Diplomarbeit

Brauer addition-subtraction chains

Martin Otto

martin@martin-otto.de

Fachbereich 17 (Mathematik/Informatik)

Universität Paderborn

Betreuer: Prof. Dr. Joachim von zur Gathen

October 14, 2001

Martin Otto
Moorkampstraße 9
59558 Lippstadt

Erklärung

Ich versichere, daß ich diese Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß diese 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 14.10.2001

Contents

Introduction **5**
Motivation – elliptic curve cryptosystems 7

1 Addition chains **11**
1.1 Introduction to addition chains 11
1.1.1 Exchanging additions and doublings 13
1.1.2 Some definitions and remarks on the notation 15
1.1.3 Limits and generalizations of addition chains 16
1.2 Algorithms for addition chains 22
1.2.1 The binary method (repeated squaring) 22
1.2.2 The m-ary method / Brauer's method 24
1.2.3 Sliding window methods 25
1.2.4 The factor method 27
1.2.5 The continued fractions method / Euclid's method . . 29
1.2.6 Knuth's powertree method 34
1.2.7 The BGMW-algorithm 35
1.2.8 The data compression method by Yacobi 40
1.2.9 Data compression by Bocharova and Kudryashov . . . 43
1.2.10 Other approaches 48
1.2.11 Summary 49

2 Cost analysis of windowing addition chains **51**
2.1 Definitions 51
2.2 The binary method 56
2.2.1 Cost analysis 57
2.2.2 Changing the direction of computation 58
2.3 The m-ary method and the Brauer method 58
2.3.1 Cost analysis of the precomputation step 59
2.3.2 Cost analysis of the exponentiation step 62
2.3.3 Cost analysis of an m-step 63
2.3.4 Complete cost analysis of the m-ary method 64
2.3.5 Brauer's method – a practical restriction 66
2.3.6 Determining an optimal value for the window size . . 68

3 Addition-subtraction chains **71**
3.1 Chances and limitations of addition-subtraction chains 73
3.2 Signed binary digit representations 75
3.3 The non-adjacent form (NAF) 78
3.3.1 Creating the NAF from the binary expansion 78
3.3.2 Creating the NAF from a signed binary representation 83
3.4 Properties of the NAF 92
3.4.1 Determining the number of NAF strings with length n 93
3.4.2 Determining the average length of the NAF 101
3.4.3 Determining occurrence and distribution of digits . . . 105
3.5 Generalizations of the NAF 108

4 Exponentiation using the NAF **111**
4.1 The NAF and the binary method 112
4.1.1 Cost analysis 112
4.2 The NAF and Brauer's approach 117
4.2.1 Analysis of the precomputation 119
4.2.2 Analysis of the main part 127
4.2.3 Experimental results 137
4.3 Cost comparisons 139
4.3.1 Cost comparison for the binary method 139
4.3.2 Cost comparison for the Brauer method 141
4.4 A generalized average case analysis 151
4.5 Conclusions 163
4.6 Beyond NAF based algorithms 167

A Optimal addition chains **169**

List of Figures **173**

List of Tables **174**

Bibliography **175**

Index **180**

Introduction

Addition chains are a very old concept. It has been used for the purpose of exponentiating numbers as early as 2000 BC in ancient Egyptian mathematics (see [Knu97], p.362). These days, addition chains have grown independent from their major application, exponentiation strategies, as questions about the optimal (shortest) length, about the complexity of computing a shortest addition chain and about efficient algorithms for a lot of applications are still awaiting answers from modern mathematics.
But even the concept of addition chains has not been fully exploited yet. In general, only the operation of adding two values to their sum seems to be feasible, because the corresponding operation in most applications is computing the power of a given number. In these general settings, the inverse operation of the addition, the subtraction, requires the explicit computation of a multiplicative inverse, a task that can be a quite costly operation in general. However, there are important exceptions, for example elliptic curve arithmetic, where inversion is cheap or free. For these fields of application, inversion can lead to a valuable improvement, to a faster exponentiation.
And faster exponentiation strategies are needed in a lot of applications, namely in modern cryptography, where whole cryptosystems like the RSA scheme (see [Sti95],§4) or the ElGamal scheme (see [Sti95],§5.1) are based upon exponentiations. Here, the computation of powers accounts for a relevant part of the computational time. Several concepts for improvements exist, however, research for improvements in this field is still necessary and new concepts have to be analyzed and optimized for better use.
The aim of this thesis is to use the inversion and hence addition-subtraction chains and to analyze when this approach to exponentiation is better than the traditional approach using only additions and doublings.
In the common literature, addition chain algorithms are evaluated and judged upon by results about the total length of the addition chains created and hence about the total number of arithmetical operations. This thesis gives results separately for every operation involved, because examples from practical applications show that the computational costs depend on the implementation and differ strongly. Hence, for practical considerations, the real costs play an important role in choosing the "right" algorithm. Therefore, the two most popular algorithms, the binary method and Brauer's

method, are analyzed in detail in their traditional version in chapter 2, they are evolved in chapters 3 and 4 into a version utilizing addition-subtraction chains by using a canonical signed binary digit representation of the exponent, the non-adjacent form (NAF), and the derived methods are analyzed in detail in chapter 4. As a result, cost comparison inequalities are formulated, which can be used to determine for any practical scenario of costs, which variant of those two methods is faster on every input or on the average. The analyses show that the problem to determine the best algorithm, has no general solution, but depends on a particular application.
Chapter 1 gives a brief overview over the numerous approaches and ideas exploited to form addition chain algorithms. It also gives a graphical summary over the generalization steps of the methods, where the most successful are all generalizations of the basic concept of the 3000 year old binary method. Much work has been spent on developing and improving existing algorithms, hence, most of the ideas and concepts presented within this thesis have been taken from literature or repeat commonly known results. Citations will be given wherever possible. This thesis contains a bibliography and an index, that also serves as a list of symbols. The following results are new to this thesis and seem not to have been published before:

- All analyses give individual results for the different arithmetical operations involved. This is not common in the literature.
- In section 3.4, the results of the recursive formula for the number of NAF strings (3.8) as well as the general explicit formula (3.9) have not been published before. The special case $q = 2$ has been shown by several authors. The results of theorems 3.19 and 3.20 have also been developed for this thesis.
- The used model of addition chains, which doesn't allow to exchange additions and doublings freely and therefore corrects a major problem of the common model, has not yet been widely used in literature.
- Finally, the analyses of the NAF-based methods in chapter 4 have not yet been given separated by the different arithmetical operations and the concept to formulate cost comparison inequalities to pay respect to the difference in applications is new to this thesis.

Acknowledgements. I would like to thank Prof. Dr. Joachim von zur Gathen for his support and especially Dr. Michael Nöcker, who supported me while Prof. Dr. von zur Gathen was convalescing.

Motivation – elliptic curve cryptosystems

The use of subtractions to generate a chain resulting in the computation of a given $e \in \mathbb{N}$ is infeasible in a general approach to exponentiation, because subtraction implies the need for multiplicative inverses, which may be very costly to compute and which may sometimes not even exist.
But in the field of cryptography, cryptosystems are also based on other fields, rings or groups besides the integers $\mathbb{Z}$ or integers modulo an integer n, $\mathbb{Z}_n := \mathbb{Z}/n\mathbb{Z}$. Especially elliptic curves have been used to implement new cryptosystems. Here, subtraction is a feasible operation, because the corresponding operation in the cryptosystem requires no further computations than the corresponding operation of the addition. To explain this major motivation of the thesis, the very basic facts about elliptic curves are briefly reviewed on the next pages (see [GG99] for a deeper insight).

Definition 1 (an elliptic curve)
Let F be a field of characteristic $\notin \{2,3\}$, and let $x^3 + a \cdot x + b \in F[x]$ be square-free. Then the set

$$\begin{aligned} E &:= \{y^2 - (x^3 + a \cdot x + b) = 0\} \cup \{\mathcal{O}\} \\ &= \{(u,v) : v^2 = u^3 + a \cdot u + b\} \cup \{\mathcal{O}\}, \end{aligned}$$

with $\mathcal{O}$ denoting the point at infinity, is called an elliptic curve.
a and b are called the Weierstrass coefficients of E.

Elliptic curves happen to have a misleading name, because they have very little to do with ellipses. The above definition uses the Weierstrass equation with the Weierstrass coefficients to define an elliptic curve. Besides that definition, there are also other equivalent ways to define elliptic curves. Elliptic curves can be provided with a group operation called addition, which gives an elliptic curve a group structure.

Definition 2 (addition on elliptic curves)
The addition on an elliptic curve, denoted + as usual, can be defined in a geometrical way:
Let $P = (P_x, P_y) \in E$ be a point on the elliptic curve, then the negative of P is defined as the point's reflection at the x-axis, namely $-P = (P_x, -P_y)$ with $-\mathcal{O} := \mathcal{O}$. Note that due to this property, the computation of inverses for the addition requires no operation (this is important for algorithmic applications).
To determine the sum $P + Q$ of two points $P = (P_x, P_y), Q = (Q_x, Q_y) \in E$, one has to find the third intersection of the line defined by P and Q with E. Let this third intersection be $S = (S_x, S_y) \in E$, then $P + Q$ is defined as $P + Q = (S_x, -S_y)$, the reflection of S at the x-axis.
Additionally, there are some special cases, which have to be defined separately:

1. *If $P = Q \neq \mathcal{O}$, then the line defined by P and Q is considered to be the tangent line at P and $P + Q$ is the reflection of the second intersection of this tangent line with the elliptic curve. Any elliptic curve is smooth in the geometrical sense (see [GG99]), hence this tangent is well defined.*

2. *If $Q = \mathcal{O}$, then the line defined is considered to be the vertical line through the point P, hence $P + \mathcal{O} = -(-P) = P$. This also holds if $P = Q = \mathcal{O}$.*

3. *If $Q = -P$, the line defined by P and Q is obviously the vertical line through P and Q, therefore, the sum $P + (-P) = -\mathcal{O} = \mathcal{O}$.*

Now this definition leads to a commutative group structure on E:

Theorem 3 (a group structure on elliptic curves)
Let E be an elliptic curve and the addition $+$ be defined as in definition 2. Then $(E, +)$ is an Abelian group.

The basics of the proof to this theorem can be found in [GG99]. Note that $\mathcal{O}$ is the neutral element and for a given point $P \in E$, $-P$ is the inverse. The closure of E under the addition, the associativity and the commutativity can easily be checked, preferably using a computer algebra system.

Definition 4 (order of a point on an elliptic curve)
The order d of a point P on an elliptic curve E is the smallest positive integer such that $d \cdot P = \mathcal{O}$. d is a divisor of the total number of points on E.

With any elliptic curve, a cryptosystem can now be defined. The following protocol implements the well known ElGamal crypto scheme using elliptic curves. It is one example for the use of elliptic curves in cryptography, others exist, too. The protocol takes the classical cryptographic form, where Alice and Bob represent the two partners of communication:

Protocol 5 (ElGamal cryptosystem using elliptic curves)

1. *Choose an elliptic curve $E = \{y^2 - (x^3 + a \cdot x + b) = 0\} \cup \{\mathcal{O}\}$ over $\mathbb{F}_q = \mathbb{Z}/q\mathbb{Z}$, with a large prime $q \geq 2^{256}$. Choose a point $P \in E$ with a very high order d. It is assumed that E, P and the rules of translating a written text into a point on E are publicly known.*

2. Alice *chooses a secret key $2 \leq s_A \leq d - 1$ and computes the public key $p_A := s_A \cdot P$*

3. Bob *translates his message into a point M on E, chooses his secret key $2 \leq s_B \leq d-1$ and computes the tuple $(s_B \cdot P, M + s_B \cdot p_A)$. Bob then transmits this tuple to Alice. M does not have to be in the subgroup generated by P.*

4. Alice *now computes $s_A \cdot (s_B \cdot P) = s_B \cdot p_A$ and creates the inverse $-(s_B \cdot p_A)$. Then she can get the message M from Bob's transmission by adding $(M + s_B \cdot p_A) + (-s_B \cdot p_A) = M$.*

Note that in the ElGamal cryptosystem using modulo integers from $\mathbb{F}_q$, q a large prime, the operations are exponentiations (see [Sti95], 5.1), while here the operation is a multiplication of the point P with itself. And because the negative of a point $P = (P_x, P_y) \in E$ has been defined to be $-P = (P_x, -P_y)$ in definition 2, this inverse can be computed by just changing the sign of the real number P_y. But this change can be done without any arithmetical operations and hence without any cost.
This fact led to the idea of using subtractions and forming addition-subtraction chains to analyze their performance for groups within which the corresponding operation of the subtraction is free or only little more expensive than the corresponding operation of the addition.

Chapter 1

Addition chains

1.1 Introduction to addition chains

Addition chains are sequences of operations that lead from 1 to a certain target integer by adding two prior computed values. They serve as an important concept for the efficient computation of powers and are mostly examined in close connection to that context, a fashion, which this thesis will also follow. The current chapter will introduce the basic concepts of addition chains, limitations, generalizations and a brief overview over several existing algorithms that create addition chains.
The basic approach is to solve an exponentiation problem as specified in the following definition.

Definition 1.1 (exponentiation problem)
An exponentiation problem $\Pi = (x, e)$ consists of an exponent $e \in \mathbb{N}$ and a base x, which is an element of a given semigroup H. It poses the task to compute the e-th power of x, denoted by x^e.

An exponentiation problem Π offers a very general definition of the task to raise a number to a certain power. That is done to include the different fields of application, like the exponentiation in $\mathbb{R}$, in $\mathbb{F}_n = \mathbb{Z}/n\mathbb{Z}$ or the exponentiation on an elliptic curve, where the "exponentiation" is really the task to compute a multiple $e \cdot P$ of a given point P. In the later chapters, it will prove crucial that every exponentiation problem defines different costs for the operations that are performed during the process of the exponentiation (see definition 2.3 on page 54). These costs will help determining if and in which cases addition-subtraction chains should replace the concept of addition chains.
An exponentiation problem can be solved in many ways, the most trivial being to simply multiply x to itself $e-1$ times, leading to huge costs for large e. It gives the upper bound of $e-1$ multiplications for an exponentiation problem, but this bound can be improved significantly.

This thesis will examine the concept of addition chains and later of addition-subtraction chains as an approximation towards a lower bound for exponentiation problems. But first, addition chains need to be introduced.

Definition 1.2 (addition chain $\chi(e)$)
(following J. von zur Gathen and M. Nöcker in [GN00], §2)
An addition chain χ is a sequence

$$\chi = ((j(1), k(1)), \ldots, (j(r), k(r))),$$

of pairs of non-negative integers with

$$0 \leq k(i) \leq j(i) < i \qquad \forall\ 1 \leq i \leq r.$$

The number r of pairs is the length $L(\chi)$ of the addition chain χ.
The semantics of χ is defined to be the set $S(\chi) = \{a_0, a_1, \ldots, a_r\}$ of integers such that

$$\begin{aligned} a_0 &= 1 \\ a_i &= a_{j(i)} + a_{k(i)} \quad \forall\ 1 \leq i \leq r. \end{aligned}$$

The tupel $(j(i), k(i))$ is said to produce the element a_i of the semantics. We may assume that $1 = a_0 < a_1 < \ldots < a_r$.
In the case where $j(i) \neq k(i)$, the operation is called an addition, in the case where $j(i) = k(i)$, the operation is called a doubling.
For any set of numbers $E = \{e_0, e_1, \ldots, e_k\} \subseteq S(\chi)$, χ is called an addition chain for E (an addition chain for $e_0, \ldots, e_k$) and may be denoted as $\chi(e_0, e_1, \ldots, e_k)$. $\chi(e)$ denotes an addition chain for a single number $e \in S(\chi(e))$.

This thesis will only be concerned with addition chains for single elements $e \in \mathbb{N}$. In the literature, the semantics of $\chi(e)$ and the addition chain itself are usually identified. By separating both aspects, the operations used to compute the semantics can be differentiated better. As this thesis will be concerned with the different costs of the two operations addition and doubling, the given definition has been chosen. Unless explicitly noted, the elements of the semantics will always be noted in the same order as they are created by the addition chain, hence, the semantics will always depict the common notation.
Addition chains for a set E of two or more numbers are sometimes referred to as *addition sequences* ([DLS81], [BC90], [Gor98]).

