcal{R}$. Then $\mathcal{R}$ has two bases $\mathcal{B}_1 = (1, \xi, \dots, \xi^{nk-1})$ and $\mathcal{B}_2 = (\xi, \dots, \xi^{nk})$ over $\mathbb{F}_q$.*

PROOF. We have $\mathcal{R} = \langle 1, \xi, \ldots, \xi^{nk-1} \rangle$ because every element of $\mathcal{R}$ can be represented by a polynomial of degree at most $nk-1$. But $\dim_{\mathbb{F}_q}\mathcal{R} = \deg \Phi_r = nk$ and hence $\mathcal{B}_1 = (1, \xi, \ldots, \xi^{nk-1})$ is a basis of $\mathcal{R}$ over $\mathbb{F}_q$.

We have $x^{nk} = \Phi_r(x) \cdot 1 - \sum_{0 \le i < nk} x^i$ and $\xi^{nk} \equiv -\sum_{0 \le i < nk} \xi^i \bmod \Phi_r$ so that $1 = \xi^0 \equiv -\xi^{nk} - \sum_{1 \le i < nk} \xi^i \bmod \Phi_r$ and every element of $\mathcal{B}_1$ is a linear combination of the elements of $\mathcal{B}_2 = \{\xi, \ldots, \xi^{nk}\}$. Since their number is the same, also $\mathcal{B}_2$ is an $\mathbb{F}_q$-basis. □

It is easy to go from one basis to another:

$$\begin{aligned} \sum_{1 \le i \le nk} a_i \xi^i &= -a_{nk}\xi^0 + \sum_{1 \le i < nk} (a_i - a_{nk})\xi^i \text{ and} \\ \sum_{0 \le i < nk} a_i \xi^i &= \sum_{1 \le i < nk} (a_i - a_0)\xi^i - a_0 \xi^{nk}. \end{aligned}$$

We therefore have to do $nk - 1$ subtractions in $\mathbb{F}_q$ to go from $\mathcal{B}_1$ to $\mathcal{B}_2$ and vice versa.

9.3. A transformation. We can choose $\xi = x \bmod \Phi_r$ as a primitive rth root of unity with Φ_r the rth cyclotomic polynomial. Let q, n, k satisfy the conditions of Theorem 8.12. Then

$$\alpha = \sum_{a \in \mathcal{K}} \xi^a$$

is a normal element in $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ according to Theorem 8.12 and $\mathcal{N} = (\alpha_0, \ldots, \alpha_{n-1})$ is a normal basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$.

We can now change between this normal basis representation of $\mathbb{F}_{q^n}$ and a polynomial representation of $\mathcal{R}$ by defining a map φ from $\mathbb{F}_{q^n}$ with basis $\mathcal{N}$ to $\mathcal{R}$ with basis $\mathcal{B}_2$. If necessary we can easily change between the two polynomial bases $\mathcal{B}_1$ and $\mathcal{B}_2$ of $\mathcal{R}$.

Define

$$\begin{array}{rrcl} \varphi\colon & \mathbb{F}_{q^n} & \to & \mathcal{R} \\ & \sum_{0 \le i < n} b_i \alpha_i & \mapsto & \sum_{1 \le j \le nk} b'_j \xi^j \text{ with } b'_j = b_i \text{ if } j \in \mathcal{K}_i. \end{array}$$

φ is well–defined since $\mathbb{Z}_r^\times = \dot{\bigcup}_{0 \le j < n} \mathcal{K}_j$.

LEMMA 9.3. *1. φ is an injective ring homomorphism which fixes $\mathbb{F}_q$.*

2. $\varphi(\beta)$ is invertible in $\mathcal{R}$ for all $\beta \in \mathbb{F}_{q^n} \setminus \{0\}$.

PROOF.

1. ◦ φ is injective: Let $\beta = \sum_{0\leq i<n} b_i\alpha_i, \gamma = \sum_{0\leq i<n} c_i\alpha_i \in \mathbb{F}_{q^n}$ with $\beta \neq \gamma$. Then there exist $i \in \{0,\ldots,n-1\}$ with $b_i \neq c_i$ and thus $b'_j \neq c'_j$ for $j \in \mathcal{K}_i$, i.e. $\varphi(\beta) \neq \varphi(\gamma)$. Because $\varphi(0) = \varphi(\sum_{0\leq i<n} 0 \cdot \alpha_i) = \sum_{1\leq j\leq nk} 0 \cdot \xi^j = 0$ and φ injective we have $\ker \varphi = \{0\}$.

   ◦ We have $\varphi(\alpha_i) = \varphi(\sum_{a\in\mathcal{K}_i} \xi^a) = \sum_{a\in\mathcal{K}} \xi^{aq^i}$ and $1 = -\sum_{0\leq i<n} \alpha_i = -\sum_{1\leq j\leq nk} \xi^j$. Hence we have for $b \in \mathbb{F}_q$: $\varphi(b) = \varphi(b \cdot 1) = \varphi(\sum_{0\leq i<n} -b\alpha_i) = \sum_{1\leq j\leq nk} -b\xi^j = b \cdot 1 = b$.

   ◦ Obviously φ is additive. Because of

$$\begin{aligned}
\varphi(\alpha_i\alpha_j) &= \varphi(\alpha^{q^i}\alpha^{q^j}) \\
&= \varphi(\alpha^{q^i}(\alpha^{q^{j-i}})^{q^i}) = \varphi((\alpha\alpha_{j-i})^{q^i}) \\
&\overset{(8.4)}{=} \sum_{a,b\in\mathcal{K}} \xi^{a(1+bq^{j-i})q^i} = (\sum_{a\in\mathcal{K}} \xi^{aq^i})(\sum_{b\in\mathcal{K}} \xi^{bq^j}) \\
&= \varphi(\alpha_i)\varphi(\alpha_j)
\end{aligned}$$

   φ is also multiplicative and hence a ring homomorphism.

2. φ is not surjective for $k > 1$: $\#\mathbb{F}_{q^n} = q^n$ but $\#\mathcal{R} = \#\mathbb{F}_q[x]/(\Phi_r) = q^{nk}$ with $\deg \Phi_r = nk$. But we have (see Gao *et al.* 1995a)

$$\mathcal{R}' = \{ \sum_{1\leq j\leq nk} b'_j\xi^j \in \mathcal{R}: b'_j \in \mathbb{F}_q \text{ and } b'_{j'} = b'_j \text{ for } j, j' \in \mathcal{K}_i, 0 \leq i < n\}$$

   a subring of $\mathcal{R}$ because $\mathcal{R}' = \operatorname{im} \varphi$. Thus $\varphi(\beta)$ is invertible in $\mathcal{R}$ for all $\beta \in \mathbb{F}_{q^n} \setminus \{0\}$.

9.4. Fast multiplication based on Gauß periods. We are now ready to present the algorithm of Gao *et al.* (1995a) to use fast multiplication within a normal basis representation. The normal basis $\mathcal{N}$ is generated by a Gauß period according to Theorem 8.12.

ALGORITHM 9.4. `fast normal basis multiplication`
Input: $q, n, k \in \mathbb{N}$ *which satisfy the conditions of Theorem 8.12. Let* $\mathcal{N} = (\alpha_0, \ldots, \alpha_{n-1})$ *be the normal basis of* $\mathbb{F}_{q^n}$ *over* $\mathbb{F}_q$ *generated by a Gauß period according to Theorem 8.12 and* $(\beta)_\mathcal{N} = (\sum_{0\leq i<n} b_i\alpha_i)_\mathcal{N} = (b_0, \ldots, b_{n-1}), (\gamma)_\mathcal{N} = (\sum_{0\leq i<n} c_i\alpha_i)_\mathcal{N} = (c_0, \ldots, c_{n-1}) \in \mathbb{F}_{q^n}$.
Output: $(\delta)_\mathcal{N} = (\beta\gamma)_\mathcal{N} = (d_0, \ldots, d_{n-1}) \in \mathbb{F}_{q^n}$.

1. *[Transformation from $\mathbb{F}_{q^n}$ into $\mathcal{R}'$:] Compute $\beta', \gamma' \in \mathcal{R}'$ with $\beta' = \varphi(\beta) = \sum_{1\leq j\leq nk} b'_j\xi^j, \gamma' = \varphi(\gamma) = \sum_{1\leq j\leq nk} c'_j\xi^j$ with $b'_j = b_i, c'_j = c_i$ if $j \in \mathcal{K}_i$.*

2. *[Fast polynomial multiplication:] Compute $\delta_1 = \beta' \cdot \gamma' = \sum_{1\leq j\leq 2nk} d_j^{(1)}\xi^j$.*

3. *[Reduction modulo $x^r - 1$:] Set $\delta_2 = \sum_{0\leq i\leq nk} d_j^{(2)}\xi^j \equiv \delta_1 \bmod (x^{nk+1} - 1)$ by computing $d_j^{(2)} = d_j^{(1)} + d_{j+nk+1}^{(1)}$ for all $0 \leq j < nk$.*

4. *[Transformation into $\mathcal{R}'$:] Compute $\delta' = \sum_{1\leq j\leq nk} d'_j\xi^j \in \mathcal{R}'$ with $d'_j = d_j^{(2)} - d_0^{(2)}$ for all $1 \leq j \leq nk$.*

5. *[Transformation from $\mathcal{R}'$ into $\mathbb{F}_{q^n}$:] Set $d_i = d'_j$ for $j \in \mathcal{K}_i, 0 \leq i < n$. Compute $\delta = \sum_{0\leq i<n} d_i\alpha_i = (d_0, \ldots, d_{n-1})_{\mathcal{N}}$.*

LEMMA 9.5. *Algorithm* `fast normal basis multiplication` *works as specified. It uses $\mathsf{M}(nk)$ multiplications and at most $2nk$ additions in $\mathbb{F}_q$ to multiply two arbitrary elements of $\mathbb{F}_{q^n}$ given in the normal basis representation corresponding to a Gauß period of type (n, k). We have to store $2nk$ elements of $\mathbb{F}_q$.*

PROOF. The correctness of the algorithm is clear from the arguments given above.