Example 1.3:

Consider the exponentiation problem $\Pi = (x \in H,\ 191)$, and the following addition chain $\chi(191)$ for the exponent:

$$\begin{aligned} \chi(191) &= ((0,0),(1,0),(1,1),(3,2),(3,3),(5,4),(6,4),(7,7),(8,8),\\ &\quad (9,6),(10,9))\\ S(\chi(191)) &= \{1,2,3,4,7,8,15,22,44,88,103,191\}. \end{aligned}$$

It is $L(\chi(191)) = 11$ and $\chi(191)$ consists of 6 additions and 5 doublings. Note that the two values 4 and 8 have been constructed by the doublings $(1,1)$ and $(3,3)$, but they could also be constructed by additions $(2,0)$ and $(4,0)$.

This shows that addition chains can be constructed in different ways, an observation that will play an important role when the costs of additions and doublings are examined and are not equal, e.g. if a doubling can be done faster than an addition, doublings should be preferred.

With the given construction of $\chi(191)$, the exponentiation problem Π is solved using this addition chain by successive computation of the values

$x^2 = (x)^2$,	$x^3 = x \cdot x^2$,	$x^4 = (x^2)^2$,	$x^7 = x^3 \cdot x^4$,
$x^8 = (x^4)^2$,	$x^{15} = x^7 \cdot x^8$,	$x^{22} = x^7 \cdot x^{15}$,	$x^{44} = (x^{22})^2$,
$x^{88} = (x^{44})^2$,	$x^{103} = x^{15} \cdot x^{88}$,	$x^{191} = x^{88} \cdot x^{103}$.	

Example 1.3 shows how the concept of addition chains leads to a solution of an exponentiation problem. In general, all powers $x^{a_1}, x^{a_2}, \ldots, x^{a_{L(\chi(e))}} = x^e$ have to be computed successively. Because of exponentiation laws, every new x^{a_i} for $i > 0$ can be calculated by multiplying two preceding values, for example $x^{103} = x^{15} \cdot x^{88}$. It also implies that the addition chain needs additional space to save all values which are used in the following computation. However, the number of arithmetical operations can be reduced significantly compared with the trivial method mentioned above – example 1.3 only needs 11 operations compared to 190 needed by the trivial approach. Exact upper bounds will be established in chapter 2, where the two most popular addition chain algorithms will be examined. But first, let's examine the problem of differentiating the two arithmetical operations addition and doubling.

1.1.1 Exchanging additions and doublings

Most authors on this subject used not to differentiate between additions and doublings at all (see for example [Knu97], [Gor98], [BC90]). Their model allows the exchange of doublings by additions and, if applicable, additions by doublings, because the interest is mostly to count the total number of operations. With this model, it may always be assumed that doublings or

squarings (as the corresponding operation in the context of exponentiation) are always cheaper than additions, because every doubling could be written as an addition, hence, if they are cheaper, doublings would be applied, if they're more expensive, they're performed as additions. This is obviously not true for additions, because most additions add two different elements, which cannot be replaced by a doubling in general, for example all odd elements of the semantics cannot be created using doublings.
This property of the usual model is definitely a problem of the model, because it doesn't reflect the real world, in which squarings may of course be more expensive than additions (see example 2.4(2)) and the assumption that every doubling may easily be replaced by an addition is not true.
Consider x^2, which can be written as $x + x$, but it is still a doubling, as doublings are exactly those additions, where an element is added to itself. The example of elliptic curve arithmetic in definition 2 on page 7 has shown that the addition of the same point $P \neq \mathcal{O}$ to itself requires the computation of a tangent, while the addition of two different points $P \neq Q$ requires the computation of the line defined by both points. These operations are different and may require different costs. It is therefore crucial to count every operation right. The interchange of the two operations is not trivial. Consider for example for any even value k besides 2 an element $k \in S(\chi(e))$, $k = 2i$ for some i and some addition chain $\chi(e)$. k can always be computed by doubling i or by an addition, for example of $1 + (k - 1)$. But in order to be able to choose between the two possibilities freely, i and the addend $k-1$ need to be present in the semantics (1 is always contained). Only in this one case both ways to compute k are possible. Thus exchanging a doubling by an addition may require the rebuilding of the addition chain and may even require a longer chain. Therefore, exchanging should not be considered as a general option.
In order to cope with this problem, the addition chain model in this thesis differentiates additions and doublings sharply as defined in definition 1.2, providing us with a more realistic model. The idea to differentiate the operations isn't new, for example in [EgK90], doublings and additions are counted separately for the binary and the m-ary method, in [GN97], doublings, additions and q-steps are counted separately for q-addition chains (see definition 1.11). Within this thesis, the substitution of doublings by additions will only be allowed if it can be shown that this still results in a valid addition chain according to the definition.
In the operation sensitive analyses, it has to be ensured that no operation is counted wrong. This is impossible for doublings, as they are defined as the addition of two equal elements, which cannot be mistaken as a normal addition, but for additions, it has to be carefully examined that all additions add two different elements.
The latter can be assured in different ways. If in an addition chain tuple $(j(i), k(i))$ of some addition chain $\chi(e)$, $j(i) \neq k(i)$, but the corresponding

elements of the semantics are equal, e.g. $a_{j(i)} = a_{k(i)}$, a *hidden doubling* would occur, counted as an addition. But in this case, not every step of the addition chain would have created a new element, e.g. step $j(i)$ and step $k(i)$ would have created the same element in the semantics, hence, the semantics would contain less elements than possible. This means that if $\#S(\chi(e)) = L(\chi(e)) + 1$, there is no hidden doubling possible (note that the $+1$ originates in the fact that the semantics always contains the start value 1).
Another way of assuring no hidden doubling is to show that the values a_i created by the addition chain $\chi(e)$ are strictly monotonously increasing. This can easily be verified for a special class of steps within an addition chain, which are the most common steps in most of the presented addition chain algorithms within this thesis.

Definition 1.4 (star step)
(following J. von zur Gathen and M. Nöcker in [GN97], §2.1)
If for a step $(j(i), k(i))$ of an addition chain χ, $j(i) \neq k(i)$ and $j(i) = i-1$ or $k(i) = i-1$, then step i is called a star step. Note that only addition steps may form a star step.

Some sources define star steps differently, including doublings of the form $(j(i), k(i)) = (i-1, i-1)$ (see [Knu97], p.467 or [BBB94]).

Lemma 1.5 (star steps prevent hidden doublings)
If an addition chain $\chi(e)$ only contains star steps and doublings of the form $(j(i), k(i)) = (i-1, i-1)$, no hidden doublings occur.

Proof:

> The first step in any addition chain is bound to be $(0, 0)$, which is such a special doubling and it creates the element 2, which will be referenced by the next step. That element 2 is the highest element within the semantics at that time. Inductively, the assumption that the first k steps within $\chi(e)$ always reference the highest element in the semantics shows that also step $k+1$ must reference the highest element a_{i-1} in the semantics at that time and by adding to or doubling a_{i-1}, it again creates a new highest element. Hence, the sequence $a_0, a_1, \ldots, a_{L(\chi(e))}$ is strictly monotonously increasing, which proves the claim. $\square$

1.1.2 Some definitions and remarks on the notation

The examination of the concept of addition chains and especially of addition chain algorithms creates the need to introduce some definitions and notations that will be widely used within this thesis. One of the basic approaches in the creation of addition chains is the use of the b-adic expansion of a number e. For these cases, the following notation will be used.

Definition 1.6 (b-adic expansion)
For a given number $n \in \mathbb{N}$, the b-adic expansion is a sequence $(n)_b := (n_{\lambda_b(n)-1}, n_{\lambda_b(n)-2}, \ldots, n_1, n_0)$ with $n_i \in \{0, 1, \ldots, b-1\}$ such that

$$n = \sum_{i=0}^{\lambda_b(n)-1} n_i \cdot b^i.$$

For such a b-adic expansion, the length of $(n)_b$, which is equal to the number of digits used, is denoted by $\lambda_b(n)$. Note that

$$\lambda_b(n) \quad = \quad \lfloor \log_b(n) \rfloor + 1.$$

It is assumed that for the highest digit (called the most significant bit in the case where $b = 2$) it is $n_{\lambda_b(n)-1} \neq 0$, allowing $(n)_b$ to be uniquely defined. If $b = 2$, the notation can be abbreviated by using $\lambda(n) := \lambda_2(n)$.
In most algorithms, a 2^d-adic expansion is used, $d \in \mathbb{N}$, where d consecutive bits are combined to one digit.

The used notation of Gaussian brackets may not be familiar to all readers, it is clarified in the following definition.

Definition 1.7 (Gaussian brackets)
Let the highest integer n lower or equal to a real number $x \in \mathbb{R}$ be denoted by

$$\lfloor x \rfloor \quad := \quad n,$$

and let the lowest integer m higher or equal to x be denoted by

$$\lceil x \rceil \quad := \quad m.$$

If $x \in \mathbb{Z}$, it is $\lfloor x \rfloor = \lceil x \rceil$.

Definition 1.8 (Hamming weight $\boldsymbol{\nu_b(n)}$)
The number of nonzero digits in the b-adic expansion $(n)_b$ of a given number $n \in \mathbb{Z}$ to a base $b \in \mathbb{N}$ is called the Hamming weight of n with respect to base b and it is denoted by $\nu_b(n)$.

1.1.3 Limits and generalizations of addition chains

While each algorithm using addition chains to solve an exponentiation problem Π establishes an upper bound on the number of arithmetical operations needed to compute x^e, also a lower bound can be given. In [Sch75], A. Schönhage proved the following lower bound for the length of addition chains, which is also a lower bound on the number of arithmetic operations.

Definition and theorem 1.9 ($l(e)$ and a lower bound for $l(e)$)
(following A. Schönhage in [Sch75])
Let $l(e)$ denote the shortest possible length for an addition chain for the number $e \in \mathbb{N}$, then the length $l(e) = L(\chi(e))$ of such a shortest or optimal addition chain for e has the following lower bound:

$$\log_2 e + \log_2 \nu_2(e) - 2.13 \;\leq\; l(e), \tag{1.1}$$

where $\nu_2(e)$ denotes the binary Hamming-weight of e.

Proof: The proof can be found in [Sch75].

In figure 1.1, this lower bound is plotted out against the computed exact values for $l(n)$ for some range of n to allow an impression about the tightness of this lower bound.

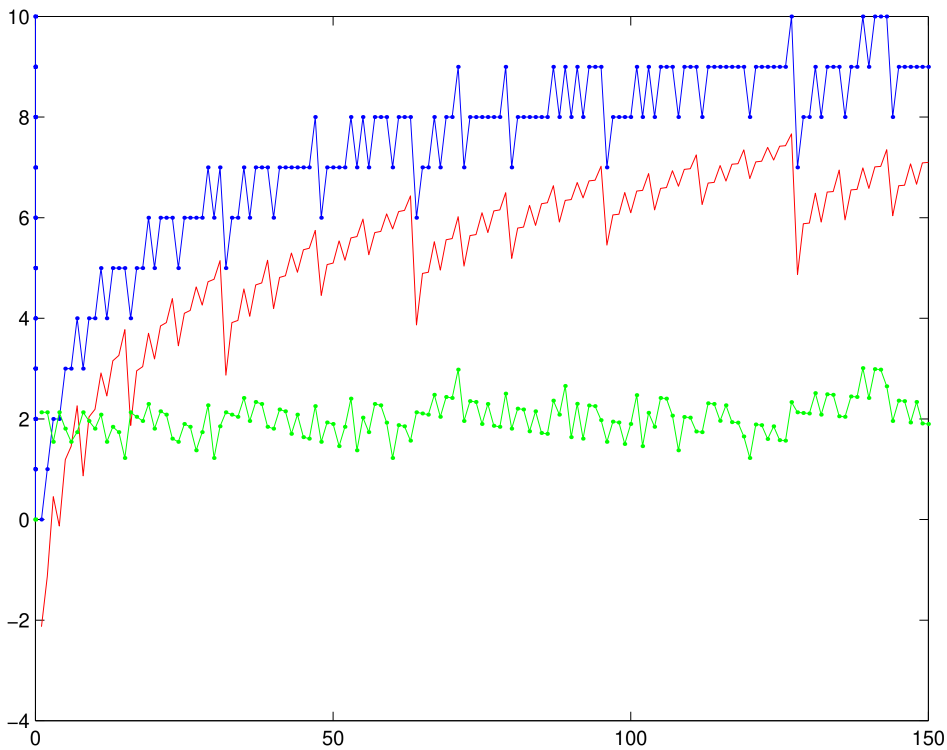

Figure 1.1: Comparison between $l(n)$ (blue) and the lower bound (1.1) by Schönhage (red). The green line shows the difference.

Obviously, an addition chain is not unique. The trivial approach to solve an exponentiation problem Π, which just multiplies x with itself $e-1$ times, also leads to the trivial addition chain $\chi(e) = ((0,0),(1,0),(2,0),(3,0),\ldots,(e-$

1, 0)) with the semantics $S(\chi(e)) = \{1, 2, 3, 4, \ldots, e\}$. Hence, for the same integer e, there are usually many different addition chains with many different lengths. It is most efficient to use an addition chain with the shortest possible length $l(e)$. Again, such an "optimal" addition chain is not unique as the following example shows.

Example:

Let $e = 9$, then there exist 3 different shortest addition chains of the length $l(9) = 5$ with the following semantics:
{1, 2, 3, 6, 9}
{1, 2, 4, 5, 9}
{1, 2, 4, 8, 9}

The observation of the sequence $(l(e))_{e\in\mathbb{N}}$ reveals a sequence which seems to be chaotic (see [Thu93]). A selection of Edward G. Thurber's examinations of this sequence, where he introduced the term $NMC(e)$ for the number of minimal chains, can be seen in table 1.1.

e	$NMC(e)$	e	$NMC(e)$
1	1	2466	1042
2	1	2467	2
3	1	2468	1126
8	1	2539	3289
9	3	2540	230110
10	4	2541	6
11	15		

Table 1.1: Some numbers of different minimal chains (taken from [Thu93])

The concept of addition chains can be generalized in different ways, all of which provide valuable facts and suggestions for notation for ordinary addition chains. The following two definitions will present two different approaches of possible generalizations of the concept of addition chains. A third concept, the generalization of addition chains to addition-subtraction chains will be the major point of interest of this thesis and examined in great detail in the last chapters.

One approach to generalize addition chains are *vectorial addition chains* (or vector addition chains, see [Oli81]). If a set of k values should be computed in a single addition chain, but the values of the addition chains for each computed target value should not be mixed, the addition chains can be computed in parallel using vectors from $\mathbb{N}^k$ to represent the set of computed addition chains. Ordinary addition chains are then vectorial addition chains with $k = 1$. Vectorial addition chains led to new approaches in the creation of ordinary addition chains, like the development of the continued fractions method presented in section 1.2.5. The following definition has been transformed to the notation used for ordinary addition chains.

Definition 1.10 (vectorial addition chain $\chi([e_1, e_2, \dots, e_k])$)
(adapting the definition by J. Olivos in [Oli81])
A vectorial addition chain χ is a sequence

$$\chi = ((j(1), k(1)), \dots, (j(r), k(r))),$$

of pairs of integers with

$$0 \le k(i) \le j(i) < i \qquad \forall\, 1 \le i \le r.$$

The number r of pairs is the length $L(\chi)$ of the vectorial addition chain χ. The semantics of χ is defined to be the set $S(\chi) = \{\vec{a}_{-k+1}, \vec{a}_{-k+2}, \dots, \vec{a}_0, \vec{a}_1, \dots, \vec{a}_r\}$ of vectors such that

$$\begin{aligned} \vec{a}_{-k+i} &= \vec{b}_i && \forall\, 1 \le i \le k \\ \vec{a}_i &= \vec{a}_{j(i)} + \vec{a}_{k(i)} && \forall\, 1 \le i \le r, \end{aligned}$$

where $\vec{b}_1, \vec{b}_2, \dots, \vec{b}_k$ denote the canonical basis of $\mathbb{R}^k$, e.g. the i-th entry of $\vec{b}_i$ is 1 and all others are zero.
For any set of elements $E = \{\vec{e}_0, \vec{e}_1, \dots, \vec{e}_k\} \subseteq S(\chi)$, χ is called a vectorial addition chain for E (a vectorial addition chain for $\vec{e}_0, \dots, \vec{e}_k$) and may be denoted as $\chi(\vec{e}_0, \vec{e}_1, \dots, \vec{e}_k)$. $\chi(\vec{e})$ denotes a vectorial addition chain for a single element $\vec{e} \in S(\chi(\vec{e}))$, $\vec{e} = [e_1, e_2, \dots, e_k]$.