Step 1+5 can be done without any operations in $\mathbb{F}_q$. Step 2 needs $\mathsf{M}(nk)$ multiplications in $\mathbb{F}_q$ because $\varphi(\beta), \varphi(\gamma)$ are polynomials of degree at most nk. Step 3 can be done with nk additions in $\mathbb{F}_q$. Step 4 also needs nk additions in $\mathbb{F}_q$.

The algorithm needs to compute $\varphi(\beta)$ and $\varphi(\gamma)$. This uses a demand on storage of at most $2 \cdot nk$ elements of $\mathbb{F}_q$. □

COROLLARY 9.6. *Algorithm* `fast normal basis multiplication` *computes the product of two elements in $\mathbb{F}_{q^n}$ given in normal basis representation using $\mathsf{M}(kn) + 2\mathsf{S}(kn) = \mathsf{M}(kn) + 2k\mathsf{S}(n)$ operations in $\mathbb{F}_q$.*

Number of operations. We summarize the results of this section in the following Theorem (cf. Gao *et al.* 1995a, Theorem 3.1):

THEOREM 9.7. *Let $q, n, k \in \mathbb{N}$ satisfy the conditions of Theorem 8.12. Then the following holds for the normal basis representation of elements of $\mathbb{F}_{q^n}$:*

1. *Addition of two elements can be done with n additions in $\mathbb{F}_q$.*

2. *Multiplication of two elements can be done with* $O(nk\log(nk)\log\log(nk))$ *operations in* $\mathbb{F}_q$.

3. *Exponentiation of an element uses at most* $O(\frac{n^2k}{\log n}\log(nk)\log\log(nk))$ *operations in* $\mathbb{F}_q$. *The algorithm needs to store* $O(\frac{n}{\log n})$ *elements of* $\mathbb{F}_{q^n}$.

PROOF.

1. Clear.

2. Lemma 9.5 in connection with Theorem 7.15.

3. According to Corollary 4.9 we need $\frac{n}{\log_q n}(1+o(1))$ multiplications in $\mathbb{F}_{q^n}$ for one exponentiation using Algorithm `bgmw` (Algorithm 3.25) because raising to the qth power is just a cyclic shift of coefficients. Algorithm `bgmw` stores at most $\frac{n}{\log_q n}(1+o(1))$ elements of $\mathbb{F}_{q^n}$.

COROLLARY 9.8. *Exponentiation of an element of* $\mathbb{F}_{q^n}$ *can be done with*

$$\log_2 q\frac{n}{\log_2 n}\mathsf{M}(kn)(1+o(1))+2k\log_2 q\frac{n}{\log_2 n}\mathsf{S}(n)(1+o(1))$$

operations in $\mathbb{F}_q$.

PROOF. Using Algorithm `bgmw` (Algorithm 3.25) we have to do at most $\frac{n}{\log_q n}(1+o(1))$ multiplications according to Corollary 4.9. Using Algorithm `fast normal basis multiplication` (Algorithm 9.4) we have $\mathsf{M}(kn)+2k\mathsf{S}(n)$ operations in $\mathbb{F}_q$ per multiplication (see Corollary 9.6) which completes the proof. □

9.5. A summarizing table. Before we introduce the results of our implementations we give a theoretical comparison of the three exponentiation algorithms for $\mathbb{F}_{q^n}$ we have analyzed. We restrict to the case $q=2$ and $k\le 2$, i.e. the following Table 4 is only true for field extensions over $\mathbb{F}_2$ for which a optimal normal basis exists.

We use the following short names:

NOTATION 9.9. ◦ `onb`: *Algorithm* `bgmw` *(Algorithm 3.25) in connection with normal basis representation for* $\mathbb{F}_{2^n}$ *and Algorithm* `Massey-Omura multiplier` *(Algorithm 8.4) for multiplication.*

Algorithm	onb	ggp	shoup
O-notation $(\omega < 3)$	$O(\frac{n^3}{\log n})$	$O(n^2 \log\log n)$	$O(n^2 \log\log n)$
block operations $c_{\mathsf{M}} \cdot \mathsf{M}(n)(1+o(1))$ $+$ $c_{\mathsf{S}} \cdot \mathsf{S}(n)(1+o(1))$ $(\omega = 3)$	$c_{\mathsf{M}} = 0$ $c_{\mathsf{S}} = 6\frac{n^2}{\log_2 n}$	$c_{\mathsf{M}} \leq k^2 \frac{n}{\log_2 n}$ $c_{\mathsf{S}} = 2k\frac{n}{\log_2 n}$	$c_{\mathsf{M}} = 9\frac{n}{\log_2 n}$ $c_{\mathsf{S}} = 2n\log_2 n$
storage (only $\mathbb{F}_{2^n}$)	$O(\frac{n}{\log n})$	$O(\frac{n}{\log n})$	$O(\frac{n}{(\log n)^2})$

Table 4: Theoretical comparision between three exponentiation algorithms over $\mathbb{F}_{2^n}$ for $n, k \in \mathbb{N}$ and (n, k) is a Gauß pair.

- `shoup`: *Short for Algorithm* `exponentiation with composition` *(Algorithm 7.28) for polynomial representation of* $\mathbb{F}_{2^n}$.
- `ggp`: *Algorithm* `bgmw` *(Algorithm 3.25) in connection with normal basis representation for* $\mathbb{F}_{2^n}$ *and Algorithm* `fast normal basis multiplication` *(Algorithm 9.4) for multiplication.*

10. Practical results for addition chain heuristics

10.1. The experiment.

Numerical results in the literature. Brickell *et al.* (1993), de Rooij (1995) and Bocharova & Kudryashov (1995) give some numerical results for original addition chains. They examine the number of steps and the number of elements to store for some of the algorithms `binary` (Algorithm 3.13), `brauer` (Algorithm 3.17), `bgmw` (Algorithm 3.25) and `bocharova` (Algorithm 3.39). A summary on their results is given in Table 5. They concentrate on average and worst case values for inputs of length 160 bit and 512 bit. We do not know of more detailed results for this four algorithms. We found no numerical results for Algorithm `yacobi` (Algorithm 3.31) in the literature. We now compare all five algorithms given in the literature and the new Algorithm `lookahead` (Algorithm 3.44) in detail.

input λ	algorithm	reference	param. r	#steps aver	#steps max	#non-doub. aver	#non-doub. max	storage aver	storage max
160	`binary`	Brickell *et al.* (1993)		237	318				
	`bgmw`	Brickell *et al.* (1993)	$\log_2 12$			50.25	54	45	45
			$\log_2 19$			43.00	45	76	76
		de Rooij (1995)	?			50		45	47
	`brauer`	de Rooij (1995)		197				9	9
512	`binary`	Brickell *et al.* (1993)		765	1022				
	bgmw	Brickell *et al.* (1993)	$\log_2 26$			127.81	132	109	109
			$\log_2 45$			111.91	114	188	188
		de Rooij (1995)	?			128		109	111
	`brauer`	de Rooij (1995)		611				17	17
		Bocharova & Kudryashov (1995)				111		62	62
	`bocharova`	Bocharova & Kudryashov (1995)				102		16	16

Description: '?' no parameter is specified

Table 5: Some numerical results on addition chain algorithms in the literature

Input parameters. Due to the numerical results that can be found in the literature we concentrate on original addition chains (i.e. $q = 2$) and examine inputs of length $\lambda = 160$ bits, $\lambda = 512$ bits and $\lambda = 1024$ bits. We also distinguish between different Hamming weights ν. For each length λ we consider inputs with low Hamming weight ($\nu \approx \frac{\lambda}{4}$), medium Hamming weight ($\nu \approx \frac{\lambda}{2}$) and high Hamming weight ($\nu \approx \frac{3\lambda}{4}$). We run all 9 combinations with 1000 randomly chosen inputs m. Explicit formulas for the chooseable parameter r in the algorithms `brauer`, `bgmw` and `bocharova` have already been given. We use these formulas with λ instead of $\log_2 m$. The parameter r in `bgmw` is computed according to the formula $r = \lfloor \log_2 \lambda - 2\log_2 \log_2 \lambda \rfloor + 2$. The additional constant 2 is chosen to obtain better results for the concrete length. This doesn't influence the asymptotical behaviour and gives also comparable results to the numerical results of Brickell *et al.* (1993).

number of bits	$\lambda = 160$	$\lambda = 512$	$\lambda = 1024$
Hamming weight	$\nu \approx \frac{1}{4}\lambda$	$\nu \approx \frac{1}{2}\lambda$	$\nu \approx \frac{3}{4}\lambda$
number of computations	1000 randomly chosen bitstrings for any combination		
parameter r	according to the theoretical results		

Table 6: Input parameter for addition chain algorithms

Output parameters. For each input m we get the total number of steps, the number of doublings and the number of further steps (called non–doublings). We also count the number of elements that have to be stored during the computation (without intermediate results). The results are given in Table 7 (for $\lambda = 160$), Table 8 (for $\lambda = 512$) and Table 9 (for $\lambda = 1024$). We analyze these results by first comparing the classical algorithms: `binary`, `brauer` and `bgmw`. Then we give a survey on the algorithms using data compression techniques: `yacobi`, `bocharova` and `lookahead`. Finally, we compare both groups of algorithms.