Example:

Consider the given vector $\vec{e} = [5, 25, 26]$. Then the following sequence is a vectorial addition chain for $\vec{e}$ of length 9:

$$\begin{array}{lllllll} \chi(\vec{e}) = & (& (0,-1), & (1,-2), & (1,1), & (3,3), & (4,2), \\ & & (5,5), & (6,5), & (7,0), & (8,6) &) \end{array}$$

$$\begin{array}{lllllll} S(\chi(\vec{e})) = \{ & [1, 0, 0], & [0, 1, 0], & [0, 0, 1], & & \\ & [0, 1, 1], & [1, 1, 1], & [0, 2, 2], & [0, 4, 4], & [1, 5, 5], \\ & [2, 10, 10], & [3, 15, 15], & [3, 15, 16], & [5, 25, 26] & \} \end{array}$$

The second approach to generalize addition chains concentrates on the operations and not on the type of values to be computed. Ordinary addition chains offer the addition and the doubling, but in some environments it is useful to generalize the doubling, which is a scalar multiplication with 2, to a general scalar multiplication. The resulting chains are called *q-addition chains* or weighted addition chains with scalars (following [Nöc01], p.17). They are motivated by the fact that in finite fields $\mathbb{F}_{q^n}$, q prime, the Frobenius automorphism can be computed very fast. The use of the Frobenius automorphism will be introduced briefly in section 1.2.10. Although q-addition chains are only used within this thesis for the special case $q = 2$ (ordinary addition chains), the concept of q-addition chains motivates the notation used for denoting numbers of doublings/squarings and q-steps/exponentiations with q as defined in definitions 2.1 and 2.2.

Definition 1.11 (q-addition chains $\chi(e, q)$)
(following M. Nöcker in [Nöc01], Def. 3.6)
An addition chain with scalar q, or q-addition chain for short, χ is a sequence of pairs of integers $(j(1), k(1)), \ldots, (j(r), k(r))$ such that $0 \leq k(i) \leq j(i) < i$, or $k(i) = -q$ and $0 \leq j(i) < i$ for $1 \leq i \leq r$. The semantics $S(\chi) = \{a_0, a_1, \ldots, a_r\}$ is defined by

$$\begin{aligned} a_0 &= 1 && \textit{and} \\ a_i &= a_{j(i)} + a_{k(i)} && \textit{if } k(i) \neq -q, \textit{ and} \\ a_i &= q \cdot q_{j(i)} && \textit{if } k(i) = -q. \end{aligned}$$

A pair $(j(i), -q)$ is called a q-step and a pair $(j(i), k(i))$ with $k(i) \neq -q$ a non-q-step.
$L(\chi) = r$ denotes the length of the q-addition chain. For a given number $e \in \mathbb{N}$ (a given set of numbers $E = \{e_0, e_1, \ldots, e_k\} \subset \mathbb{N}$), a q-addition chain for e (E) may be denoted as $\chi(e, q)$ ($\chi(E, q) = \chi(e_0, e_1, \ldots, e_k, q)$).

Note that for $q = 2$, the notation $(j(i), -2)$ in the given definition is equal to the notation $(j(i), j(i))$ defining a doubling step in definition 1.2.
Example:

Consider the exponentiation problem $\Pi = (x \in H, 219)$ and a given scalar $q = 4$. Assume the 4-ary expansion of 219 to be given as $(219)_4 = (3, 1, 2, 3)$. Then a q-addition chain $\chi(219, 4)$ can be formed as

$$\begin{aligned} \chi(219, 4) &= ((0,0), (1,0), (2,-4), (3,0), (4,-4), (5,1), (6,-4), (7,2)) \\ S(\chi(219, 4)) &= \{1, 2, 3, 12, 13, 52, 54, 216, 219\} \end{aligned}$$

with $12 = 4 \cdot 3$, $52 = 4 \cdot 13$ and $216 = 4 \cdot 54$ being created using q-steps (and additions for the remaining elements). For further details on this example refer to example 1.13.

These generalizations ease the achievement of some results about addition chains. First, the limits of the concept of addition chains can be analyzed by determining the complexity class of the problem of finding a shortest addition chain.
The fact that there are sometimes so many different shortest addition chains for a given number e seems to ease the search for such a shortest addition chain. Unfortunately, the very similar problem of the search for a shortest addition chain for several e_i in one chain has been proved to be NP-complete

(see [DLS81]). Although the result has not yet been verified for addition chains for one value e, this problem is commonly also expected to be NP-complete, making it impossible to compute such a shortest addition chain in polynomial time (under the widely accepted assumption that $P \neq NP$). Therefore it is necessary to develop approximation algorithms, which at least produce optimal results for a satisfying number of inputs or which output addition chains, whose length is close to $l(e)$.

1.2 Algorithms for addition chains

This section provides a brief overview of the existing addition chain algorithms solving exponentiation problems. It is intended to introduce concepts rather than analyses and to show generalization lines in the development of such algorithms. Bibliographical references will be given to suggest further reading. Some of the presented algorithms will be analyzed in detail in the second chapter and will be used to introduce and explore the concept of addition-subtraction chains later. Most of the methods introduced have originally been described with a different definition of addition chains, not differentiating between the addition chain and its semantics. These methods have been modified in their notation for comparison according to the definition used within this thesis.
The following algorithms are the most commonly known addition chain algorithms:

- The binary method (also known as repeated squaring)
- The m-ary method / Brauer's method
- Sliding window methods
- The factor method
- The continued fraction method / Euclid's method
- Knuth's powertree method
- The BGMW-algorithm
- The data compression method by Yacobi
- The data compression method by Bocharova and Kudryashov
- Other Approaches (Frobenius-automorphism, normal bases)

1.2.1 The binary method (repeated squaring)

The binary method is the oldest known efficient algorithm for the computation of powers, it has been described as early as 200 B.C. in Piṅgala's *Chandah-sûtra*-writings in classical Indian mathematics, the classical Arab mathematician al-Uqlīdisī of Damascus described the method in AD 952 for $x = 2$. See [Knu97], or directly [Dat35] p. 76, [Sai75] pp. 341-342 und [Sac79] pp. 132-136 for further details on the history of this method.

idea: The binary method is based on the binary expansion of the exponent e. It creates x^e by starting with x and squaring ("S") the accumulated result for every binary digit and multiplying ("M") it with the base x for every nonzero digit.

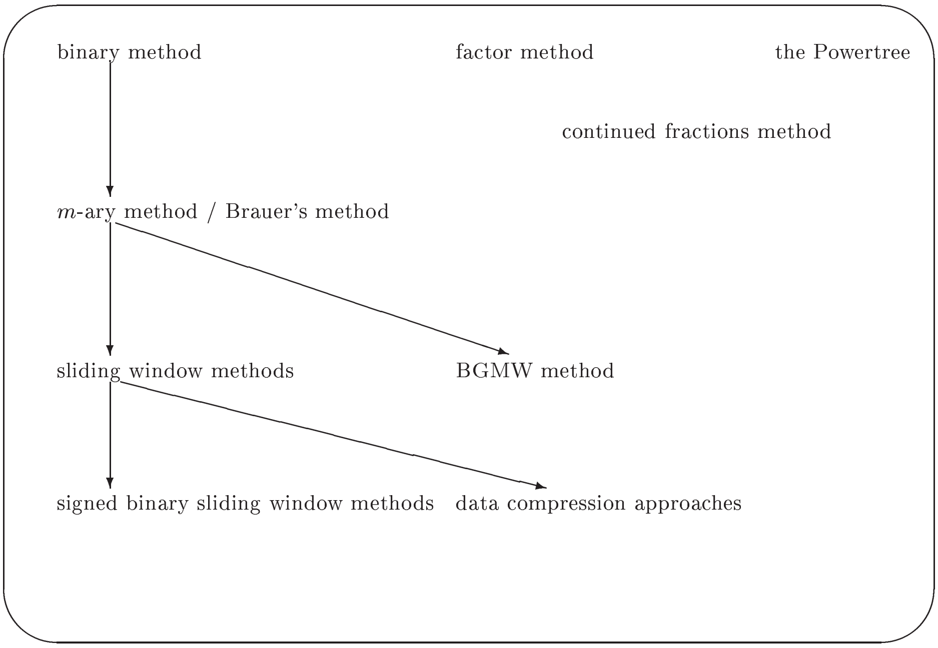

Figure 1.2: Evolution of addition chain methods

Example 1.12:

(1) Consider the exponentiation problem $\Pi = (x \in H, 219)$ to demonstrate the binary method:

$$\begin{array}{rcl} (219)_2 & = & \begin{array}{cccccccccc} (& 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 &) \\ & \downarrow & \searrow\downarrow & \searrow & \searrow\downarrow & \searrow\downarrow & \searrow & \searrow\downarrow & \searrow\downarrow & \\ & \mathrm{x} & \mathrm{SM} & \mathrm{S} & \mathrm{SM} & \mathrm{SM} & \mathrm{S} & \mathrm{SM} & \mathrm{SM} & \end{array} \\ x^{219} & = & \\ & = & (((((((x^2)\cdot x)^2)^2 \cdot x)^2 \cdot x)^2)^2 \cdot x)^2 \cdot x \\ \chi(219) & = & ((0,0),(1,0),(2,2),(3,3),(4,0),(5,5),(6,0),(7,7), \\ & & (8,8),(9,0),(10,10),(11,0)) \\ S(\chi(219)) & = & \{1,2,3,6,12,13,26,27,54,108,109,218,219\} \\ L(\chi(219)) & = & 12 \end{array}$$

(2) The smallest counterexample to the assumption that this method is optimal is $e = 15$, $(15)_2 = 1111$:
binary method computes $\{1,2,3,6,7,14,15\}$ - 6 Operations
an optimal chain computes $\{1,2,3,5,10,15\}$ - 5 Operations
(there is only one optimal addition chain of length 5 for $e = 15$)

The binary method is based on the binary expansion of the exponent e and squares the accumulated result for every binary digit and multiplies it with the base x for every nonzero digit. The method has been regarded as optimal for a long time ([Knu97], p.463), which it isn't. Example 1.12(2) shows the smallest counterexample $e = 15$. Although, as doublings are the fastest way to increase the value of the numbers, it is optimal for all powers of 2.

The binary method requires $\lambda(e) + \nu(e) - 2$ arithmetical operations, where $\lambda(e)$ denotes the length of the binary expansion and $\nu(e)$ denotes the binary Hamming-weight of e. A detailed analysis of this method will be shown in section 2.2.

1.2.2 The m-ary method / Brauer's method

The m-ary method (see [Knu97], [Bra39]) is a generalization of the binary method. It allows the exponent e to be in any m-adic expansion. Brauer's method is a special case of the general m-ary method, where special bases of the form $m = 2^d$ are used to combine d digits of the binary expansion to create one digit of the 2^d-ary expansion.

idea: The m-ary method is based on the m-adic expansion $(e)_m$ of the exponent e. It creates x^e by starting with x and raising the accumulated result to its m-th power ("$\mathrm{S^m}$") for every m-adic digit and multiplying ("$\mathrm{M_i}$") it with the value of the current m-adic digit for every nonzero digit.

The method requires the precomputation of the values of the m-adic digits $x^2, x^3, \ldots, x^{m-1}$ and a strategy to raise the accumulated result to its m-th power (using the binary method, the m-ary method recursively or an optimal addition chain if m is small enough).

As the m-adic expansion is shorter than the binary expansion, its Hamming-weight is smaller on average, resulting in less multiplications. However, the additional need to precompute certain values arises.

Example 1.13:

Consider the same exponentiation problem $\Pi = (x \in H, 219)$ as in example 1.12(1) of the binary method and the window length $d = 2$, e.g. $m = 2^2 = 4$, for the m-ary method. The exponentiations with 4 will be performed using two squarings, because this is optimal according to the last section.

$$
\begin{array}{rl}
(219)_2 = & (\ \underline{1\,1}\quad \underline{0\,1}\quad \underline{1\,0}\quad \underline{1\,1}\ ) \\
(219)_4 = & (\ 3\quad 1\quad 2\quad 3\ ) \\
& \downarrow\quad \searrow\downarrow\quad \searrow\downarrow\quad \searrow\downarrow \\
x^{219} = & \mathsf{x}^3\quad \mathsf{S}^2\mathsf{M}_1\quad \mathsf{S}^2\mathsf{M}_2\quad \mathsf{S}^2\mathsf{M}_3 \\
x^{219} = & x^2,\ x^3,\ (((\ x^3\ )^4 \cdot\ x\ )^4 \cdot\ x^2\ )^4 \cdot x^3 \\
\chi(219) = & ((0,0),\ (1,0),\quad (2,2),\ (3,3),\ (4,0),\ (5,5),\ (6,6),\ (7,1), \\
& (8,8),\ (9,9),\ (10,2)) \\
S(\chi(219)) = & (1,\ 2,\ 3,\quad 6,\ 12,\ 13,\ 26,\ 52,\ 54,\ 108,\ 216,\ 219) \\
L(\chi(219)) = & 11
\end{array}
$$

It can be seen that this method needs one operation less for $e = 219$ than the binary method (see example 1.12(1)). The elements before the gap belong to the precomputation step.

It is easy to see that the binary method is a special case of the m-ary method with $m = 2$. In most practical applications, Brauer's method ($m = 2^d$) is used and an optimal value for d can be determined. This will be examined in detail in section 2.3. The Brauer method requires

$$2^d - 2 + \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1 \right) \cdot \left(\frac{2^d - 1}{2^d} + d \right)$$

arithmetical operations on average (see section 2.3.5).

1.2.3 Sliding window methods

Sliding window methods are generalizations of Brauer's method, the m-ary method for $m = 2^d$. They have been introduced by J. Bos and M. Coster in [BC90] and have been modified and analyzed by a number of authors, see for example [Koç95], [PPC99] or [Gor98].

idea: A close look at Brauer's method shows that the windows are created in the same manner for every binary expansion, possible special properties of the binary expansion are not exploited. The sliding window methods determine windows in a flexible way based on the binary expansion and aim to reduce the average number of operations by taking advantage of the pattern of the binary expansion.

While the length of the binary expansion always accounts for a fixed number of squarings, the number of multiplications is caused by the number of nonzero windows, which this method tries to reduce. Therefore, sliding window methods allow zero windows to be of any length, while only nonzero windows are restricted to a certain (maximal) length.

The nonzero windows are determined according to the binary expansion. The current nonzero window slides across the binary expansion from one side to the other and initiates a new nonzero window whenever a nonzero digit occurs.

There are basically two different sliding window methods known, the constant length nonzero window method, that requires nonzero windows to be exactly of a certain length, and the variable length nonzero window method, that requires nonzero windows to be equal to or less than a certain length.

Example 1.14:

(1) Consider the exponentiation problem $\Pi = (x \in H, 219)$, that has also been used as an example for both the binary and the m-ary method. Let the window length be $d = 2$ and let the nonzero window length be required to consist of exactly d bits.

$$
\begin{array}{rl}
(219)_2 = & (\ \underline{1\,1}\quad \underline{0}\quad \underline{1\,1}\quad \underline{0}\quad \underline{1\,1}\ ) \\
\text{values}: & \quad 3 \qquad 0 \qquad 3 \qquad 0 \qquad 3 \\
& \quad \downarrow \quad \searrow \quad \searrow\downarrow \quad \searrow \quad \searrow\downarrow \\
x^{219} = & \quad \mathsf{x}^3 \quad \mathsf{S}^1 \quad \mathsf{S}^2\mathsf{M}_3 \quad \mathsf{S}^1 \quad \mathsf{S}^2\mathsf{M}_3 \\
x^{219} = & x^2,\ x^3,\ ((((\ x^3\ )^2\ )^4 \cdot\ x^3\ )^2\ )^4 \cdot\ x^3 \\
\chi(219) = & ((0,0),\ (1,0),\quad (2,2),\ (3,3),\ (4,4),\ (5,2),\ (6,6),\ (7,7), \\
& (8,8),\ (9,2)) \\
S(\chi(219)) = & \{1, 2, 3,\quad 6, 12, 24, 27, 54, 108, 216, 219\} \\
L(\chi(219)) = & 10
\end{array}
$$

This example shows that for this example, the constant length nonzero window method with $d = 2$ needs one operation less than the m-ary method with $d = 2$ (see example 1.13).

(2) Another example compares the use of both kinds of sliding window methods for a longer binary expansion.

Let $(e)_2$ = 1010000011001001101000001100010010100011101 (e = 5,524,620,191,005), then with a constant length nonzero window method with $d = 5$ digits, the binary expansion would be partitioned into the following windows (nonzero windows framed):

$$\boxed{101}\,00000\,\boxed{11001}\,0\,\boxed{01101}\,0\,\boxed{00001}\boxed{10001}\boxed{00101}\,000\,\boxed{11101}$$

Now assume a variable length nonzero window method is used, which stops nonzero windows whenever runs of $\delta = 2$ or more zeros occur. The binary expansion would then be partitioned as follows (nonzero windows framed):

$\boxed{101}\,00000\,\boxed{11}\,00\,\boxed{1}\,0\,\boxed{01101}\,00000\,\boxed{11}\,000\,\boxed{1}\,00\,\boxed{101}\,000\,\boxed{11101}$

While the constant length nonzero window method creates 7 nonzero windows on this input, the variable length nonzero window method creates 8. This is not something extraordinary, because analyses show the constant length nonzero sliding window method to be better on average (see for example [Koç95] and [PPC99]).

The cost analyses of both variants of the sliding window method can be found in [PPC99]. The methods require

$$2^{d-1} + \lambda_2(e) - 1 + NWIN - WLEN$$

arithmetical operations, with $NWIN$ denoting the average number of nonzero windows and $WLEN$ denoting the length of the leftmost window (following the notation in [PPC99]). The values for both $NWIN$ and $WLEN$ can be computed using Markov-chains. The analyses result in summation formulas, which can be found in [PPC99]. The results show that the m-ary method is inferior to this approach on average.

1.2.4 The factor method

Another method based on a different idea than the methods mentioned before is the factor method (see [Knu97]). It uses the prime factorization of the exponent to compute an addition chain.

idea: If e is not prime, it is $e = p \cdot q$ for some prime p and x^e can be computed as $x^e = (x^p)^q$. If $e > 3$ is prime, the same approach can be utilized computing $x^e = x^{e-1} \cdot x$. This basic approach can be used recursively for p and q until both are 1 or 2.

If the prime factorization of e is given, e.g.

$$e = \prod_{i=1}^{r} \rho_i^{\kappa_i},$$

then x^e can be computed by subsequently finding an addition chain for each ρ_i using the basic approach mentioned above and then repeating that chain κ_i times (starting with the result of the last chain for the usual starting value 1), before exponentiating according to the next ρ. This will lead to a computation of x^e.

For the sake of a good computational behaviour (to let accumulating factors be as small as possible), assume that the prime factors are sorted ascending by value.

Example 1.15:

(1) Consider the example $\Pi = (x \in H, 219)$ used as a standard example before. The addition chain $\chi(219)$ is created during the process and χ' is used to denote the accumulating addition chain. The prime factorization of 219 is

$$219 = 3 \cdot 73 \qquad \Rightarrow x^{219} = (x^3)^{73}$$

Compute $y_1 := x^3$ first: (resulting chain)

$$y_1 = x^3 = x^2 \cdot x \qquad \Rightarrow \chi' = ((0,0),(1,0))$$

Compute $z := y_1^{73}$ next:

$$73 = 72 + 1 \qquad \Rightarrow z = y_1^{73} = y_1^{72} \cdot y_1$$

Compute $z_1 = y_1^{72}$ using the factor method recursively:

$$72 = 2^3 \cdot 3^2 \qquad \Rightarrow y_1^{72} = ((((y_1^2)^2)^2)^3)^3$$

5 recursive steps are necessary: (resulting chains)

$$\begin{aligned}
y_2 &= y_1^2 & &\Rightarrow \chi' = ((0,0),(1,0),(2,2)) \\
y_3 &= y_2^2 & &\Rightarrow \chi' = (\ldots (2,2),(3,3)) \\
y_4 &= y_3^2 & &\Rightarrow \chi' = (\ldots (3,3),(4,4)) \\
y_5 &= y_4^3 &= y_4^2 \cdot y_4 &\Rightarrow \chi' = (\ldots,(4,4),(5,5),(6,5)) \\
y_6 &= y_5^3 &= y_5^2 \cdot y_5 &\Rightarrow \chi' = (\ldots (6,5),(7,7),(8,7))
\end{aligned}$$

That results at the end of the recursion for 72 to:

$$\begin{aligned}
x^{219} = y_1^{72} \cdot y_1 = y_6 \cdot y_1 \Rightarrow \chi' = &((0,0),(1,0),(2,2),(3,3),\\
&(4,4),(5,5),(6,5),(7,7),\\
&(8,7),(9,2))
\end{aligned}$$

And therefore

$$\begin{aligned}
\chi(219) &= ((0,0),\ (1,0),\ (2,2),\ (3,3),\ (4,4),\ (5,5),\ (6,5),\ (7,7),\\
&\quad (8,7),\ (9,2)) \\
S(\chi(219)) &= \{1,2,3,6,12,24,48,72,144,216,219\} \\
L(\chi(219)) &= 10 \text{ (which is the optimal result for } e = 219)
\end{aligned}$$

(2) Now consider the exponentiation problem $\Pi = (x \in H, 33)$. The factor method for $e = 33$ works as follows:

$$33 = 3 \cdot 11 \qquad \Rightarrow x^{33} = (x^3)^{11}$$

Compute $y_1 := x^3$ first: (resulting chain)

$$y_1 = \quad x^3 = \quad x^2 \cdot x \qquad \Rightarrow \chi' = ((0,0),(1,0))$$

Compute $z := y_1^{11}$ next:

$$11 = \quad 10 + 1 \qquad \Rightarrow z = y_1^{11} = y_1^{10} \cdot y_1$$

Compute $z_1 = y_1^{10}$ using the factor method recursively:

$$10 = \quad 2 \cdot 5 \qquad \Rightarrow y_1^{10} = (y_1^2)^5$$

2 recursive steps are necessary: (resulting chains)

$$y_2 = \quad y_1^2 \qquad \Rightarrow \chi' = ((0,0),(1,0),(2,2))$$

$$y_3 = \quad y_2^5 = (y_2^2)^2 \cdot y_2 \quad \Rightarrow \chi' = (\ldots(2,2),(3,3),(4,4),(5,3))$$

That results at the end of the recursion for 10 to:

$$x^{33} = \quad y_1^{10} \cdot y_1 = y_3 \cdot y_1 \qquad \Rightarrow \chi' = (\ldots(5,3),(6,2))$$

And therefore

$$\chi(33) = \quad ((0,0),\ (1,0),\ (2,2),\ (3,3),\ (4,4),\ (5,3),\ (6,2))$$

$$S(\chi(33)) = \quad \{1,2,3,6,12,24,30,33\}$$

$$L(\chi(33)) = \quad 7$$

This second example shows that the factor method is also not optimal, because the binary method creates an addition chain $\chi(33)$ computing $S(\chi(33)) = \{1,2,4,8,16,32,33\}$ with $L(\chi(33)) = 6$. It is the smallest example for a number, where the binary method excels the factor method (see [Knu97], p. 463).

The factor method obviously has one great disadvantage, that prevents its use in practical applications: It requires the prime factorization of the exponent e and that of intermediate values like $e-1$ to be known in advance. But as the computation of the prime factorization of an integer is still considered a good candidate for the complexity class NP, there is no known algorithm to compute the prime factorization of a given integer in polynomial time yet (and it may in fact be impossible if the assumed complexity proves right). Whole cryptographic systems (like RSA) depend on this assumption. Therefore, the factor method has not been widely used in practical applications, which use exponents e of 200 and more binary digits, making it impossible at the moment to compute the prime factorization at all.

1.2.5 The continued fractions method / Euclid's method

The idea to use continued fractions to create addition chains was presented in [BBBD89] and evolved in [BCHM95]. The authors developed this method while looking for an efficient way to compute monomials of the form $x^a y^b$

for some $a, b \in \mathbb{N}$ and it is based on vectorial addition chains (see definition 1.10).

idea: Given a vectorial addition chain for a vector $\vec{e}$ that contains an element $e := \vec{e}_i$, for which an addition chain is seeked, the projection of the i-th component of the vector addition chain produces an addition chain for e, if all zeros and repetitions are deleted. Recall for example the vectorial addition chain from example 1.1.3:

$$\begin{array}{rllll} \chi(\vec{e}) = & (\text{(-1,-2)}, & \text{(1,0)}, & \text{(1,1)}, & \text{(3,3)}, \\ & \text{(4,2)}, & \text{(5,5)}, & \text{(6,5)}, & \text{(7,-2)}, \\ & \text{(8,6)}) & & & \\ S(\chi([5,25,26])) = & \{ [0,0,1], & [0,\boxed{1},0], & [1,0,0], & [0,1,1], \\ & [1,1,1], & [0,\boxed{2},2], & [0,\boxed{4},4], & [1,\boxed{5},5], \\ & [2,\boxed{10},10], & [3,\boxed{15},15], & [3,15,16], & [5,\boxed{25},26]\} \end{array}$$

It contains with the highlighted elements the semantics of an addition chain for $e = 25$. The derived addition chain is

$$\begin{array}{rl} \chi(25) = & ((0,0), \quad (1,1), \quad (2,0), \quad (3,3), \quad (4,3), \quad (5,4)) \\ S(\chi(25)) = & \{1, 2, 4, 5, 10, 15, 25\} \\ L(\chi(25)) = & 6 \quad \text{(which is optimal)} \end{array}$$

The use of continued fractions of the form $\frac{a}{b} = [u_1, u_2, \ldots, u_k]$ denoting

$$\frac{a}{b} = u_1 + \cfrac{1}{u_2 + \cfrac{1}{u_3 + \cfrac{1}{\ddots + \cfrac{1}{u_k}}}}$$

now leads to an addition chain for (a, b) using the Euclidean Algorithm (see [GG99], §4.6) to compute the sequence $[u_1, u_2, \ldots, u_k]$, which is

$$\begin{array}{rcll} a & = & u_1 \cdot b + r_1 & \text{with } r_1 < b \\ b & = & u_2 \cdot r_1 + r_2 & \text{with } r_2 < r_1 \\ & \vdots & & \\ r_{k-2} & = & u_k \cdot r_{k-1} & \end{array}$$

With these values known, the following two definitions of concatenation operations for addition chains can be used. The definition has been modified in its notation to adapt to the enhanced definition of addition chains:

Definition 1.16 (concatenation of addition chains)
(original definition by Bergeron, Berstel and Brlek in [BBB94])
Let $\chi_1(a) = ((j(1),k(1)),(j(2),k(2)),\ldots,(j(m),k(m)))$ *and* $\chi_2(b) = ((s(1),t(1)),(s(2),t(2)),\ldots,(s(n),t(n)))$ *be two addition chains for* a *and* b *respectively and let* $c \in \mathbb{N}$. *Then the two operators* $\otimes$ *and* $\oplus$ *act on* χ_1, χ_2 *and* c *as follows. Assume that* $(j(m),k(m))$ *produces* a*:*

$$\begin{aligned}\chi_1(a)\otimes\chi_2(b) &:= ((j(1),k(1)),\ldots,(j(m),k(m)), \\ &\qquad (m,m)+(s(1),t(1)),\ldots,(m,m)+(s(n),t(n))) \\ &\quad \textit{with } (m,m)+(s,t) = (m+s,m+t)\end{aligned} \tag{1.2}$$

$$\chi_1(a)\oplus c := \begin{cases} ((j(1),k(1)),\ldots,(j(m),k(m)),(m,p)) \\ \qquad\qquad if\ (j(p),k(p))\ produces\ c \\ ((j(1),k(1)),\ldots,(j(m),k(m)),(m,0)) \\ \qquad\qquad if\ c=1 \\ \chi_1(a) \\ \qquad\qquad if\ c \notin S(\chi_1(a)) \end{cases} \tag{1.3}$$

Obviously, $\chi_1(a)\otimes\chi_2(b)$ *gives an addition chain for* $a\cdot b$*, because* $(j(m),k(m))$ *produces* a *and* $S(\chi_2(b))$ *contains* b*, if started with 1, e.g.* $b_0 = 1$*, and therefore* $a\cdot b$ *if started with* $b_0 = a$*.* $\chi_1(a)\oplus c$ *gives an addition chain for* $a+c$ *iff* c *appears in* $\chi_1(a)$*.*

Using this definition, any equation of the Euclidean decomposition can be used to create an addition chain. For the i-th line of the Euclidean Algorithm, it is

$$r_{i-2} = u_i \cdot r_{i-1} + r_i$$

and hence

$$\chi(r_{i-2}) = \chi(r_{i-1}) \otimes \chi(u_i) \oplus r_i.$$

The arguments of the $\otimes$ operator have been switched compared to the usual Euclidean equation notation to ensure that r_i appears in the resulting addition chain and therefore the $\oplus$ operator really outputs an addition chain for r_{i-2}.

All chains for all values of r_i are known from the line below the i-th line (whose addition chain is, similar to the Extended Euclidean Algorithm, computed first), just the addition chains for the u_i values and for the last r_{k-1}, the greatest common divisor, have to be computed separately – using the continued fraction method recursively.

To deduce an algorithm that only requires a single integer e as input, a good choice for b must be made. In [BCHM95], the authors suggest using

$$b = a \text{ div } 2^{\lfloor \frac{\lambda_2(e)}{2} \rfloor}, \tag{1.4}$$

with div returning the quotient of the division with remainder. The suggested algorithm involves some special cases: The values for any $e = 2^j$ for some j are computed separately with the addition chain

$$\chi(2^j) = (1, 2, 4, \ldots, 2^j), \tag{1.5}$$

because their optimality is known. This saves some computational effort in the recursion. To end the recursion, addition chains for the first three values are provided, e.g.

$$\chi(1) = (),\ \chi(2) = ((0,0)) \text{ and } \chi(3) = ((0,0),(1,0)). \tag{1.6}$$

Note that $\chi(1)$ is the empty chain, because every addition chain's semantics contains 1.

Besides the given choice for b, many other choices are possible, some of them representing already known methods, such as the binary method. A number of different strategies to determine a suitable value for b can be found in [BBB94].

Example 1.17:

(1) Consider the running example $\Pi = (x \in H, 219)$ to demonstrate this algorithm. First, compute an appropriate value for b according to formula (1.4) as

$$b = a \text{ div } 2^{\lfloor \frac{\lambda_2(e)}{2} \rfloor} = 219 \text{ div } 2^4 = 13.$$

Then compute the Euclidean decomposition of (219,13):

Euclidean Algorithm			resulting chain concatenation:
219	$=$	$16 \cdot 13 + 11$	$\chi(219) = \chi(13) \otimes \chi(16) \oplus 11$
13	$=$	$1 \cdot 11 + 2$	$\chi(13) = \chi(11) \otimes \chi(1) \oplus 2$
11	$=$	$5 \cdot 2 + 1$	$\chi(11) = \chi(2) \otimes \chi(5) \oplus 1$
2	$=$	$2 \cdot 1$	$\chi(2) = \chi(1) \otimes \chi(2)$

This shows the continued fractions decomposition of $\frac{219}{13}$ to be [16,1,5,2]. The greatest common divisor of 219 and 13 is 1.
Now addition chains for 1, 2, 5 and 16 must be computed recursively, leading to

$\chi(16)$	$=$	((0,0),(1,1),(2,2),(3,3))	by special rule (1.5)
$\chi(1)$	$=$	()	by special rule (1.6)
$\chi(2)$	$=$	((0,0))	by special rule (1.6)

And as $5 = 2 \cdot 2 + 1$ in the recursive application of the algorithm, it is

$$\chi(5) = ((0,0)) \otimes ((0,0)) \oplus 1 = ((0,0),(1,1),(2,0))$$

With these values known, $\chi(219)$ can be computed easily:

$$\begin{aligned}
\chi(11) &= ((0,0)) \otimes ((0,0),(1,1),(2,0)) \oplus 1 \\
&= ((0,0),(1,1),(2,2),(3,1),(4,0)) \\
S(\chi(11)) &= \{1,2,4,8,10,11\} \\
\chi(13) &= ((0,0),(1,1),(2,2),(3,1),(4,0)) \otimes () \oplus 2 \\
&= ((0,0),(1,1),(2,2),(3,1),(4,0),(5,1)) \\
S(\chi(13)) &= \{1,2,4,8,10,11,13\} \\
\chi(219) &= ((0,0),(1,1),(2,2),(3,1),(4,0),(5,1)) \\
&\quad \otimes ((0,0),(1,1),(2,2),(3,3)) \oplus 11 \\
&= ((0,0),(1,1),(2,2),(3,1),(4,0),(5,1),(6,6),(7,7), \\
&\quad (8,8),(9,9),(10,5)) \\
S(\chi(219)) &= \{1,2,4,8,10,11,13,26,52,104,208,219\} \\
L(\chi(219)) &= 11
\end{aligned}$$

The example shows how the algorithm works for $e = 219$. It also shows that the algorithm is not optimal, because we saw in example 1.14(1) a chain for 219 with length 10.