10.2. The classical algorithms. The algorithm `binary` computes addition chains that are only acceptable for low hamming weight ($\nu \approx \frac{1}{4}\lambda$). These addition chains contain nearly the same number of doublings as the addition chains produced by `brauer` and `bgmw`. But the number of star steps is 2–4 times as high as using `brauer` or `bgmw` for high Hamming weight ($\nu \approx \frac{3}{4}\lambda$). The advantage of `binary` is that we have to store only the input.

`brauer` and `bgmw` both need more storage — to reduce the number of non–doubling–steps! The trade–off between the number of non–doublings and storage is worth while — even for relatively low hamming weight. The number of non–doublings can be reduced clearly comparing with `binary`. The number of doublings is constant.

`brauer` and `bgmw` differ only in two points conspicuously: For low Hamming weight ($\nu \approx \frac{1}{4}\lambda$) `bgmw` computes shorter addition chains. But `brauer` beats

Hamming			algorithm	#steps			#doublings			#non-doublings			storage		
min	aver	max		min	aver	max	min	aver	max	min	aver	max	min	aver	max
25	40	58	`binary`	183	198	216		159		24	39	57		**1**	
			`brauer`	187	196	206		**156**		31	40	50		15	
			`bgmw`	184	194	206		159		25	35	47		54	
			`yacobi`	190	202	215	169	173	179	19	28	39	15	20	26
			`bocharova`	181	**191**	204		159		22	32	45		3	
			`lookahead`	179	194	218	159	167	188	18	**26**	36	14	19	25
63	80	103	`binary`	221	238	261		159		62	79	102		**1**	
			`brauer`	201	**206**	209		**156**		45	50	53		15	
			`bgmw`	204	210	216		159		45	51	57		54	
			`yacobi`	213	224	233	176	180	185	35	43	51	23	27	32
			`bocharova`	201	212	222		159		42	53	63		3	
			`lookahead`	200	215	237	165	174	190	33	**40**	51	20	25	30
104	120	141	`binary`	262	278	299		159		103	119	140		**1**	
			`brauer`	206	**208**	209		**156**		50	52	53		15	
			`bgmw`	213	216	217		159		54	57	58		54	
			`yacobi`	219	231	241	177	182	186	40	49	55	23	27	31
			`bocharova`	220	227	235		159		61	68	76		3	
			`lookahead`	205	228	266	166	180	200	35	**47**	66	16	21	28

Table 7: Output parameters for $\lambda = 160$ bit

Hamming			algorithm	#steps			#doublings			#non-doublings			storage		
min	aver	max		min	aver	max	min	aver	max	min	aver	max	min	aver	max
93	128	163	`binary`	603	638	673		511		92	127	162		**1**	
			`brauer`	600	614	628		**507**		93	107	121		31	
			`bgmw`	590	608	625		508		82	100	117		128	
			`yacobi`	607	630	652	543	551	559	61	78	94	45	54	64
			`bocharova`	571	**589**	607	511	512	515	60	77	93		11	
			`lookahead`	586	615	653	524	542	568	56	**73**	90	40	51	60
216	256	294	`binary`	726	766	804		511		215	255	293		**1**	
			`brauer`	629	635	639		**507**		122	128	132		31	
			`bgmw`	631	641	647		508		123	133	139		128	
			`yacobi`	668	684	706	560	567	575	104	116	132	68	72	78
			`bocharova`	620	**630**	641	513	513	514	106	116	127		11	
			`lookahead`	648	673	706	544	561	584	99	**111**	129	56	65	72
353	384	415	`binary`	863	894	925		511		352	383	414		**1**	
			`brauer`	636	638	639		**507**		129	131	132		31	
			`bgmw`	645	648	649		508		137	140	141		128	
			`yacobi`	678	696	712	564	571	577	113	125	136	63	70	76
			`bocharova`	622	**634**	646	511	512	515	108	**121**	133		11	
			`lookahead`	668	709	760	556	580	611	107	129	151	47	56	65

Table 8: Output parameters for $\lambda = 512$ bit

Hamming min	Hamming aver	Hamming max	algorithm	#steps min	#steps aver	#steps max	#doublings min	#doublings aver	#doublings max	#non-doublings min	#non-doublings aver	#non-doublings max	storage min	storage aver	storage max
199	257	302	`binary`	1221	1279	1324		1023		198	256	301		**1**	
			`brauer`	1202	1219	1236		1018		184	201	218		63	
			`bgmw`	1183	1205	1223		1020		163	185	203		205	
			`yacobi`	1208	1239	1264	1085	1096	1107	121	143	162	85	99	110
			`bocharova`	1139	**1165**	1181	1024	1028	1031	114	137	152		19	
			`lookahead`	1185	1220	1260	1061	1085	1118	111	**134**	152	77	93	105
459	512	563	`binary`	1481	1534	1585		1023		458	511	562		**1**	
			`brauer`	1240	1247	1250		**1018**		222	229	232		63	
			`bgmw`	1239	1247	1254		1020		219	227	234		205	
			`yacobi`	1314	1334	1356	1115	1125	1134	195	209	224	123	130	140
			`bocharova`	1218	**1230**	1242	1028	1028	1029	189	**201**	213		19	
			`lookahead`	1285	1324	1366	1098	1120	1152	182	203	222	109	117	127
728	768	815	`binary`	1750	1790	1837		1023		727	767	814		**1**	
			`brauer`	1248	1249	1250		**1018**		230	231	232		63	
			`bgmw`	1249	1253	1254		1020		229	233	234		205	
			`yacobi`	1328	1351	1376	1120	1130	1140	207	221	237	117	124	132
			`bocharova`	1210	**1231**	1244	1024	1028	1031	186	**203**	215		19	
			`lookahead`	1331	1392	1478	1124	1156	1203	207	235	275	88	100	111

Table 9: Output parameters for $\lambda = 1024$ bit

`bgmw` if the Hamming weight is $\nu \geq \frac{\lambda}{2}$. The other point to emphasize is the demand on storage: `brauer` stores only $\frac{1}{2}$ to $\frac{1}{3}$ of the number of elements of `bgmw`. But `bgmw` only stores powers of 2. `brauer` stores all elements of the set $\{1, \ldots, 2^r - 1\}$.

Results. Algorithm `binary` should only be used for relatively low Hamming weight ($\nu \approx \frac{1}{4}\lambda$). In the other case `brauer` and `bgmw` produce addition chains with much fewer elements. For $\nu > \frac{\lambda}{2}$ and according to the choice of r `brauer` and `bgmw` produce addition chains of nearly the same length. The decision which of both algorithms should be used depends on the storage: normally `brauer` should be prefered. But if these algorithms are transformed to create exponentiation algorithms and the cost for squaring is negliglible, `bgmw` is the right choice. In this case `bgmw` requires no memory because all powers of 2 can be computed for free.

10.3. Algorithms based on data compression. The three algorithms `yacobi`, `bocharova` and `lookahead` use data compression techniques to generate addition chains. We take a look at the average case in the following:

Comparing the total numbers of steps we find that `lookahead` is not useful for high Hamming weight ($\nu \approx \frac{3}{4}\lambda$). `yacobi` and `lookahead` create much longer addition chains than `bocharova` if we have $\lambda = 512$ or $\lambda = 1024$. All three algorithms need more doublings than the binary method to evaluate an addition chain. It can be generally noticed that `bocharova` is the only one of the three algorithms based on data compression with a nearly constant number of doubling steps for given λ. For the other two algorithms the doublings depend on the Hamming weight. On the other hand the number of star steps is relatively small. `bocharova` is best in the case of $\nu \geq \frac{\lambda}{2}$.

The demand on storage is fixed for `bocharova` due to the chosen parameter. But for `yacobi` and `lookahead` the demand on storage depends on the Hamming weight of the input. The worst case seems to be $\nu \approx \frac{\lambda}{2}$.
All three algorithms beat the binary method by using storage. But there is no typical 'winner'. Perhaps `bocharova` gives the best results. `lookahead` can be used for $\nu \approx \frac{\lambda}{4}$.