In [BCHM95], the authors state that this method is better than the binary method on average and its worst case is equal to the binary method's average case. Exact analyses about the average number of arithmetical operations depend upon the chosen strategy to determine a value for b. See [BBB94] for some possible strategies.

Although the arithmetical operations are not counted separately for additions and doublings in this example, it is interesting to note that no hidden doublings occur. While the used operation of concatenating two addition chains always yields the danger of producing such hidden doublings, which may be counted as additions falsely, the addition chains produced by the continued fractions method always produce chains solely consisting of star steps and doublings of the form $(i-1, i-1)$ (see [BBB94], p. 24 and definition 1.4). For such chains, lemma 1.5 ensures that no such miscounting occurs.

1.2.6 Knuth's powertree method

This method has been described by Knuth (see [Knu97]) and uses a tree structure to compute addition chains for all numbers up to a given number e.

idea: The powertree is a tree structure that stores values in a way such that the path from the root to the corresponding knot forms an addition chain. To achieve this, the tree must be build up large enough to contain the searched value for e.

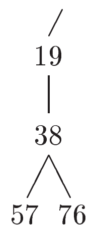

Figure 1.3: Continuing the powertree

The tree is build by initializing the root with the value 1 and by building the k-th level from the $(k-1)$-st level in the following way: take every number on the path from the root to the current knot and attach the sum of that number and of the value of the current knot to the current knot, unless it already appears somewhere in the tree.

Consider continuing to build the tree depicted in figure 1.4. Then for the leftmost leaf, $e = 38$, the following values might be attached to that leaf: (38+1, 38+2, 38+3, 38+5, 38+7, 38+14, 38+19, 38+38). But as (39, 40, 41, 43, 45, 52) already appear somewhere else in the tree, only 57 and 76 may be attached to 38 as new leaves. The result is shown in figure 1.3.

Example 1.18:

(1) The running example $e = 219$ is not easily presented as an example for the powertree, for the tree has a depth of 10 levels and several hundred knots. Figure 1.4 shows Knuth's example (compare [Knu97]) of the powertree with depth 7. The value for $e = 219$ lies down the path highlighted in the figure and it gives the addition chain

$$\begin{aligned}
\chi(219) &= ((0,0),(1,0),(2,2),(3,2),(4,4),(5,4),(6,6),(7,7),\\
&\quad (8,2),(9,8))\\
S(\chi(219)) &= \{1,2,3,6,9,18,27,54,108,111,219\}\\
L(\chi(219)) &= 10.
\end{aligned}$$

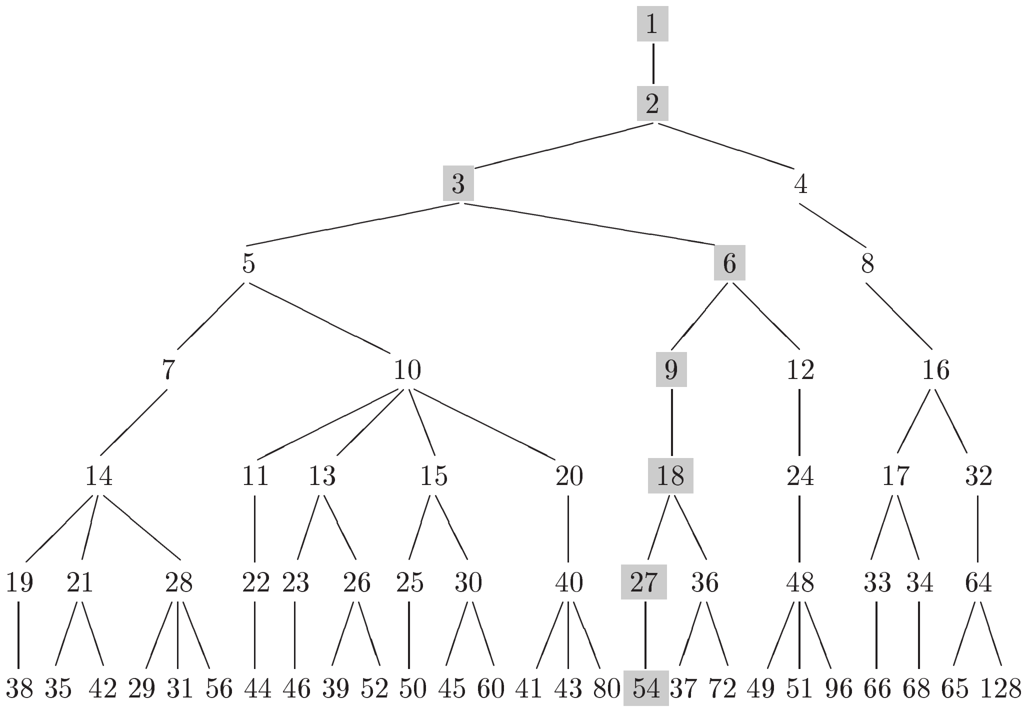

Figure 1.4: The powertree up to level 7 (taken from [Knu97], p.464)

(2) As it can be seen in figure 1.4, addition chains are easily taken from the powertree once it has been constructed. The following addition chains are immediate from the tree:

$$\begin{aligned}
\chi(15) &= ((0,0),(1,0),(2,1),(3,3),(4,3)) \\
S(\chi(15)) &= \{1,2,3,5,10,15\} \\
\chi(33) &= ((0,0),(1,1),(2,2),(3,3),(4,0),(5,4)) \\
S(\chi(33)) &= \{1,2,4,8,16,17,33\}
\end{aligned}$$

As one might imagine, the building of the powertree involves a lot of memory and a lot of work, especially for huge values for e. Nevertheless, for small values of e, e.g. $e < 100,000$, the method gives remarkably good results. As Knuth states (see [Knu97], p.464), for the values of $e < 100,000$, this method excels the factor method in almost 89% of the cases, ties in about 11% and only loses 6 times. The latter also shows that the powertree method is not optimal.

1.2.7 The BGMW-algorithm

The exponentiation approaches discussed up to this point always assume a new exponentiation problem every time they're applied. But in practical applications, there are a number of cases, where this is not true, because the same basis has to be raised to different powers for a number of times. This problem occurs for example if the Diffie-Hellman key exchange protocol is used (see [Sti95], §8.4). In cases like these, precomputation may save a substantial part of the operations usually needed to solve a number of similar exponentiation problems. This was first suggested by E. F. Brickell, D. M. Gordon, K. S. McCurley and D. B. Wilson (BGMW) in [BGMW92], a summary can be found in [Gor98]. [GN00] gives comparisons to other addition chain algorithms. The description of the idea follows [Gor98].

idea: The BGMW algorithm contains two ideas, precomputation and the use of special number systems.

The basic idea is to use a well known addition chain algorithm, e.g. the binary method or the m-ary method and precompute a number of powers of the given base of an exponentiation problem $\Pi = (x \in H,\ e \in \mathbb{N})$. These values are stored and can be used during the main exponentiation process. If a base is repeatedly raised to some powers, the precomputation step has to be performed only once, thus saving a huge number of steps successively.

A simple application is to use the binary method and to precompute all 2^k-ary powers of the base x. The binary method will then be slightly modified by not using repeated squaring but just multiplying the appropriate powers of the base indicated by the binary expansion. This requires the same number of multiplications as the traditional binary method presented in section 1.2.1, but no squaring at all. See example 1.20(1) for an application of this technique.

An application using the m-ary method would either precompute and store all powers x^{m^k}, or it may even use a more sophisticated method described by BGWM, where powers with equal coefficients are multiplied together and then raised to powers:

Let $h \in \mathbb{N}$ be some bounding number (which will determine the used number system), then for a set of appropriate choices of n_i (usually powers of a certain base b, e.g. $n_i = 2^i$ allows the binary expansion), the exponent e can be written as

$$e = \sum_{i=0}^{l-1} e_i \cdot n_i \qquad \text{with } 0 \leq e_i \leq h \text{ for all i.} \tag{1.7}$$

Now x^e can be written as a distinct coefficient decomposition as

$$x^e = \prod_{d=1}^{h} {c_d}^d, \tag{1.8}$$

where the c_d values represent the powers with equal coefficients

$$c_d = \prod_{i:e_i=d} x^{n_i}.$$

Gordon ([Gor98]) states that the formula (1.8) can be computed efficiently using only $l + h - 2$ multiplications using the formula

$$\prod_{d=1}^{h} {c_d}^d = c_h \cdot (c_h \cdot c_{h-1}) \cdots (c_h \cdot c_{h-1} \cdots c_1). \tag{1.9}$$

This formula can be easily implemented using the following iteration, where the value of b is constantly increased, thus for every loop, b takes the value of the next factor in brackets of the above formula (1.9). Note that the values x^{n_i} are precomputed and accessible.

Algorithm 1.19 (BGMW computation of x^e)
(following [BGMW92] p. 202)

Input: *the base x, the exponent e and the BGMW decomposition of the exponent e from formulas (1.7) and (1.8).*
Output: x^e

```
b := 1
a := 1
from d = h downto 1 do
    for each i such that e_i = d do
        b := b · x^{n_i}
    a := a · b
return a
```

The second idea, to use special number systems, arises from the fact that this algorithm works for any set of n_i's, which leads to the possibility to represent all integers in a desired range. BGMW themselves suggest some number systems, e.g. $h = m - 1$ and $\{n_i\} = \{m^i\}$ (m-ary method), or quite unusual systems like $h = 8$ and $\{n_i\} = \{\pm 1, -2, 9, 10\} \cdot \{29^i\}$ ([Gor98] p. 141). The use of these so-called basic digit sets for a certain base may improve the ratio between the operations saved and the use of memory.

The common values (see [GN00]§3) are $\{n_i\} = \{b^i\}$ with the base b determined from

$$b = 2^{\lfloor \log_2 \lambda_2(e) - 2\log_2 \log_2 \lambda_2(e) \rfloor + 1}.$$

Example 1.20:

(1) Consider the exponentiation problem $\Pi = (x \in H, 23)$. Suppose the idea of precomputation should be used and the binary method should be applied as the main exponentiation algorithm. As $\lambda_2(23) = 6$, the values $1, x^{2^0}, x^{2^1}, x^{2^2}, x^{2^3}, x^{2^4}$ are precomputed and stored. As the binary expansion of 23 is $(23)_2 = 101110$, the modified binary method looks up the values for the four ones as $x^{2^4}, x^{2^2}, x^{2^1}$ and x^{2^0}. The result can be computed using only multiplications as $x^{23} = x^{16} \cdot x^4 \cdot x^2 \cdot x^1 = x^{16+4+2+1}$.

This approach leads to the addition chain

$$\begin{aligned} \chi(23) &= ((0,0),(1,1),(2,2),(3,3),(4,2),(5,1),(6,0)) \\ S(\chi(23)) &= \{1,2,4,8,16,20,22,23\} \end{aligned}$$

with length $L(\chi(219)) = 7$.

(2) Suppose the exponentiation problem $\Pi = (x \in H, 219)$, to be given. Assume that the improved m-ary approach should be used for exponentiation.

Let the number system be $h = 3$ and $\{n_i\} = \{(2^2)^i\}$ (forcing a 4-adic expansion of 219). Then the BGMW algorithm works as follows:

$$\begin{array}{lllll} \text{h} = & 3 \Longrightarrow \text{precompute: } x^4,\ x^{4^2},\ x^{4^3} & & & \\ (219)_4 = & (\ \underline{1\,1}\ \ \underline{0\,1}\ \ \underline{1\,0}\ \ \underline{1\,1}\ ) & & & \\ \text{values}: & \ \ 3\ \ \ 1\ \ \ 2\ \ \ 3 & & & \\ c_d: & c_1 = x^{4^2} & c_2 = x^{4^1} & c_3 = x^{4^3} \cdot x^{4^0} & \\ & x^{16} & x^4 & x^{65} & \end{array}$$

$$x^{219} \overset{(1.9)}{=} c_3 \cdot (c_3 \cdot c_2) \cdot (c_3 \cdot c_2 \cdot c_1)$$

The last line is now computed using algorithm 1.19:

b = 1	a = 1
b = $x^{4^3} \cdot x^{4^0} = x^{65}$	a = x^{65}
b = $x^{4^3} \cdot x^{4^0} \cdot x^{4^1} = x^{69}$	a = $x^{65} \cdot x^{69} = x^{134}$
b = $x^{4^3} \cdot x^{4^0} \cdot x^{4^1} \cdot x^{4^2} = x^{85}$	a = $x^{134} \cdot x^{85} = x^{219}$

The resulting addition chain $\chi(219)$ is therefore:

$$\begin{aligned}
\chi(219) &= (\boxed{(0,0)}, (1,1), \boxed{(2,2)}, (3,3), \boxed{(4,4)}, (5,5), \quad (6,0),\\
&\quad (7,2), (8,4), (9,8), (10,9))\\
S(\chi(219)) &= \{1, \boxed{2}, 4, \boxed{8}, 16, \boxed{32}, 64, \quad 65, 69, 85, 134, 219\}\\
L(\chi(219)) &= 11
\end{aligned}$$

For this input, the BGMW method produces a different chain than the ordinary m-ary method (see example 1.13) with the same length. Again, this method is not optimal, as the table in appendix A lists an addition chain of length 10 for $e = 219$. But if this exponentiation problem is part of a sequence of exponentiations of x, the precomputation would have been done before, hence, only the 5 multiplications would have to be performed – giving savings of more than 50% !

Note that the values before the gap are values that have been computed during precomputation using only squarings, all values after the gap have been computed using algorithm 1.19, where only multiplications are used. The shaded values are values that are needed in order to precompute the base multiples, because the operation to raise a number to its 4-th power can only be done using two squarings. Therefore, the shaded values are not used within the algorithm, they're – from that perspective – created in vain.

The BGMW method specializes the m-ary method in assuming that some powers are precomputed and are stored for look up for a number of following exponentiation problems with the same base x. In cases like these, the method is obviously a very good choice, but it is not in a general application, where exponents and bases are assumed to be unpredictable (for a comparison of some values see [GN00], table 4.2). The BGMW method also generalizes the m-ary method in allowing more number representations than the usual m-adic expansions, which are required for the m-ary method.
As this thesis is focused on general exponentiation problems, without a pattern in the choice of bases, the BGMW method is not closely examined. The BGMW method works for any group and it can be parallelized to save additional time. Although, besides its use for repeated bases, another disadvantage of the BGMW method is the use of memory. It can be adapted to deal with restricted memory, but it still needs a lot space to store all precomputed values and it tends to use too much memory. These facts, as well as approaches to solve this problem, are described in greater detail in [Gor98], §5.2.
Results about the costs state that the number of arithmetical operations including the precomputation is at most

$$(1 + o(1)) \cdot \frac{\lambda_2(e)}{\log_2 \lambda_2(e)} + \lambda_2(e),$$

according to [GN00], table 4.1.

1.2.8 The data compression method by Yacobi

Y. Yacobi suggested exploiting the similarities between exponentiation and data compression. He notes that there are two main similarities, the use of repetitions – frequent messages, which are assigned short codes in data compression, and frequent computations, which should be reused in exponentiation – and the use of small differences between subproblems – sending the difference to a previous message (Δ modulation) in data compression and using the exponentiation law $x^{n+\Delta} = x^n \cdot x^\Delta$ in exponentiation ([Yac90]). He applies the well known compression algorithm of Lempel and Ziv (see [ZL78]) to build an exponentiation algorithm.
The method is described in [Yac90] and [Yac98], comparisons to other addition chain algorithms can be found in [GN00], §4.

idea: The most promising approach to reduce the number of operations in exponentiation is to reduce the number of multiplications. Most algorithms use precomputation in order to reduce operations in the main exponentiation step. Data compression algorithms try to optimize the needed number of precomputation steps by choosing an optimal set of precomputed values.

While the m-ary method and the constant length nonzero window method use windows of a fixed length, which do hardly adapt to the structure of the input, and the variable length nonzero window method is also bounded in its choice for the length of windows, the approach of Yacobi does not impose restrictions on the length of windows and combines the precomputation step with the partitioning of the input into windows. In doing this, it assures to precompute only those values, which actually appear as windows in the partition.

This task is achieved by building up a compression tree in the precomputation step, where window values are stored. This is done exactly in the same way the Lempel-Ziv algorithm suggests: The binary expansion of the exponent e is scanned from right to left and windows are created on the fly. A new window W_i starts with the digit 1 (tailing zeros are skipped) and while scanning the digits of the input the tree is traversed accordingly. If a leaf is reached, the next digit scanned closes the new window W_i and creates a new child of the leaf. The new node in the tree gets the value x^{W_i}, which is the parent value if the last digit scanned was 0, or $x^{2^k} \cdot x^{W'}$ if the last digit scanned was 1, where k is the current depth of the tree (the number of digits within the new window) and W' is the parent window pattern. The node is named after the window pattern $(W_i)_2$. For this step, several values of x^{2^i} have to be precomputed explicitly, too.

The construction of the compression tree is demonstrated in example 1.21(1). The precomputation results in a partition of the binary expansion into several windows W_i. The main exponentiation is then done in the same way as it was presented with the m-ary method and the sliding window methods, the result accumulates by squaring as many times as there are digits within the next window plus the number of tailing zeros. This result is multiplied with the precomputed value of the next window. Consider the resulting sequence of windows to be

$$(e)_Y \quad = \quad (W_{\lambda_Y(e)-1}, 0^{z_{\lambda_Y(e)-1}}, W_{\lambda_Y-2}, 0^{z_{\lambda_Y(e)-2}}, \ldots, W_0, 0^{z_0}),$$

and the length of the i-th window plus tailing zeros of the $(i+1)$-st window to be $L_i = \#W_i + z_{i+1}$, then the exponentiation problem $\Pi = (x, e)$ is solved as

$$x^e \quad = \quad ((\ldots(x^{W_{\lambda_Y(e)-1}})^{2^{L_{\lambda_Y(e)-2}}} \cdot x^{W_{\lambda_Y(e)-2}})^{2^{L_{\lambda_Y(e)-3}}} \cdots x^{W_0}.$$

Example 1.21:

(1) Consider the exponentiation problem $\Pi = (x \in H,\ e = 54,962,861)$. The binary expansion of e is partitioned as follows:

First the compression tree is initialized with a root named as node 1, which contains the window value $x^1 = x$. Then the binary expansion $(e)_2$ is scanned from right to left and new windows are created as described above. New windows always start with a 1, hence tailing zeros are skipped. The windows end when a leaf in the compression tree is found. The next digit scanned closes the window and creates a new node in the tree. The following lines show the partitioning of the binary expansion of e until the 5th window is found:

$$\begin{array}{rcll}
(e)_2 & = & (11010001101010101010101101) & \\
 & = & (1101000110101010101010110\underline{1}) & \text{window } W_0 \\
 & = & (11010001101010101010101\underline{11}0\underline{1}) & \text{window } W_1 \\
 & = & (1101000110101010101\underline{01}0\underline{11}0\underline{1}) & \text{window } W_2 \\
 & = & (1101000110101010\underline{101}\underline{01}0\underline{11}0\underline{1}) & \text{window } W_3 \\
 & = & (11010001101\underline{0101}0\underline{101}\underline{01}0\underline{11}0\underline{1}) & \text{window } W_4 \\
 & = & (1101000{\color{red}1101}\underline{0101}0\underline{101}\underline{01}0\underline{11}0\underline{1}) & \text{window } {\color{red}W_5}
\end{array}$$

The sixth window is shown in red. The digit sequence of that window

W_5 (the first window is W_0) determines the highlighted path in the compression tree depicted in figure 1.5. After the first three digits 101 are read, the next digit determines a node not yet in the tree, therefore the window is closed and the requested new leaf is created. The new value is computed as $x^{2^3} \cdot x^{W_3}$. Therefore, the values x^2, x^{2^2} and x^{2^3} also need to be precomputed.

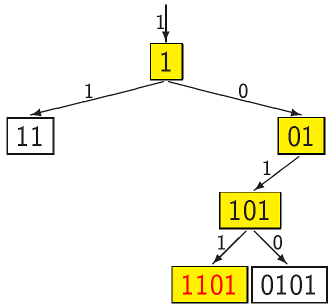

Figure 1.5: The partially constructed Yacobi-compression tree at the time the window 1101 is to be inserted

The complete partitioning is

$$(e)_Y = \quad (\underline{1\,1\,0\,1}\,0\,0\,0\,\underline{1\,1\,0\,1}\,\underline{0\,1\,0\,1}\,0\,\underline{1\,0\,1}\,\underline{0\,1}\,0\,\underline{1\,1}\,0\,\underline{1})$$

and figure 1.5 already shows the complete tree, because the last window W_6 is due to the end of the binary expansion the same as W_5. The exponentiation problem is solved as

$$x^{54,962,861} = ((((((x^{W_6})^{2^7} \cdot x^{W_5})^{2^4} \cdot x^{W_4})^{2^4} \cdot x^{W_3})^{2^2} \cdot x^{W_2})^{2^3} \cdot x^{W_1})^{2^2} \cdot x^{W_0}$$

(2) Consider the standard example $\Pi = (x \in H,\ 219)$. The binary expansion of 219 is partitioned according to the Lempel-Ziv compression algorithm and the tree depicted in figure 1.6 results. The tree nodes are named after the corresponding window pattern and their value is $x^{pattern}$:

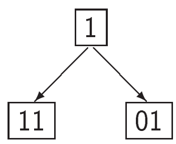

Figure 1.6: The Yacobi-compression tree for $e = 219$

$$\begin{array}{rcl}
(219)_2 & = & (\quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad) \\
(219)_Y & = & (\quad \underline{1\,1} \quad 0 \quad \underline{1\,1} \qquad \underline{0\,1} \qquad \underline{1} \quad) \\
 & = & (\quad W_3 \qquad W_2 \qquad W_1 \qquad W_0 \quad) \\
x^{219} & = & (((\, x^{W_3} \,)^{2^3} \cdot x^{W_2} \,)^{2^2} \cdot x^{W_1} \,)^2 \cdot x^{W_0} \\
 & = & (((\, x^3 \,)^8 \cdot \; x^3 \,)^4 \cdot \; x \,)^2 \cdot \; x \\
 & = & x^{((24+3)\cdot 4+1)\cdot 2+1} = x^{(108+1)\cdot 2+1} = x^{219} \\
\chi(219) & = & (0,0),(1,0), \quad (2,2),(3,3),(4,4),(5,2),(6,6),(7,7), \\
 & & (8,0),(9,9),(10,0)) \\
S(\chi(219)) & = & \{1,2,3, \quad 6,12,24,27,54,108,109,218,219\} \\
L(\chi(219)) & = & 11
\end{array}$$

The highlighted values, which are the same as the values before the gap in $\chi(219)$, have been precomputed. The optimal value of $L(\chi(219)) = 10$ could not be reached, the data compression approach is therefore not optimal.

The data compression approach of Yacobi creates balanced compression trees on average, as the Lempel-Ziv algorithm does, in spite of example 1.21(1), which seems to suggest otherwise. The approach is most successful for compressible exponents, where the gain compared to the m-ary method is the compression ratio on average. Asymptotically though, the m-ary method is superior, especially for random exponents (which are assumed within this thesis). Therefore, this method is not examined in detail in the following chapters.
The expected number of arithmetical operations for a random exponent $e \in \mathbb{N}$ is

$$\lambda_2(e) + \frac{3}{2} \cdot \frac{\lambda_2(e)}{\log_2 \lambda_2(e)} \cdot (1 + o(1)) + o\left(\frac{\lambda_2(e)}{\log_2 \lambda_2(e)}\right),$$

according to [GN00], table 4.1.

1.2.9 The data compression method by Bocharova and Kudryashov

The data compression approach of Yacobi, who uses the Lempel-Ziv compression algorithm in exponentiation, motivates the search for other data compression techniques, which could prove useful in the field of exponentiation. In [BK95], I. E. Bocharova and B. D. Kudryashov suggest the use of typical sets and VF-codes for this task, [KY98] extend the idea to addition-subtraction chains. Comparisons of experiments can be found in [GN00].

idea: The basic idea to this approach is the use of a proper subset D of the set Ω of all possible window values. If the input's binary expansion was partitioned into windows of equal length d, the set Ω would be the set $\{0, 1, \ldots, 2^d - 1\}$.

In the precomputation step, the values x^k are precomputed and stored for all elements $k \in D$. They can be used in the main exponentiation step, where the precomputed values can be accessed through table look-up whenever a window W_i appears, whose value is in D. If the window's value is not in D, the needed value X^{W_i} is computed using the binary method. If long frequent substrings are in the set D, the number of multiplications in the main exponentiation step can be reduced.

Obviously, the choice of D determines the performance of this approach.

The first idea to find a good choice for D is the use of typical sets (see [BK95], where [CT91] is recommended for further reading). A typical set D is a subset of the set Ω of elements which are assigned probabilities according to a probability distribution $F(x \in \Omega)$ with the property that for a fixed window size of d digits (m-ary partitioning) the following equations hold:

$$\begin{aligned} |D| &= 2^{d\cdot(\mathcal{H}(p)+\delta(d))}, \\ \sum_{k \in D} \mathrm{Prob}(k) &= 1 - \epsilon(d) \end{aligned}$$

where

$$\epsilon(d), \delta(d) \longrightarrow 0 \text{ if } d \longrightarrow 0,$$

and $\mathcal{H}(p) := -p \log_2 p - (1-p)\log_2(1-p)$ is the binary entropy function for a probability p, according to which the input sequence is independent identically distributed with $\mathrm{Prob}(e_i = 1) = p$. If the source probability distribution is known, this approach is better than Yacobi's approach using the Lempel-Ziv algorithm and it is better than the m-ary method for entropy less than 1. Although, this advantage takes effect only for very large values of $\lambda_2(e)$ for an exponentiation problem $\Pi = (x, e)$, it is therefore not recommended by the authors.

The improved idea is to construct a proper set D using the variable-to-fixed length code, the optimal Tunstall VF-code. The algorithm is described in [JS72], where the original algorithm is contributed to [Tun68]. A detailed discussion of this technique for exponentiation can be found in [KY98]. The name does not imply that the windows have a fixed length, the naming originates from the use of this algorithm in data compression, where all words of D are assigned code words of a fixed length. The improved idea also makes use of an idea used in the constant length nonzero sliding window method (see section 1.2.3): as zeros only account for squarings, a new window is only started at the digit 1, zeros are skipped.

The set D is created in the following way:

For a given size K, a set of K words is constructed using the following algorithm 1.22. The probability p of the digit 1 can either be assumed to be $\frac{1}{2}$, as this is the probability that any single digit in the binary expansion of the exponent e is 1, or it can adapt to the actual input by counting the number of ones and deriving the actual distribution for each input (see example 1.23(2)). Note that the notation w0 denotes the concatenation of w with 0. Lines 9-11 are not part of the original Tunstall algorithm.

Algorithm 1.22 (optimal binary Tunstall VF-code)
(following F. Jelinek and K. S. Schneider in [JS72], Algorithm 1)

Input: *the requested size K of the output set, the probability p of the digit* 1

Output: *a set D of K words*

00 Set $D := \{0,\ 1\}$
01 Set $\mathsf{p}_1 := \mathsf{p}$ # probability of word 1
02 Set $\mathsf{p}_0 :=$ 1-p # probability of word 0
03 while $\#D < K$ do
04   Choose w $\in D$ with maximal probability p_w
05   Set $D := D \backslash \{$ w $\}$ # remove word to be extended
06   Set $D := D \cup \{\mathsf{w0}, \mathsf{w1}\}$ # add new words
07   Set $\mathsf{p}_{\mathsf{w0}} := (1 - \mathsf{p}) \cdot \mathsf{p}_\mathsf{w}$ # set probabilities of new words
08   Set $\mathsf{p}_{\mathsf{w1}} := \mathsf{p} \cdot \mathsf{p}_\mathsf{w}$
09 for all w $\in D$ do # adapt to exponentiation problem
10   Set w := 1w
11   Remove tailing zeros from w
12 return D

After the precomputation is done, the input sequence is partitioned according to the possible words. The Tunstall algorithm ensures that before line 9 is reached the set D is complete, e.g. every sufficiently long input string has a prefix that is a word in D, and proper, e.g. no element of D is the prefix of another element of D.

The algorithms described in [BK95], [JS72] and [KY98] all parse the input's binary expansion from left to right, although it can also be done vice versa. In this sense, lines 9-11 of algorithm 1.22 ensure that new windows always start with the digit 1 and tailing zeros on words are also omitted, reducing the length of the addition sequence that computes all x^k for $k \in D$. After adding a 1 as the leading digit, all windows must start with a 1, but the set D may not be complete and proper anymore. However, the partitioning is still unambiguously possible, if the longest possible match must be applied.

As runs of zeros between windows are skipped, the input is partitioned as

$$(e)_B = (w_0 0^{z_0} w_1 0^{z_1} \ldots w_{\lambda_B(e)-1} 0^{z_{\lambda_B(e)-1}}),$$

where $\lambda_B(e)$ denotes the number of nonzero windows created by this approach.

To perform the precomputations of x^k efficiently, the construction of the set D can also be interpreted as the creation of a tree, where the extension step equals the creation of new children for a leaf (as described in [KY98]). The precomputation follows that tree structure. The main exponentiation step is equivalent to the Yacobi method, with accumulative squarings and multiplications with precomputed window values.

The variable K is an optimization parameter, the performance of the exponentiation algorithm depends on a good choice for K. An analytical solution is difficult and seems not to be present in literature. [KY98] suggest exhaustive search to find the right value for K. In the case where $p = \frac{1}{2}$, which is the probability assumed for random inputs and which will be the behaviour the input sequence will show for large values of $\lambda_2(e)$, the authors state that for "almost all" values of $\lambda_2(e)$, the algorithm is optimized when $K = 2^{i-1}$ for some $i \in \mathbb{N}$. Examples are $K = 16$ for $\lambda_2(e) = 512$, $K = 32$ for $\lambda_2(e) = 1024$ etc. In these cases, all words in D have the same length.

More thoughts about optimization concerns can be found in [KY98].

Example 1.23:

(1) Consider the example $\Pi = (x \in H,\ 219)$ and the choice $K = 2$. Recall the binary expansion of $e = 219$ to be

$$(219)_2 = (\ 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1\ )$$

The Tunstall algorithm 1.22 initializes the set D as $D = \{0, 1\}$ and as these are already two elements, no extension steps are necessary. In the adaption step, as a leading 1 is added and tailing zeros are omitted, D is transformed into $D = \{1, 11\}$ which leads to the precomputation of the values (x^2, x^3). Although x^2 does not correspond to a word in D, it has to be computed in order to compute x^3. $(219)_2$ is then partitioned according to D in the following manner:

$$\begin{aligned}
(219)_B &= (\ \underline{1\,1}\ \ 0\ \ \underline{1\,1}\ \ 0\ \ \underline{1\,1}\ ) \\
&= (\ W_0 \qquad W_1 \qquad W_2\ ) \\
x^{219} &= ((\ x^{W_0}\ )^{2^3} \cdot x^{W_1}\ )^{2^3} \cdot x^{W_2} \\
&= ((\ x^3\ )^8 \cdot x^3\ )^8 \cdot x^3 = x^{27 \cdot 8 + 3} = x^{219} \\
\chi(219) &= (0,0),(1,0),\quad (2,2),(3,3),(4,4),(5,2),(6,6),(7,7), \\
&\quad (8,8),(9,2)) \\
S(\chi(219)) &= \{1,2,3,\quad 6,12,24,27,54,108,216,219\} \\
L(\chi(219)) &= 10
\end{aligned}$$

With $L(\chi(219)) = 10$, this algorithm is one of the few algorithms that compute an optimal addition chain for $e = 219$. However, this algorithm is not optimal in general.

(2) Now consider the Tunstall algorithm once again being applied to the exponentiation problem $\Pi = (x \in H, 219)$ and assume $K = 3$. If the distribution of the digits is derived from the input sequence by counting the digit occurrences, the set D is initialized with $\{0, 1\}$ and the probabilities $p_0 = \frac{1}{4}, p_1 = \frac{3}{4}$. Thus the third value is created as the extension of 1, $D = \{0, 10, 11\}$, which leads to the adapted output $D = \{1, 11, 111\}$, requiring the four precomputation steps of (x^2, x^3, x^4, x^7), with x^2 and x^4 being intermediate values (appendix A shows that this corresponds to an optimal addition chain for 7). 219 is partitioned as follows:

$$\begin{aligned}
(219)_B &= (\ \underline{1\,1}\ \ 0\ \ \underline{1\,1}\ \ 0\ \ \underline{1\,1}\ ) \\
\chi(219) &= ((0,0),(1,0),(1,1),(3,2),\quad (2,2),(5,5),(6,6),(7,2), \\
&\quad (8,8),(9,9),(10,10),(11,2))
\end{aligned}$$

$$
\begin{aligned}
S(\chi(219)) &= \{1,2,3,4,7, \quad 6,12,24,27,54,108,216,219\} \\
L(\chi(219)) &= 12
\end{aligned}
$$

because the main exponentiation step is the same as in example 1. Note that the numbers 2,3,4 and 7 belong to the precomputation step.