10.4. Comparison between the two methods. Concentrating on the total number of steps the classical algorithms are better than `yacobi` and `lookahead`. But `bocharova` seems to be quite as good as `brauer` or `bgmw` — depending on the chosen parameter. The addition chain algorithms based on data compression have one nice property: they need less star steps but more doublings than the classical algorithms. This can be interesting if doublings do not count (cf. the problem of squaring in $\mathbb{F}_{2^n}$ given in normal basis representation). But there is one point left that has to be emphasized: The length of an addition chain computed by one of the algorithms `yacobi`, `bocharova` or `lookahead` can be scattered in a wide range to the average length. For the classical algorithms the length is close to the average length. Therefore for an arbitrary input the classical algorithms `brauer` and `bgmw` should be prefered.

10.5. Addition chains: theory vs. practice. In theory and practice the binary method has been shown to be unacceptable for long bitstrings and high Hamming weight. The other five algorithms can be divided into two groups: `brauer`, `bgmw` and `bocharova` need a parameter. The theoretical values for r are optimal in an asymptotical sense. In practice the parameter r has to be chosen carefully to get good results (cf. the numerical results in Brickell *et al.* 1993). `yacobi` and `lookahead` cannot be optimized by a parameter. They produce long addition chains on average and in the worst case — both in theory and practice. The number of doublings is a little bit higher than for the other algorithms — according to the theoretical results. But it is generally noticed that the number of non–doublings is lower than for the algorithms `brauer` and bgmw. We assume two reasons for this discrepancy:

1. The estimations are asymptotically. But the examined values are fairly small — only 160 to 1024 bit.

2. For theoretical results we assume that the stored values in yacobi and lookahead can be ordered in a balanced binary tree according to the literature. For our experiments of only 1000 values this assumption may not be correct.

In theory bocharova and bgmw have the same asymptotical number of non–doubling steps: $A \leq \frac{\lambda}{\log \lambda}(1 + o(1))$. Due to our theoretical results an upper bound on A for brauer can be given by $A \leq 2\frac{\lambda}{\log \lambda}(1+o(1))$. In practice brauer is a little bit better than bgmw and bocharova comparing the number of non–doubling steps. Indeed brauer is the 'winner' of our experiments comparing only the total length of the computed addition chains. There are some reasons for this:

1. All estimates are upper bounds: In theory we give the estimate $\nu_{2^r}(m) \leq \frac{1}{r}\log_2 m$. This isn't a good upper bound if the Hamming weight is low according to the 2^r-ary representation of m.

2. The parameter r is chosen according to the theoretical results. But these results are given mainly for asymptotic purposes. Therefore a (slightly) different choice of r for bgmw or bocharova could give shorter addition chains.

The demand on storage given by theoretical results is validated by our practical experiments. Summarizing the results our experiments show that the best algorithms in theory are also most useful in practice: brauer, bgmw and bocharova.

11. Practical comparison of exponentiation algorithms

11.1. The experiment.

Implementation. Because we compare the running times of three exponentiation algorithms for $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ and various $n \in \mathbb{N}$, we describe briefly our implementation.

We implemented the three algorithms `onb`, `ggp` and `shoup` (cf. Notation 9.9) on a Sun Sparc Ultra 1 computer, rated at 143 MHz. The software is written in C++. The coefficient lists of both the polynomial and the normal basis representation are represented as arrays of 32-bit unsigned integers, and 32 consecutive coefficients are packed into one machine word. For normal basis representation we built a C++ class over $\mathbb{F}_2$ offering standard operations like cyclic shifting, and arithmetic operations like addition and multiplication according to Algorithm `Massey-Omura multiplier` (Algorithm 8.4) using the multiplication table $T_{\mathcal{N}}$. We also implemented Algorithm `fast normal basis multiplication` (Algorithm 9.4) transforming from normal basis representation to polynomial representation according to Gao *et al.* (1995a).

For polynomial arithmetic we used the software library written in C++ by J. Gerhard that is described in von zur Gathen & Gerhard (1996), Section 10. This library offers fast polynomial arithmetic over $\mathbb{F}_2$ including several algorithms for polynomial multiplication over $\mathbb{F}_2$: the classical method, Karatsuba & Ofman's algorithm and the method introduced by Cantor (1989). The algorithm of Schönhage (1977) has not been implemented. The different multiplication algorithms are used in the following way: Two polynomials of degree less than 576 are multiplied by the classical method. Polynomials of degree between 576 and 35840 are multiplied using Karatsuba & Ofman's algorithm. For polynomials of degree at least 35840 Cantor' algorithm is used.

The library also contains an implementation of Algorithm `modular composition` (Algorithm 7.23) according to Brent & Kung (1978) using classical matrix multiplication. We used this algorithm implementing modular composition within Algorithm `shoup/exponentiation with composition` (Algorithm 7.28).

Chosen input. We concentrate on field extensions over $\mathbb{F}_2$ of degree n for which an optimal normal basis exists, i.e. the normal basis corresponds to a Gauß period of type (n, k) with $k \in \{1, 2\}$. We use two different series of values for n:

- We choose $n \in \mathbb{N}, n \approx 200 \cdot i, 1 \leq i \leq 50$ (see Table 10) as *test series 1* to examine in detail practical aspects of the three exponentiation algorithms. In cryptography values for n between 512 and 1024 are often used for cryptosystems (cf. the remarks in Brickell *et al.* 1993 and Odlyzko 1985). We also want to show for which n cryptosystems based on exponentiation can be used within a CPU-time of about 60 seconds.

normal basis by Gauß period of type (n,k)		irreducible polynomial f with $\mathbb{F}_{2^n} \cong \mathbb{F}_2[x]/(f)$	normal basis by Gauß period of type (n,k)		irreducible polynomial f with $\mathbb{F}_{2^n} \cong \mathbb{F}_2[x]/(f)$
n	k	$n = \deg f$	n	k	$n = \deg f$
209	2	205	5199	2	
398	2	393	5399	2	5402
606	2	587	5598	2	
803	2	798	5812	1	
1018	1	1037	6005	2	
1199	2	1201	6202	1	
1401	2	1476	6396	1	
1601	2	1607	6614	2	6563
1791	2	1824	6802	1	6756
1996	1	1898	7005	2	
2212	1	2197	7205	2	7245
2406	2	2355	7410	1	
2613	2	2665	7602	1	
2802	1	2825	7803	2	7891
3005	2	3066	8003	2	
3202	1	3165	8218	1	
3401	2	3364	8411	2	8325
3603	2	3590	8601	2	
3802	1	3831	8802	1	
4002	1	3924	9006	2	9085
4211	2	4099	9202	1	
4401	2	4273	9396	1	
4602	1	4629	9603	2	9659
4806	2		9802	1	
5002	1		9998	2	10001

Table 10: Input values for $n \leq 10000$ (test series 1)

- The other series (*test series 2*) consists of $n \in \mathbb{N}, n \approx 2^i, 10 \leq i \leq 16$ and some intermediate values (see Table 11). Using this input we want to give an idea of the asymptotic behaviour of the three exponentiation algorithms.

normal basis by Gauß period of type (n,k)		irreducible polynomial f with $\mathbb{F}_{2^n} \cong \mathbb{F}_2[x]/(f)$	normal basis by Gauß period of type (n,k)		irreducible polynomial f with $\mathbb{F}_{2^n} \cong \mathbb{F}_2[x]/(f)$
n	k	$n = \deg f$	n	k	$n = \deg f$
1034	2	1037	23903	2	23894
2141	2	2141	32075	2	32071
4098	1	4099	43371	2	43371
8325	2	8325	51251	2	51251
16679	2	16881	61709	2	61709

Table 11: Input values for $n \approx 2^i, 10 \leq i \leq 16$ (test series 2)

The exponents are randomly chosen and uniformly distributed between $\{1, \ldots, 2^n - 1\}$. We combine Algorithm`onb` and Algorithm `ggp` with the addition chain algorithm `bgmw` according to the theoretical results. Algorithm `shoup` includes a special combination of `bgmw` and Horner's rule according to Shoup (1994).

We had to cope with two difficulties:

- We had to find irreducible polynomials of degree $\approx n$. One possibility is to choose such a polynomial randomly and then verify that it is indeed irreducible, using a probabilistic algorithm (cf. Cantor & Zassenhaus 1981; Knuth 1981, Section 4.6.2). Ben-Or (1981) suggests to test a randomly chosen monic polynomial for irreducibility by showing that it has no factors of low degree. We appropriate the idea of factorization in another way: we used the polynomial factorization software described in von zur Gathen & Gerhard (1996) and the irreducible polynomials computed by it. Most of the polynomials we used for computation are mentioned in von zur Gathen & Gerhard (1996), Table 10.6.

- We want to use optimal normal bases. The tables which are given in the literature (e.g., Menezes *et al.* 1993, Table 5.1; Jungnickel 1993, Table 3.1; Gao *et al.* 1995b, Appendix) contain only values up to $n = 2000$. We use the criteria given in Theorem 8.12 to check if there exists an optimal normal basis generated by a Gauß period for $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ (cf. Schlink 1996a, Section 7).