If the initial distribution is assumed to be the expected distribution, hence $p_1 = p_0 = \frac{1}{2}$, the digit $\mathtt{0}$ could be chosen for extension, resulting in $D = \{\mathtt{00}, \mathtt{01}, \mathtt{1}\}$, which gives the output $D = \{\mathtt{1}, \mathtt{101}, \mathtt{11}\}$. The binary expansion of 219 would be partitioned in the same way, but the precomputation would be (x^2, x^3, x^5). This creates the following addition chain of length 11:

$$
\begin{aligned}
\chi(219) &= ((0,0),(1,0),(2,1), \quad (2,2),(4,4),(5,5),(6,2),(7,7), \\
&\quad\; (8,8),(9,9),(10,2)) \\
S(\chi(219)) &= \{1,2,3,5, \quad 6,12,24,27,54,108,216,219\}
\end{aligned}
$$

This shows that the choice of K and that of the initial distribution is crucial to the performance of this algorithm.

The authors state that the VF-code is better than the m-ary method if the digits of the binary input sequence are independent identically distributed with $\mathrm{Prob}(X_i = 1) = \frac{1}{2}$ and it uses less memory. Experimental results from [GN00] show this for the number of multiplications for input sequences with $\lambda_2(e) \in \{512, 1024\}$, but the number of squarings is slightly higher than that of the m-ary method. Although this method seems to be faster than the m-ary method for large $\lambda_2(e)$, the following chapters will not closely examine this method, because of the greater popularity of the m-ary method and the fact that this method is build upon the m-ary method, which should be examined first. [KY98] state that this method is more efficient than the m-ary method and the constant length nonzero sliding window method for the finite case as well as for the asymptotic case.

Analyses in [GN00], table 4.1, show that the method requires a total of

$$\lambda_2(e) + \frac{\lambda_2(e)}{(\log_2 \lambda_2(e))^2} + \frac{\lambda_2(e)}{\log_2 \lambda_2(e)} \cdot (1 + o(1))$$

arithmetical operations on average.

1.2.10 Other approaches

There are also other approaches, which cover special fields of applications. One of the most important approach is the use of normal bases. In some groups, e.g. $\mathbb{F}_{p^n}$ for a prime p, normal basis representation can be chosen, which allows to compute the Frobenius automorphism for virtually no costs, because it can be achieved through a cyclic shift of the coordinates. In the

most common case of $\mathbb{F}_{2^n}$, squarings become irrelevant. An example for an algorithm in this situation can be found in [Gat91b].
The following definition introduces normal bases:

Definition 1.24 (normal basis)
(from M.Nöcker in [Nöc01], Def. 6.1)
A basis of $\mathbb{F}_{q^n}$ as a vector space of $\mathbb{F}_q$ of the form $\mathcal{N} = (\alpha, \ldots, \alpha^{q^{n-1}})$ consisting of the conjugates of a suitable element $\alpha \in \mathbb{F}_{q^n}$ with respect to $\mathbb{F}_q$ is called a normal basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$. The element α is then called normal (or free) in $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$.

Theorem 1.25 (normal basis theorem)
(from M.Nöcker in [Nöc01], The. 6.2)
There exists a normal basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$.

Proof:

The proof is not presented as it would extend the scope of this thesis. M. Nöcker suggests different proofs presented in [Hen88] and [Jun93]. $\square$

For those groups, where a normal basis representation is possible, squarings or some exponentiation steps are free, thus reducing the amount of arithmetical operations significantly. As this thesis tries not to concentrate on special groups, this approach is not analyzed in detail.
There are also approaches to use the concept of addition chains in parallelization of exponentiation. See [Nöc01] for detailed readings on that matter.

1.2.11 Summary

As this overview of the most important algorithms for ordinary addition chains shows, there are many approaches to find fast exponentiation algorithms motivated by very different mathematical concepts. The concept being most interesting is the general windowing technique. This approach uses an arbitrary digit representation of the exponent, and all the practical useful methods as the binary, the m-ary and the sliding window methods are clearly members of that family. But even more sophisticated methods as the continued fractions method, the BGMW-method and the data compression methods basically use some form of windowing or base representation. This interpretation is natural for most algorithms, like for the BGMW-method (as stated by the inventors themselves in [BGMW92]) or the data compression method (where every tree knot represents a window). But also methods like the continued fractions approach can be interpreted in this way (for this example by using the continued fractions vector $u = [u_1, u_2, \ldots, u_k]$ as a

basis for the representation of e, which can be uniquely defined using the Euclidean Algorithm, because it outputs uniquely defined values).
All windowing method algorithms follow the steps of a general algorithm:

Algorithm 1.26 (general addition chain algorithm)

```
partitioning of the input into windows
precomputation of window values
exponentiation with window values present
```

For some algorithms, not all steps are explicitly performed, for example the binary method doesn't require the partitioning of the binary expansion into 1-digit windows and it doesn't have to precompute the possible window values 1 and x.
The other methods are not applicable in cryptographic applications, they either use too much memory (like the powertree) or too much time (like the prime factorization of the factor method would do).
For these reasons, it is most reasonable to take a closer look into the general windowing methods, especially into the basic method, the binary method, and its direct generalization, the m-ary method. This will be the task of the next chapter.

Chapter 2

Cost analysis of the basic windowing algorithms for addition chains

This chapter will take a closer look on the two most popular addition chain methods, the binary method and the m-ary method. Both methods will be analyzed in detail to provide results that can be compared to those of the corresponding addition-subtraction chains developed during the remaining chapters of this thesis. To give a framework for comparisons, cost measures are introduced and motivated first.

2.1 Definitions

Most cost analyses for addition chain algorithms found in the literature only regard the total number of arithmetical operations. This makes sense if the length of the resulting addition chains is of interest.

But although the sole fact that an algorithm needs fewer operations than some other algorithms might suggest its superiority over the other algorithms, the fields of applications differ, and so do the individual costs of the different operations. In practical applications, the mere length of an addition chain is not the only cost factor. It is also important to know what the corresponding operation for the doubling and the addition are in the semigroup H of a given exponentiation problem $\Pi = (x \in H, e \in \mathbb{N})$ and how fast they can be computed. It has already been pointed out in section 1.1.1, that there are cases, where the real costs of the corresponding operations may differ. This has led to the conclusion that doublings may not generally be replaced by additions.

Because of this fact, it is important to pay respect to the different kinds of operations involved within an algorithm and to differentiate between them when stating results about the costs. The algorithms within this section will give individual results and in order to do so, some definitions and agreements about the notation are helpful. First, counters for the numbers of arithmetical operations "addition" and "q-step" are defined. In definition 3.2, a cost measure for "inversion" will be added.

Definition 2.1 (number of additions A)
The number of addition steps that create an element by adding two different prior computed elements (as opposed to a doubling step, which adds the same prior computed element to itself), in an addition chains $\chi(e)$ for a number $e \in \mathbb{N}$ will be denoted as $A_{\chi(e)}$. This number is equal to the number of multiplications used to solve an exponentiation problem $\Pi = (x \in H, e \in \mathbb{N})$, that presents the task to raise x to its $e-th$ power x^e using this addition chain $\chi(e)$.
If a whole algorithm is to be analyzed, the total number of additions is denoted as A.

In order to allow a general representation of the used concepts and to prevent the creation of many notations denoting the same concept, the notation for counting doublings will be defined following the notation motivated by q-addition chains (see definition 1.11) as defined in [Nöc01]. It allows to include the counting of general q-steps, where a base is raised to some q-th power. This concept is most effective if general bases are used to represent the exponent e, as it is done by the m-ary method and, to some extend, by windowing methods in general. This definition doesn't effect the definition of $A_{\chi(e)}$, because the problem of adding is the same for any base.

Definition 2.2 (number of q-steps Q(q))
The number of q-steps in a q-addition chain $\chi(e,q)$ for a given number $e \in \mathbb{N}$ is denoted by $Q_{\chi(e)}(q)$.
In ordinary addition chains, which will be the sole subject of this thesis, only 2-steps (or doublings) appear. For this reason, $Q_{\chi(e)}$ may be used as an abbreviation for $Q_{\chi(e)}(2)$.
$Q_{\chi(e)}(q)$ also denotes the number of exponentiations with q in the computation of a given power of a given base in the case of using the addition chain $\chi(e)$ to perform the task.
If a whole algorithm is to be analyzed, the total number of q-steps is denoted as $Q(q)$. The notation can (and will) also be used to denote the total number of exponentiations with the power q within an algorithm.

It may seem a little bit confusing first to use the q-addition chain notation to denote arbitrary exponentiations, if only ordinary addition chains with $q = 2$ are used. But this concept has many advantages. On the one hand, it eliminates the need to create a new notation denoting known concepts, on the other hand, some algorithms use arbitrary exponentiation. These cases, where y^q must be computed, correspond to some subsequence within the addition chains, which has to perform that task using a known addition chain algorithm to solve the exponentiation problem $\Pi' = (y, q)$, which appears as a subproblem in the computation of $\Pi = (x, e)$. If these exponentiations can be counted separately first, it allows to generalize the methods in special applications, where the computation of a q-th power may be done faster than using only multiplications and squarings, for example by forming an explicit formula or using special properties of q in a hardware implementation. As this concept is a highly specialized case, it will not be examined in detail within this thesis.

Example:

Recall example 1.13, where the exponentiation problem $\Pi = (x, 219)$ has been solved using the m-ary method for $m = 4$. As the base of the expansion is 4, the digits are concatenated using exponentiations with 4, hence creating the need to solve the exponentiation problem $\Pi' = (x, 4)$, which is very easily and best solved using two consecutive squarings,

$$\begin{aligned} x^{219} &= (((x^3)^4 \cdot x)^4 \cdot x^2)^4 \cdot x^3 \\ &= ((((((x^3)^2)^2 \cdot x)^2)^2 \cdot x^2)^2)^2 \cdot x^3. \end{aligned}$$

The number of arithmetical operations for this algorithm and this example are therefore (not including the needed precomputations !)

$$\begin{aligned} A &= 3 \\ Q(4) &= 3 \quad \Rightarrow Q(2) = Q = 6. \end{aligned}$$

Together with the one addition and one squaring to form the precomputation, the total costs of this example add up to A + Q = A + Q(2) = 4 + 7 = 11 operations, as example 1.13 states, too.

The example also shows two things: On the one hand, it can be seen that some algorithms mix different kinds of q-steps. The m-ary method as performed in the above example needs one doubling (2-step) to precompute x^2 and 3 4-steps to form other values. As other authors may find fast implementations for different values of q, which may not scale linearly, the differentiation of these operations is reasonable. On the other hand, every method using windows will face the need to rise an intermediate value to a power equal to the window size. Hence, the algorithms examined within this thesis will all face the appearing of q-steps.

Now that the operational costs can be determined and results about the number of arithmetical operations can be stated, weight functions for the real (computational) costs must be established.

The set of commands, that a computer has to execute in order to compute the result of an arithmetical operation, may be quite different for the two arithmetical operations "addition" and "q-step". Section 1.1.1 already showed that the doubling of a point on an elliptic curve is a different task

involving different operations and therefore different costs than a normal addition of two different points. In the following example 2.4, more cases will be shown, where the costs of the squaring and of the multiplication differ substantially.
To take this fact into account, the individual costs of the operations can be included in the analyses as weight functions. The exact cost can then be inserted for special applications. These costs may then be arithmetical costs or hardware restrictions.

Definition 2.3 (operation dependent cost measures)
Let $c(Q(q))$ denote the cost of a q-step with $c(Q)$ being used synonymously for $c(Q(2))$ and let $c(A)$ denote the cost of an addition step.
These measures may indicate the costs in the underlying semigroup of an addition chain as well as the costs in the underlying semigroup of exponentiation operations. They can be used to get exact results about the costs of the addition chain methods for chosen exponentiation problems.
Let $c(I)$ denote the cost of an inversion.
For an addition chain $\chi(e_0, e_1, \ldots, e_k)$, let

$$c(\chi(e_0, \ldots, e_k)) := \sum_{q \geq 2} \big(c(Q(q)) \cdot Q_{\chi(e_0,\ldots,e_k)}(q)\big) + c(A) \cdot A_{\chi(e_0,\ldots,e_k)}$$

denote the costs of the addition chain.

With these measures and weight functions defined, some practical examples can be examined. The measure I corresponding to $c(I)$ will be introduced in definition 3.2, but in order to get the following examples complete, the costs of this operation are already listed.

Example 2.4:

Consider an exponentiation problem $\Pi = (x \in H, e \in \mathbb{N})$. The following cases are practical examples for choices of H and give an overview over the costs for the operations in the semigroup H:

(1) Consider an implementation of the RSA cryptosystem (see [Sti95] 4.3), that uses exponentiation of arbitrary integers x modulo a huge integer N (of 512 or more binary digits). The semigroup H for this exponentiation problem is

$$H = \mathbb{Z}_N = \mathbb{Z}/N\mathbb{Z}.$$

In this semigroup, the problem to square a value x has the same cost as to multiply two values $x \cdot y$, if no special representation of the integers x, y is chosen. To compute the inverse of an element x modulo N, provided that it exists, the Extended Euclidean Algorithm reveals costs of $O(M \cdot \log(\lambda_2(N)))$, where M denotes the cost of a multiplication (see [GG99], §11.1). Hence it is

$$c(A) = c(Q(2)) = 1$$
$$c(I) = \log(\lambda_2(N)).$$

(2) As a second example, consider an elliptic curve cryptosystem as depicted in protocol 5 on page 8. Here, the problem arises to compute sums of points P and Q on an elliptic curve and to compute multiples $2 \cdot P$. The costs for these two operations depend heavily on the chosen representation of the elliptic curve. Assume the points P and Q be given in affine coordinates and Optimized Extension Fields (OEF) be used for finite field representation as the field underlying the elliptic curve (see [Sma00] for more details). Then according to N. P. Smart, running times measured on his computational environment took $47\mu s$ for an affine addition and $59\mu s$ for an affine doubling. As these numbers represent the ratio between the two basic operations and as the inversion on an elliptic curve is free (see the presentation of elliptic curve arithmetic in the motivation), it is possible to use those as weight functions and hence

$$\begin{aligned} c(A) &= 47 \\ c(Q(2)) &= 59 \\ c(I) &= 0 \end{aligned}$$

and additions are faster than doublings.

(3) Similarly to example (2), the same paper of N. P. Smart suggests the use of an elliptic curve cryptosystem using Generalized Mersenne Primes p of the form $p = f(2^k)$ (see [Sma00]) to form prime fields $\mathbb{F}_p$ as underlying fields for the elliptic curves (as it is recommended in standards from such bodies as ANSI, NIST, SECG and WAP). Suppose the elliptic curve points P and Q to be in mixed coordinates. Then time measures of N. P. Smart show that a mixed addition requires $100\mu s$ and a mixed doubling requires $60\mu s$, while inversions on elliptic curves are free again. With these results, it is

$$\begin{aligned} c(A) &= 100 \\ c(Q(2)) &= 60 \\ c(I) &= 0 \end{aligned}$$

and this time doublings are faster than additions

(4) As a last example, assume a finite fields $\mathbb{F}_{2^n}$ to be given and the elements being represented in normal basis representation. In this case, the Frobenius automorphism can be computed without costs, because

it can be implemented by a cyclic shift of the coordinates. In cases like these, squaring (e.g. application of the Frobenius automorphism in the prime field $\mathbb{F}_2$) is free, hence addition chains solving an exponentiation problem in this field $H = \mathbb{F}_{2^n}$ may assume $c(Q(2)) = 0$. In [Nöc01] (table 6.3), M. Nöcker shows the arithmetic costs for multiplications to be $c(A) < 2n^3$ and according to the results stated in result 6.18 of [Nöc01], division in $\mathbb{F}_{q^n}$ can be computed with costs $c(I) < (4n^2-2n)\cdot((\log_2(n-1)+\log_2(q-1))\cdot(1+o(1))+3)$ operations in $\mathbb{F}_q$. It is therefore in $\mathbb{F}_{2^n}$

$$\begin{array}{rcl} c(A) & < & 2n^3 \\ c(Q(2)) & = & 0 \\ c(I) & < & (4n^2-2n)\cdot(\log_2(n-1)\cdot(1+o(1))+3) \end{array}$$

and multiplication is more expensive than inversions while squarings are free.