Although we have some differences for both test series due to these problems, our chosen inputs can be used to compare the three exponentiation algorithms in the sequel.

11.2. Remarks on the algorithms.

Shoup's algorithm. We have already mentioned that our implementation of Algorithm `shoup` uses classical matrix multiplication, i.e. the multiplication exponent $\omega = 3$. But this can be neglected comparing the three exponentiation algorithms because of two reasons:

- We have implemented a fast matrix multiplication algorithm according to Strassen (1969) using the version of Winograd (1971). Figure 11.1 shows that the crossover point in this implementation is about $n = 1000$ rows/columns. But using matrix multiplication within modular composition according to Brent & Kung (1978) means that we can use classical matrix multiplication in our implementation for field extensions over $\mathbb{F}_2$ of degree $n < 1000000$ because we multiply only matrices in $\mathbb{F}^{\sqrt{n}\times\sqrt{n}}$ for given $n \in \mathbb{N}$.

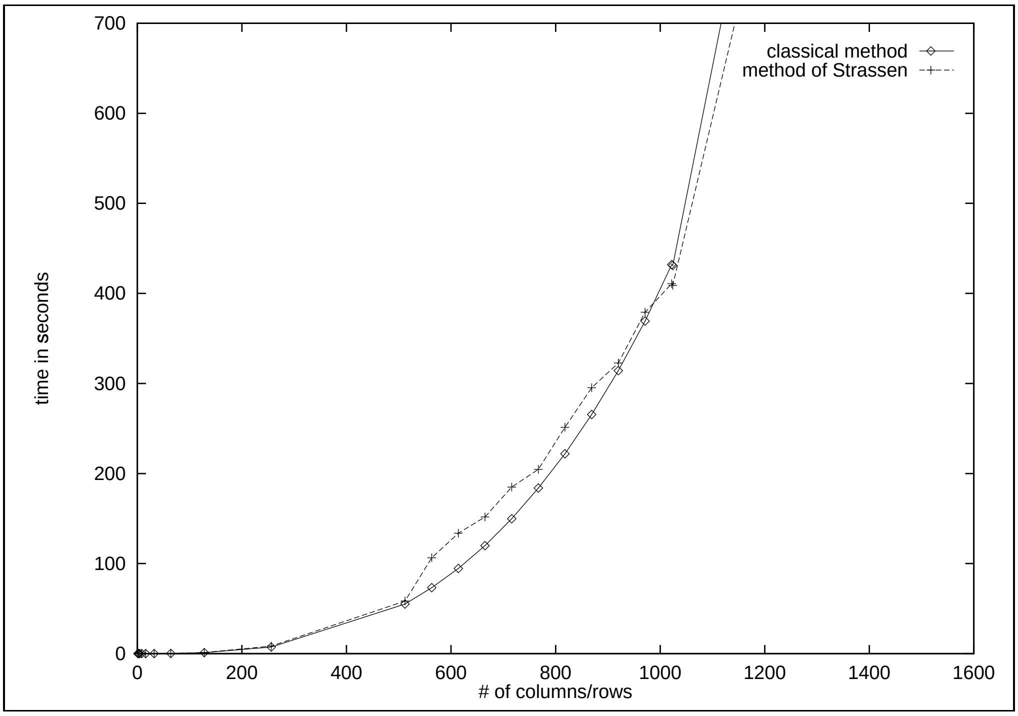

Figure 11.1: Comparison of classical matrix multiplication and *à* la Strassen

- The library used for polynomial arithmetic uses Karatsuba & Ofman to multiply polynomials of degree $576 \leq n < 35840$ which is caused in the crossover points of the implementation (cf. von zur Gathen &

Gerhard 1996). But then the (theoretical) running time is dominated by polynomial multiplications and not by matrix multiplication (cf. also the remark of Shoup 1994). For Karatsuba & Ofman we have $\mathsf{M}(n) = O(n^{\log_2 3})$ (Lemma 7.2) and hence a total of $O(\frac{n^{2.6}}{\log n})$ for Algorithm `shoup`. This corresponds to our practical results.

Optimal normal bases. We concentrate on optimal normal bases $\mathcal{N}$ for practical tests. Therefore the density of $T_{\mathcal{N}}$ doesn't depend on k, because $c_{\mathcal{N}} = 2n - 1$. The multiplication matrix $T_{\mathcal{N}}$ is sparse and so we used a list structure for implementation only storing the positions with non–zero entries.

Our implementation confirms the customary (theoretical) assumption that the cost of raising to a power of 2 can be neglected using a normal basis representation over $\mathbb{F}_2$. The running–times for cyclic shifts are nearly unmeasurable and very close to zero. Figure 11.2 shows that raising to a 2nd power is indeed for free compared to the time needed for multiplication using the multiplication matrix $T_{\mathcal{N}}$.

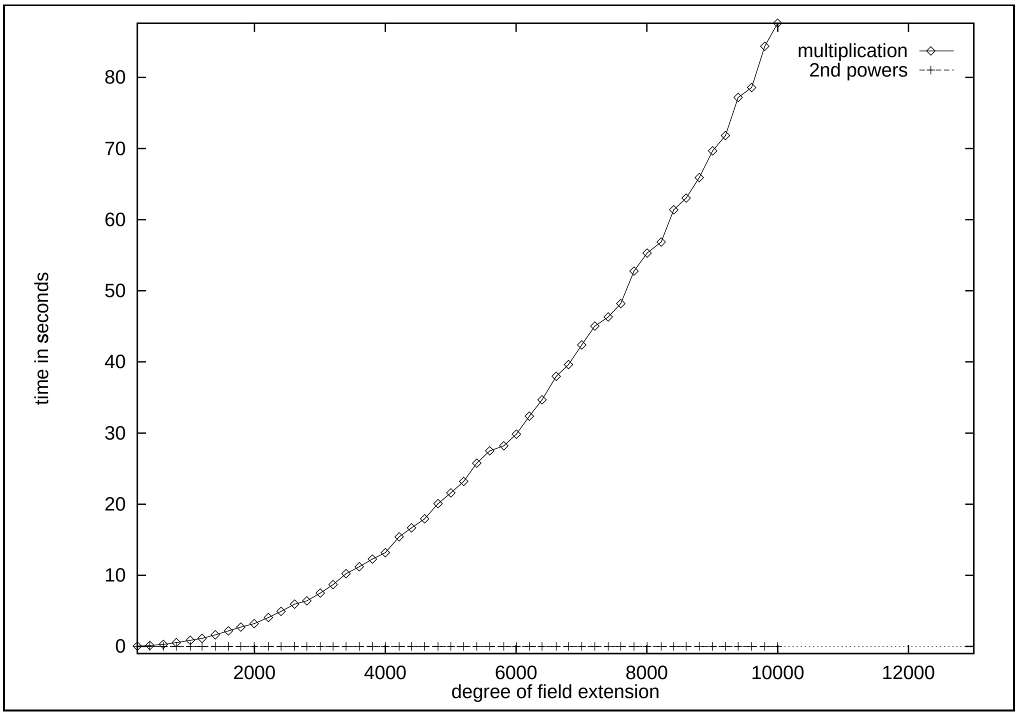

Figure 11.2: Running times for multiplication using $T_{\mathcal{N}}$ and raising to a 2nd power for a normal basis representation. This multiplication is used in Algorithm `onb`.

Polynomial multiplication within normal basis representation. In theory Algorithm `fast normal basis multiplication` depends not only on the degree n of the field extension over $\mathbb{F}_2$ but also on $k \in \mathbb{N}$ with n, k satisfying the conditions of Theorem 8.12. Figure 11.3 shows that this is also true for practical results: for optimal normal bases with $k = 2$ we have a multiplication time of about 2–3 times the multiplication time for $k = 1$. In theory we have $k\mathsf{M}(n) \leq \mathsf{M}(kn) \leq k^2\mathsf{M}(n)$ and hence it is important to find $k \in \mathbb{N}$ as small as possible for given n independent of the search for normal bases $\mathcal{N}$ with low density $c_{\mathcal{N}}$.

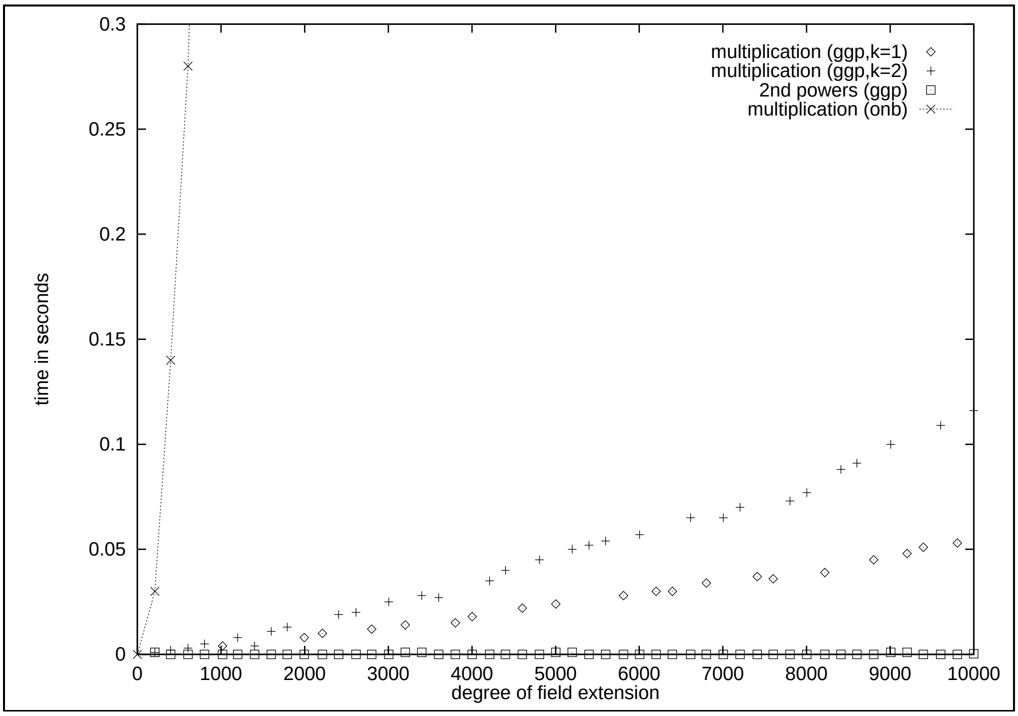

Figure 11.3: The dependence of the multiplication time in `ggp` on k

To eliminate the dependence on k for asymptotic results we selected normal bases with $k = 2$ for test series 2 (except for $n = 4098$, where $k = 1$). Finally we emphazise one further point: because the crossover point within the library between the multiplication algorithm of Karatsuba & Ofman and Cantor is about degree $nk = 35840$ we use Cantor's method for field extensions of degree more than $n = 17920$. Examining the asymptotical behaviour (test series 2) of `ggp` we therefore mainly use Cantor's algorithm for polynomial multiplication. Algorithm `shoup` uses mostly the algorithm of Karatsuba & Ofman. For field extensions of degree $n \leq 10000$ (test series 1) both ggp and shoup contain Karatsuba & Ofman for polynomial multiplication.

11.3. Results.

Field extensions of degree at most 10^4. The results of our practical comparison for $\mathbb{F}_{2^n}, n \leq 10000$ are clear with respect to normal basis representation (cf. Figure 11.4): using a multiplication matrix — even with low density — for multiplication is too slow. A software based implementation of Algorithm `Massey-Omura multiplier` is only useful for small field extensions over $\mathbb{F}_2$. For degree $n > 1000$ this is too time–consuming. This corresponds to our theoretical results: onb uses $O(\frac{n^3}{\log n})$ operations in $\mathbb{F}_2$ (Theorem 8.26), but ggp and shoup both use about $O(\frac{n^{2.6}}{\log n})$ operations because for $\deg f = n \leq 10000$ Karatsuba & Ofman's algorithm is implemented for polynomial multiplication with $\mathsf{M}(n) = O(n^{\log_2 3})$.

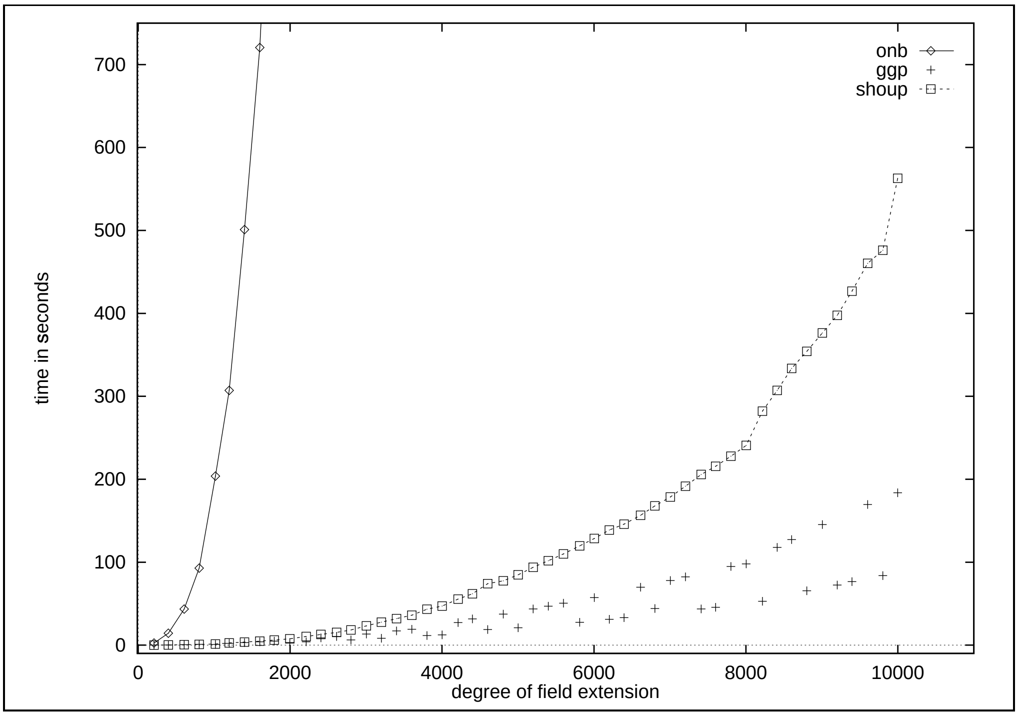

Figure 11.4: Comparison of the three exponentiation algorithms for $n \leq 10000$

On the other hand Algorithm ggp beats Algorithm shoup. But as discussed in the previous section this depends on the normal basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ for given $n \in \mathbb{N}$. If there exists a normal basis generated by a Gauß period with $k = 2$

onb			ggp			shoup	
n	k	t/sec	n	k	t/sec	n	t/sec
209	2	2.44	209	2	0.09	205	0.06
398	2	14.30	398	2	0.26	393	0.23
606	2	43.47	606	2	0.6	587	0.50
803	2	92.86	803	2	1.0	798	0.89
1018	1	203.79	1018	1	0.9	1037	1.80
1199	2	307.07	1199	2	2.18	1201	2.87
1401	2	500.96	1401	2	3.08	1476	4.53
1601	2	720.60	1601	2	3.95	1607	5.40
1791	2	1049.14	1791	2	4.75	1824	7.05
1996	1	1251.76	1996	1	3.19	1898	7.65
2212	1	1738.70	2212	1	4.04	2197	10.90
2406	2	2256.20	2406	2	8.81	2355	13.02
2613	2	2921.65	2613	2	10.45	2665	17.39
2802	1	3332.23	2802	1	6.28	2825	19.90
3005	2	4138.09	3005	2	13.41	3066	25.97
3202	1	5037.51	3202	1	8.28	3165	28.30
3401	2	6088.73	3401	2	17.23	3364	33.26
3603	2	7314.72	3603	2	19.18	3590	38.45
3802	1	8296.18	3802	1	11.54	3831	44.38
4002	1	9513.86	4002	1	12.39	3924	46.85
4211	2	11348.90	4211	2	27.27	4099	54.42
4401	2	13025.20	4401	2	31.61	4273	61.20
4602	1	15209.50	4602	1	18.78	4629	75.85
4806	2	16138.80	4806	2	37.40		
5002	1	17545.40	5002	1	20.93		
5199	2	20449.90	5199	2	43.70		
5399	2	21961.90	5399	2	46.92	5402	109.23
5598	2	24424.30	5598	2	50.62		
5812	1	27082.60	5812	1	27.56		
6005	2	30688.90	6005	2	57.39		
			6202	1	31.13		
			6396	1	33.18		
			6614	2	69.76	6563	165.39
			6802	1	44.12	6756	177.16
			7005	2	77.96		
			7205	2	82.39	7245	207.50
			7410	1	43.63		
			7602	1	45.60		
			7803	2	94.78	7891	248.36
			8003	2	97.88		
			8218	1	52.80		
			8411	2	117.90	8325	318.92
			8601	2	127.33		
			8802	1	65.49		
			9006	2	145.43	9085	415.14
			9202	1	72.45		
			9396	1	76.56		
			9603	2	169.51	9659	488.38
			9802	1	83.83		
			9998	2	183.65	10002	531.11

Table 12: Running times for test series 1

or even $k = 1$, ggp is faster than shoup. In theory both algorithms need about $O(\frac{n^{2.6}}{\log n})$ operations. But a closer look at the hidden constants shows that in ggp for $k = 2$ we have $c_{\mathsf{M}} = k^{\log_2 3}\frac{n}{\log_2 n} = 3\frac{n}{\log_2 n}$ (Corollary 9.8) and for shoup we have $c_{\mathsf{M}} = 9\frac{n}{\log_2 n}$ (Corollary 7.32). This factor of 3 is also valid in practice (cf. Table 12): ggp is about 2–3 times faster than shoup for $k = 2$. Using our theoretical results this will change for $k \geq 4$ because $9 \leq k^{\log_2 3}$ for $k \geq 4$; then shoup should be faster than ggp. But ggp is best if an optimal normal basis exists for a field extension of degree n over $\mathbb{F}_2$. Using Definition 8.18 we get the following result:

REMARK 11.1. *If $\kappa_2'(n) \leq 4$ then Algorithm* ggp *should be used for exponentiation in $\mathbb{F}_{2^n}$. Otherwise Algorithm* shoup *should be prefered.*

Exponentiation in huge field extensions. Algorithm onb should not be used for field extensions of high degree over $\mathbb{F}_2$. For degree $n = 4098$ onb has a running time of 2h 53' (see Table 13) — which is of no practical use. Algorithm

onb			ggp			shoup	
n	k	t/sec	n	k	t/sec	n	t/sec
1034	2	205.36	1034	2	1.63	1037	1.67
2141	2	1595.74	2141	2	7.28	2141	9.47
4098	1	10401.90	4098	1	14.5	4099	51.98
8325	2	78019.00	8325	2	127.76	8325	302.86
			16679	2	565.89	16881	1759.61
			23903	2	1064.7	23894	4489.31
			32075	2	1856.83	32071	7545.09
			43371	2	3593.04	43371	15530.10
			51251	2	4990.81	51251	22039.70
			61709	2	6973.74	61709	34297.50

Table 13: Running times for test series 2

ggp beats shoup clearly for bigger $n \in \mathbb{N}$ when fixing $k = 2$ (see Figure 11.5). There are some reasons for this result:

- ggp uses the multiplication algorithm of Cantor (1989) for $n \geq 17920$. But then $\mathsf{M}(n) = O(n(\log n)^3)$ and $\mathsf{M}(kn) \approx k(\log k)^3\mathsf{M}(n)$. For Karatsuba & Ofman we have $\mathsf{M}(kn) \approx k^{\log_2 3}\mathsf{M}(n)$. Hence in ggp we have $c_{\mathsf{M}} < 2\frac{n}{\log_2 n}$ for $n \geq 17920$ instead of $c_{\mathsf{M}} = 3\frac{n}{\log_2 n}$ for smaller n.
- shoup uses Cantor's method for $n \geq 35840$. But then the classical matrix method dominates the number of operations in the theoretical estimates.

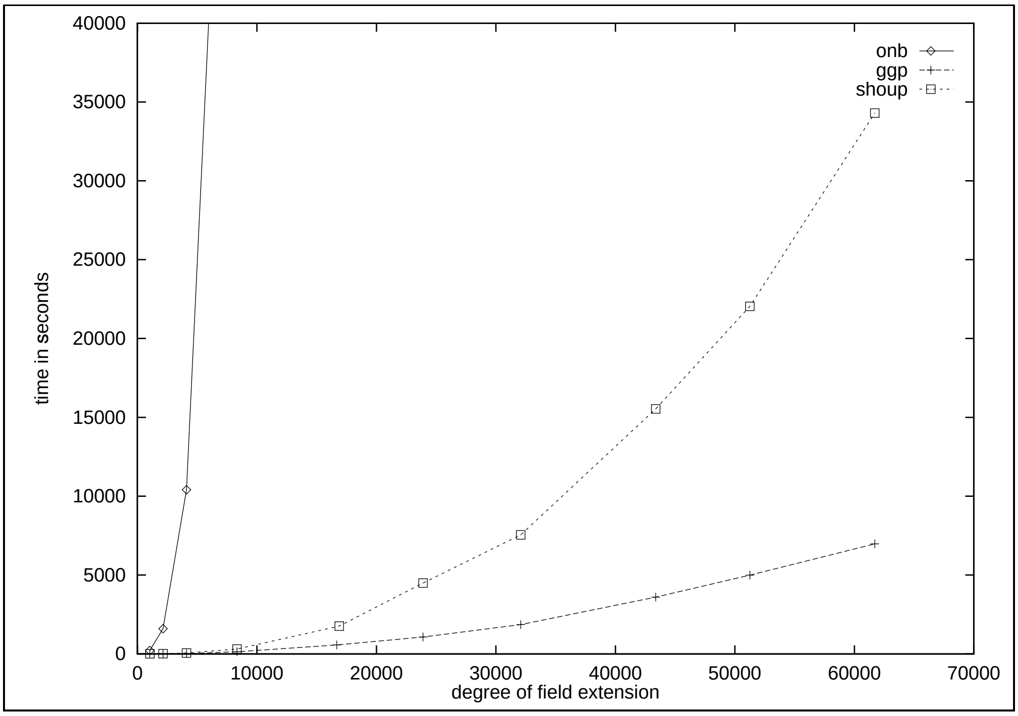

Figure 11.5: Comparison of the three exponentiation algorithms for $n \approx 2^i, 10 \le i \le 16$

We have already shown that in our implementation this plays no crucial role because the crossover point between classical matrix multiplication and Strassen's algorithm is about $\sqrt{n} = 1000$ which means $n = 1000000$. But for large field extensions modular composition is nevertheless time–consuming.

Finally we can summarize:

- `ggp` is a very good exponentiation algorithm if there exists an optimal normal basis (or even a normal basis with small k) for a given field extension $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$. We easily can compute the necessary map to go from normal basis representation to polynomial representation and vice versa.
- `shoup` can be used for all $n \in \mathbb{N}$. If $\kappa_2'(n) > 4$ and hence no normal basis with small k exists then this algorithm beats `ggp`.

Both algorithms can be used even for exponentiation in huge field extensions over $\mathbb{F}_2$.

12. Conclusion

At the end we want to outline the main properties for a fast exponentiation algorithm in $\mathbb{F}_{2^n}, n \in \mathbb{N}$:

1. The algorithm should use fast matrix multiplication. We have shown that multiplication by multiplication tensors doesn't work efficiently. Classical polynomial arithmetic isn't either fast enough even for relatively small n.

2. The algorithm should be based upon an addition chain for the exponent e with a small number of total steps and a small number of non–doubling steps. This is illustrated in Figure 12.1 where `ggp` based upon `binary` is compared to `ggp` using `bgmw`.

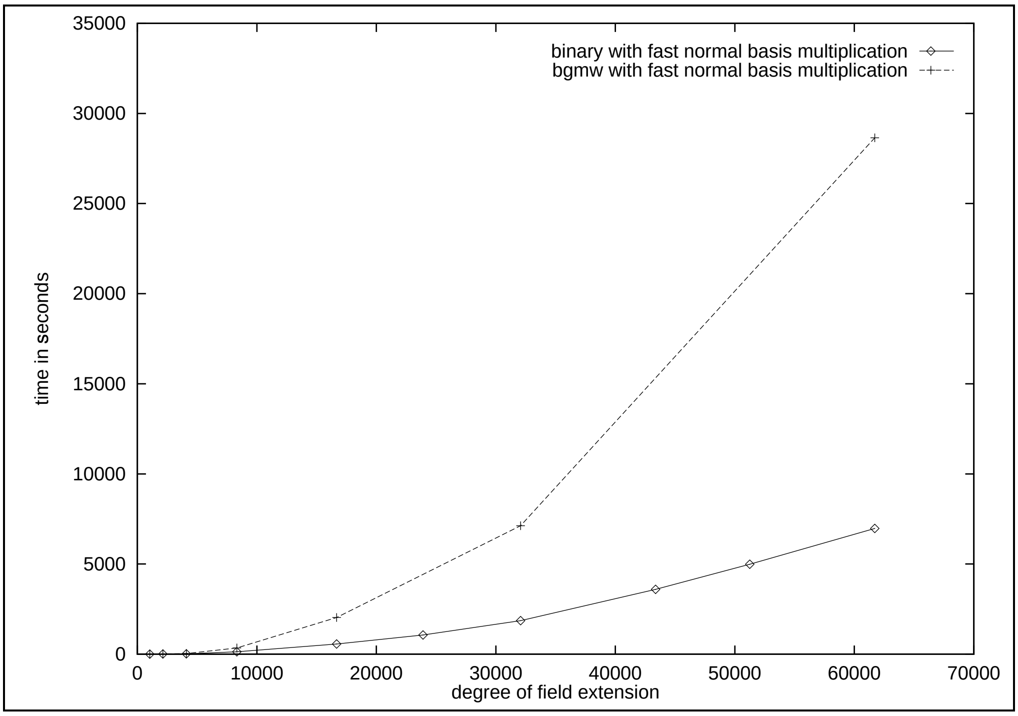

Figure 12.1: Comparison of the algorithms `binary` and `bgmw` using `fast normal basis multiplication`

3. The algorithm should offer a cheap way to compute $\alpha^{2^m} \in \mathbb{F}_{2^n}$ for $m \in \mathbb{N}$ and $\alpha \in \mathbb{F}_{2^n}$. A very efficient way is using the properties of a normal basis representation. Then raising to a power of 2 is just a cyclic shift of the coefficients.

The most important point for a fast exponentiation algorithm is to combine these three properties.

References

G. B. AGNEW, R. C. MULLIN, AND S. A. VANSTONE, Fast exponentiation in $GF(2^n)$. In *Advances in Cryptology—EUROCRYPT '88*, ed. C. G. GÜNTHER, vol. 330 of *Lecture Notes in Computer Science*, 251–255. Springer, Berlin, 1988.

A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading MA, 1974.

D.W. ASH, I.F. BLAKE, AND S.A. VANSTONE, Low complexity normal bases. *Discrete Applied Mathematics* **25** (1989), 191–210.

E. BACH AND J. SHALLIT, Factoring with cyclotomic polynomials. *Math. Comp.* **52** (1989), 201–219.

M. BEN-OR, Probabilistic algorithms in finite fields. In *Proc. 22nd IEEE Symp. Foundations Computer Science*, 1981, 394–398.

J. BERSTEL AND S. BRLEK, On the length of word chains. *Inform. Process. Lett.* **26** (1987), 23–28.

T. BETH, W. GEISELMANN, AND F. MEYER, Finding (good) normal bases in finite fields. In *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation / ISSAC'91*, ed. S. WATT, 1991, 173–178.

I. BOCHAROVA AND B. KUDRYASHOV, Fast exponentiation in cryptography. In *Applied algebra, algebraic algorithms and error correcting codes: proceedings / 11th International Symposium AAECC*, ed. G. COHEN, Lecture notes in computer science **948**, Berlin, 1995, Springer, 146–157.

G. BRASSARD AND P. BRATLEY, *Algorithmics - Theory & Practice*. Prentice Hall, 1988.

A. BRAUER, On addition chains. *Bull. Amer. Math. Soc.* **45** (1939), 736–739.

R. P. BRENT AND H. T. KUNG, Fast algorithms for manipulating formal power series. *J. Assoc. Comput. Mach.* **25** (1978), 581–595.

E. BRICKELL, D. GORDON, K. MCCURLEY, AND D. WILSON, Fast exponentiation with precomputation. In *Advances in cryptology: Proceedings / EUROCRYPT '92*, ed. R. RUEPPEL, Lecture notes in computer science **658**, Berlin, 1993, Springer, 200–207.

H. BRUNNER, A. CURIGER, AND M. HOFSTETTER, On computing multiplicative inverses in $GF(2^m)$. *IEEE Transactions on Computers* **42**(8) (1993), 1010–1015.

D. G. Cantor, On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A* **50** (1989), 285–300.

D. G. Cantor and E. Kaltofen, On fast multiplication of polynomials over arbitrary algebras. *Acta. Inform.* **28** (1991), 693–701.

D. G. Cantor and H. Zassenhaus, A new algorithm for factoring polynomials over finite fields. *Math. Comp.* **36** (1981), 587–592.

D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions. *J. Symb. Comp.* **9** (1990), 251–280.

W. Diffie and M. Hellman, New directions in cryptography. *IEEE Trans. Inform. Theory* **22** (1976), 644–654.

P. Downey, B. Leong, and R. Sethi, Computing sequences with addition chains. *SIAM J. Comput.* **10**(3) (1981), 638–646.

T. ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on information theory* **IT-31**(4) (1985), 469–472.

S. Gao and H. W. Lenstra, Optimal normal bases. *Designs, Codes, and Cryptography* **2** (1992), 315–323.

S. Gao, J. von zur Gathen, and D. Panario, Gauss periods and fast exponentiation in finite fields. In *Proc. Latin '95, Valparaiso, Chile*, Springer Lecture Notes in Computer Science **911**, 1995a, 311–322.

S. Gao, J. von zur Gathen, and D. Panario, Gauss periods, primitive normal bases, and fast exponentiation in finite fields. Technical Report 296-95, Dept. Computer Science, University of Toronto, 1995b.

J. von zur Gathen, Efficient and optimal exponentiation in finite fields. *Comput complexity* **1** (1991), 360–394.

J. von zur Gathen, Processor-efficient exponentiation in finite fields. *Inform. Process. Lett.* **41** (1992), 81–86.

J. von zur Gathen and J. Gerhard, Skript Computeralgebra I, 1995.

J. von zur Gathen and J. Gerhard, Arithmetic and factorization of polynomials over $\mathbb{F}_2$. Technical Report tr-rsfb-96-018, University of Paderborn, Germany, 1996. 43 pages.

J. von zur Gathen and M. Giesbrecht, Constructing normal bases in finite fields. *J. Symb. Comp.* **10** (1990), 547–570.

J. von zur Gathen and V. Shoup, Computing Frobenius maps and factoring polynomials. *Computational complexity* **2** (1992), 187–224.

W. Geiselmann, *Algebraische Algorithmenentwicklung am Beispiel der Arithmetik in endlichen Körpern.* Dissertation, Universität Karlsruhe, Aachen, 1994.

T. Itoh and S. Tsujii, A fast algorithm for computing multiplicative inverses in $gf(2^m)$ using normal bases. *Information and Computation* **78** (1988), 171–177.

F. Jelinek and K. Schneider, On variable-length-to-block coding. *IEEE Transactions on Information Theory* **IT-18**(6) (1972), 765–774.

D. Jungnickel, *Finite Fields: Structure and Arithmetics.* BI Wissenschaftsverlag, Mannheim, 1993.

A. Karatsuba and Y. Ofman, Умножение многозначных чисел на автоматах. *Dokl. Akad. Nauk USSR* **145** (1962), 293–294. Multiplication of multidigit numbers on automata, Soviet Physics–Doklady **7** (1963), 595-596.

D. E. Knuth, *The Art of Computer Programming, Vol.2, Seminumerical Algorithms.* Addison-Wesley, Reading MA, 2 edition, 1981.

R. Lidl and H. Niederreiter, *Finite Fields*, vol. 20 of *Encyclopedia of Mathematics and its Applications.* Addison-Wesley, Reading MA, 1983.

M. Lothaire, *Combinatorics on Words.* Addison–Wesley Reading, MA, 1983.

J. L. Massey and J. K. Omura, Computational method and apparatus for finite fields arithmetic, 1981. U. S. Patent Application.

Alfred J. Menezes, Ian F. Blake, XuHong Gao, Ronald C. Mullin, Scott A. Vanstone, and Tomik Yaghoobian, *Applications of finite fields.* Kluwer Academic Publishers, Norwell MA, 1993.

R. Moenck, Fast computation of gcd's. In *Proc. 5th Ann. ACM Symp. Theory of Computing*, 1973, 142–151.

R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson, Optimal normal bases in GF(p^n). *Discrete Applied Math.* (1989), 149–161.

A. Odlyzko, Discrete logarithms and their cryptographic significance. In *Advances in Cryptology, Proceedings of Eurocrypt 1984.* Springer-Verlag, 1985, 224–314.

V. Pan, *How to multiply matrices faster*, vol. 179 of *Lecture Notes in Computer Science.* Springer Verlag, New York NY, 1984.

R. L. RIVEST, A. SHAMIR, AND L. ADLEMAN, A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM* **21** (1978), 120–126.

P. DE ROOIJ, Efficient exponentiation using precomputation and vector addition chains. In *Advances in cryptology: proceedings / EUROCRYPT '94*, ed. A. DE-SANTIS, Lecture notes in computer science **950**, Berlin, 1995, Springer, 389–399.

S. SCHLINK, Normalbasen mit hilfe von verallgemeinerten gauß–perioden, 1996a. Diplomarbeit.

S. SCHLINK, Normal bases via Gauß periods. Technical report, FB 17 Mathematik–Informatik, Universität–GH Paderborn, 1996b.

A. SCHÖNHAGE, Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica* **1** (1971), 139–144.

A. SCHÖNHAGE, Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Inf.* **7** (1977), 395–398.

A. SCHÖNHAGE AND V. STRASSEN, Schnelle Multiplikation großer Zahlen. *Computing* **7** (1971), 281–292.

V. SHOUP, Exponentiation in $GF(2^n)$ using fewer polynomial multiplications. preprint, 1994.

D. R. STINSON, Some observations on parallel algorithms for fast exponentiation in $GF(2^n)$. *SIAM J. Comput.* **19** (1990), 711–717.

V. STRASSEN, Gaussian elimination is not optimal. *Numer. Mathematik* **13** (1969), 354–356.

V. STRASSEN, The computational complexity of continued fractions. *SIAM J. Comput.* **12** (1983), 1–27.

B. P. TUNSTALL, *Synthesis of noiseless compression codes*. Ph.d. dissertation, Georgia Inst. Technol., 1968.

A. WASSERMANN, Zur Arithmetik in endlichen Körpern. *Bayreuther Math. Schriften* **44** (1993), 147–251.

S. WINOGRAD, On multiplication of 2×2 matrices. *Linear Algebra and Appl.* **4** (1971), 381–388.

Y. YACOBI, Exponentiating faster with addition chains. In *Advances in cryptology: proceedings / EUROCRYPT '90*, ed. I. DAMGARD, Lecture notes in computer science **473**, Berlin, 1991, Springer, 222–229.

J. Ziv and A. Lempel, Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory* **IT-24**(5) (1978), 530–536.

Erklärung

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Paderborn, den 28.10.1996