The above example shows that costs can be quite different. This may lead to different results about the superiority of one algorithm of the other. In such cases, the total costs of an algorithm can be improved by paying respect to the individual costs by choosing an exponentiation strategy, that yields the cheap operation over the other.
For this reason, the following analyses pay respect to the kind of operations needed and give individual results for the different arithmetical operations. The results will be given as exactly as possible and – for practical concerns – for the worst case and for the average case. To prevent misinterpretations, the analyses will use the following definition to state results about the average case:

Remark 2.5 (average case analysis)
When examining the average case, unless explicitly noted, it will always be assumed that all inputs are i.i.d. (independent identically distributed, within the whole thesis: distributed with the uniform probability distribution) in the set $\Omega_n := \{\ e \in \mathbb{N} \mid \lambda_2(e) = n\}$*, which is the set of all integers whose binary expansion requires exactly* n *digits.*
Especially for the binary expansion, the probability for a 1 *occurring at any position but the leftmost will be assumed to be* $\frac{1}{2}$.

2.2 The binary method

After the naive method for the computation of an addition chain for a given number $e \in \mathbb{N}$, which just creates $\{1, 2, 3, \ldots, e\}$, the binary method, also known as repeated squaring, is the oldest known method for the computation of a power x^e and therefore also for creating an addition chain for e.

This method was known to the ancient Indian mathematicians around 200 BC (see [Knu97] or [Dat35]) and to the classical Arabic mathematicians at least since the 10th century AD (see [Sai75]). In addition to that, the ancient Egyptians used a multiplication scheme quite similar to the binary method with a right to left direction of computation as early as 2000 BC (see [Str30]).

In the modern world, the binary method was a long time considered to create optimal results for addition chain lengths (see [Knu97], p. 463), which is indeed not true, even for a number as low as 15 (see example on page 23).

The binary method uses the binary expansion of a given integer e and creates an addition chain by squaring for every digit and multiplying with the base for every one in the binary expansion.

Algorithm 2.6 (exponentiation, binary method, left to right)

Input: $x, (e)_2 = (e_{\lambda_2(e)-1}, e_{\lambda_2(e)-2}, \ldots, e_0)$, $e \neq 0$

Output: x^e

```
Set A := x                    # the most significant bit is assumed to be ≠ 0
for i from λ(e) − 2 downto 0 do
    Set A := A²
    if eᵢ ≠ 0 then A := A · x
return A
```

2.2.1 Cost analysis

It is easily observed that only the number of digits determine the number of squarings, not the value of the digits themselves. Therefore, the binary method always requires $\lambda_2(e) - 1$ many squarings. This is the same in the worst case as in the average case.

The number of multiplications however is only dependent on the Hamming weight of the exponent e, as for every 1 in the binary expansion (besides the leading one), one multiplication with the base x is necessary. This leads to a consumption of $\nu_2(e) - 1$ multiplication steps. As in the addition chain constructed by algorithm 2.6, only doublings of the highest value and star steps occur, no doublings can be mistaken as multiplications (see lemma 1.5).

It is clear that $\nu_2(e)$ has the upper bound $\lambda_2(e)$, hence in the worst case $\lambda_2(e) - 1$ multiplications occur. In the average case it has to be determined how many digits can be expected to be 1. Because every digit is 1 with probability $\frac{1}{2}$ and the leading digit is 1 by default (unless $e = 0$), the expected value of the Hamming weight for a random binary expansion is $\frac{1}{2} \cdot (\lambda_2(e) -$

1) + 1, which is one more than the expected number of multiplications on average. Altogether, this gives the costs depicted in table 2.2.

	Q(e)	A(e)
exact costs	$\lambda_2(e) - 1$	$\nu_2(e) - 1$
worst case	$\lambda_2(e) - 1$	$\lambda_2(e) - 1$
average case	$\lambda_2(e) - 1$	$\frac{1}{2} \cdot (\lambda_2(e) - 1)$

Table 2.2: Costs of the binary method

In the average case, the number of squarings is twice as high as the number of multiplications. If squarings could be made faster, this would mean an important improvement to the speed of the algorithm. Another chance to improvement is to reduce the number of multiplications.

2.2.2 Changing the direction of computation

The algorithm 2.6 scans the bits of the binary expansion of e from left to right. There is also a version that scans $(e)_2$ from right to left. This version is described in [Knu97], §4.6.3, and adapts to the direction modern computers handle bit strings usually. However, this second version uses one additional multiplication and it also needs more memory than the left to right method. Although this tends not to matter anymore, because it is little compared to what standard computers offer today, the presented algorithm 2.6 should be used, if the least amount of operations is searched for.

2.3 The m-ary method and the Brauer method

The m-ary method (see [Knu97],§4.6.3), is a generalization of the binary method from the last section. The new method can handle any m-adic expansion to any base $m \in \mathbb{N}_{>1}$. The algorithm presented here assumes this representation of the exponent e to be given. With this assumption, the algorithm differs only slightly from the binary method algorithm 2.6, but it introduces the important step of precomputation. This becomes necessary, because the values of the digits are not known in advance anymore. The basic idea of improvement is that precomputation steps can save multiplications in the exponentiation step:

Algorithm 2.7 (exponentiation, m-ary method, left to right)

Input: x,$(e)_m = (E_{\lambda_m(e)-1}, E_{\lambda_m(e)-2}, \ldots, E_0)$, $e \neq 0$

Output: x^e

01 Precompute $x^0, x^1, \ldots, x^{m-1}$.
02 Set $A := x^{E_{\lambda_m(e)-1}}$
03 for i from $\lambda_m(e) - 2$ downto 0 do
04   Set $A := A^m$
05   if $E_i \neq 0$ then $A := A \cdot x^{E_i}$
06 return A

The analysis of this algorithm will be presented in the next sections.

2.3.1 Cost analysis of the precomputation step

The m-adic expansion is usually much shorter than the binary expansion, it has less digits and therefore lower costs in the main loop. The trade off for this advantage is the need to precompute the values of the digits which are needed for the multiplication in line 5. These costs have to be added to the costs of the main loop in order to determine the total costs of this algorithm and they may vary depending on the implementation. As there are two different operations (multiplications and squarings), there are two possible addition chains maximizing each of the operations.
One possibility to get all digits is to successively multiply x with itself, creating the addition chain

$$\chi(1, 2, \ldots, m-1) \quad = \quad ((0,0), (1,0), (2,0), \ldots, (m-2, 0)). \qquad (2.1)$$

This choice of χ requires only one doubling ($(0,0)$ for $x \cdot x = x^2$) and $m - 3$ additions. As only star steps occur after the initial doubling, no hidden doubling may show up, hence, the number of additions is correct (see lemma 1.5). This addition chain needs the minimum number of operations, if all m digits have to be precomputed (0 and 1 do not require an operation). The following algorithm implements the precomputation using the above addition chain. Let $v = (v_0, v_1, \ldots, v_{m-1})$ be the vector in which the values $v_i = x^i$ for $0 \leq i < m$ are being stored (v corresponds to the semantics of $\chi(1, 2, \ldots, m-1)$):

Algorithm 2.8 (Precomputation with maximum multiplications)

Set $v_0 := 1$
Set $v_1 := x$
for i from 2 to m-1 do
    $v_i := v_{i-1} \cdot x$

However, if the cost for a squaring is lower than the cost of a multiplication, an addition chain preferring doublings would be more feasible and such a chain is of course also possible. The following addition chain for the precomputation step computes all even multiples of 1 using doublings. It is

$$\begin{aligned} \chi(1,2,\ldots,m-1) &= ((j(1),k(1)),(j(2),k(2)),\ldots,(j(m-1),k(m-1))) \\ & \quad \textit{with} \\ (j(i),k(i)) &= \begin{cases} (i-1,0) & \textit{if } i \textit{ is even} \\ \left(\frac{i-1}{2},\frac{i-1}{2}\right) & \textit{if } i \textit{ is odd} \end{cases} \end{aligned} \tag{2.2}$$

This addition chain needs $\frac{m-2}{2}$ doublings and $\frac{m-2}{2}$ additions if m is even, and $\lfloor\frac{m-2}{2}\rfloor+1=\lceil\frac{m-2}{2}\rceil$ doublings and $\lfloor\frac{m-2}{2}\rfloor$ additions if m is odd, therefore there are always $\lceil\frac{m-2}{2}\rceil$ doublings and $\lfloor\frac{m-2}{2}\rfloor$ additions necessary. Again, the total number of operations, $m-2$, is minimal. Note that no hidden doubling may occur as the needed values are created one after the other in a strictly monotonous way.
The following algorithm implements the precomputation step using the above addition chain (let v be explained as above):

Algorithm 2.9 (Precomputation with maximum squarings)

```
Set v_0 := 1
Set v_1 := x
for i from 2 to m-1 do
    if (i mod 2 == 0)
        then v_i := (v_{i/2})^2
        else v_i := v_{i-1} · x
```

Theoretically, it may be helpful to observe that only those digits have to be precomputed, which really appear in the m-adic expansion of e. But in practice, if long representations of e with some hundred digits occur, the chance of saving operation in the precomputation diminishes rapidly, making such efforts to save time needless. It may only be applicable where possible inputs of e follow certain patterns. But then, precomputation may be made in advance for all following exponentiations.
Because the costs for squarings and multiplications depend on the underlying group and vary significantly, the number of the different operations should be adjusted to the algorithm that provides best costs. Hence, these numbers of operations should be introduced into the cost analysis as functions:

Definition 2.10 (costs of precomputation addition chain $\gamma(m)$)
In the precomputation step, all values within $1,\ldots,m-1$ *for some* $m\in\mathbb{N}$

must be computed in one addition chain. The window values x^k for $1 \leq k < m$ are then computed according to that addition chain.
Let

$$\Gamma(m) \quad := \quad \{ \; \chi(1, 2, \ldots, m-1) \; \}$$

denote the set of all possible addition chains χ, that compute all values needed in the precomputation step. Elements from $\Gamma(m)$ will usually be denoted as $\gamma(m)$.
Analogously to definitions 2.1 and 2.2, let $A_{\gamma(m)}$ denote the number of additions and $Q\gamma(m)(q)$ denote the number of q-steps in an addition chain $\gamma(m) \in \Gamma(m)$.
Analogously to definition 2.3, let

$$c(\gamma(m)) \quad := \quad c(Q(q)) \cdot Q_{\gamma(m)}(q) + c(A) \cdot A_{\gamma(m)}$$

denote the costs of the addition chain $\gamma(m) \in \Gamma(m)$.

The results are combined in the following lemma. As the precomputation doesn't depend on the input, the stated results apply for all, the exact analysis, the worst case and the average case, if m is fixed.

Lemma 2.11 (costs of the precomputation of the m-ary method)

The precomputation of the m-ary method for a given base $m \in \mathbb{N}$ requires the following operations, e.g. there exist two addition chains $\gamma_1(m), \gamma_2(m) \in \Gamma(m)$, such that:

choosing minimal squarings:	$Q_{\gamma_1(m)}(2)$	$=$	1
	$A_{\gamma_1(m)}$	$=$	$m-3$
choosing maximal squarings:	$Q_{\gamma_2(m)}(2)$	$=$	$\lceil \frac{m-2}{2} \rceil$
	$A_{\gamma_2(m)}$	$=$	$\lfloor \frac{m-2}{2} \rfloor$

Proof:

The algorithms 2.8 and 2.9 require the stated costs. The corresponding addition chains have been introduced in the remarks introducing the two algorithms. With $\gamma_1(m)$ defined as the addition chain depicted in (2.1), and $\gamma_2(m)$ defined as the addition chain depicted in (2.2), the result is shown. $\square$

2.3.2 Cost analysis of the exponentiation step

The exponentiation step of the m-ary method depends upon the knowledge of the precomputed window values $x^0, x^1, \ldots, x^{m-1}$. If those are known, the analysis of the main loop is completely analogous to the binary method.

Lemma 2.12 (costs of the exponentiation of the m-ary method) *The exponentiation step of the m-ary method requires the following costs:*

measure	exact analysis	worst case	average case
$Q(m) =$	$\lambda_m(e) - 1$	$\lambda_m(e) - 1$	$\lambda_m(e) - 1$
$A =$	$\nu_m(e) - 1$	$\lambda_m(e) - 1$	$\frac{m-1}{m} \cdot (\lambda_m(e) - 1)$

Proof:

The number of exponentiations with m, which appear in line 4 of algorithm 2.7 are performed once for every execution of the loop. The number of runs through the loop doesn't depend on the configuration of $(e)_m$, but only on the length. Therefore, there are $\lambda_m(e) - 1$ such exponentiations needed in any case.

The number of multiplications however depends only on the configuration of $(e)_m$. Line 5 of algorithm 2.7 accounts for a multiplication whenever the current m-adic digit is nonzero. As the leftmost digit, which is required to be nonzero, is used for initialization of the accumulator A, and the total number of nonzero digits is given as the Hamming weight of the m-adic expansion of e, $\nu_m(e) - 1$ operations are necessary. In the worst case, this number can go up to $\lambda_m(e) - 1$, if no zero digits occur.

Note that by concatenating the precomputation and the main exponentiation, no hidden doublings occur in line 5 of algorithm 2.7. The reason for this is, that the first window value used for initializing the accumulator is $\mathsf{A} = x^i$ with $i \geq 1$ (because the leftmost digit is assumed to be nonzero) and it is first raised to its m-th power, giving a value of $\mathsf{A} = x^j$ with j at least $m \cdot 1 = m$. But this value of the accumulator is not equal to any window value, which is bound to be x^k with $1 \leq k \leq m-1$ (assume $x \notin \{0, 1\}$). After that multiplication (if it occurs at all), the power of x represented by the accumulator A only grows, which prevents the chance that it may ever become a window value again. Hence, no hidden doublings can be created by line 5.

For the average case, every possible window value is equally likely for any but the leftmost window. As there are m different possible window values, an expected $\frac{m-1}{m} \cdot (\lambda_m(e) - 1) + 1$ windows are nonzero on average, which gives one more than the average number of multiplications, because the leading window still doesn't require an operation. □

2.3.3 Cost analysis of an m-step

Additional costs arise from the need of raising the accumulated partial result to an m-th power (line 4 in algorithm 2.7). These costs have an upper bound derived from the binary method for exponentiation with m, giving costs of $\lambda_2(m) - 1$ squarings and $\nu_2(m) - 1$ multiplications. But faster implementations may be possible. In many practical applications, the same value of m will be used many times. This leaves room to compute a more efficient direct formula for exponentiation with m, which can reduce the number of operations compared to the binary method. One obvious way is to compute the optimal addition chain for m if m is not too large. Because of this, the cost for this exponentiation with m is also introduced as a function into the total cost analysis:

Definition and lemma 2.13 (costs of performing an m-step $Q(m)$)

The task of raising an accumulated value to its m-th power is assumed to be done by an addition chain $\chi(m)$. If there exist fast implementations of performing this task directly, m-steps may be allowed, thus the addition chain χ could be replaced by an m-addition chain $\chi(m, m)$, that will just consist of $((0, -m))$ with the semantics $S(\chi(m, m)) = \{1, m\}$.
In the usual case, $Q_{\chi(m)}(q)$ denotes the number of q-steps to compute an addition chain for m and $A_{\chi(m)}$ denotes the number of additions. Again, $Q_{\chi(m)}(q)$ also represents the number of exponentiations with q and $A_{\chi(m)}$ represents the number of multiplications for the computation of x^m for any x using the addition chain $\chi(m)$.
If as usual addition chains only allow 2-steps (doublings), the task can be performed with the following costs:

$Q_{\chi(m)}(2)$	$=$	$\lambda_2(m) - 1$
$A_{\chi(m)}$	$=$	$\nu_2(m) - 1$

In the special case where $m = 2^d$ for some $d \in \mathbb{N}$, this table simplifies to $Q_{\chi(m)}(2) = d$ and $A_{\chi(m)} = 0$, which is optimal for such inputs.

Proof:

The stated results origin from the application of the binary method as stated in table 2.2. In the case where $m = 2^d$, the optimal addition chain for 2^d can be used, consisting of d doublings only, with the semantics $\{1, 2, 2^2, 2^3, \ldots, 2^d\}$. For such values of m, the binary method is optimal. □

2.3.4 Complete cost analysis of the m-ary method

This leads to the following overview of costs.

Theorem 2.14 (Complete cost analysis, m-ary method)
Let $Q(m)$, $Q(2)$ and A denote the operations needed by the m-ary method (algorithm 2.7). Let $\gamma(m)$ denote any addition chain from $\Gamma(m)$ to precompute the values $(0, 1, \ldots, m-1)$ and let $\chi(m)$ denote any addition chain to perform the exponentiation with m in step 4 of algorithm 2.7. The exponentiation with m can be done with one step, if the addition chain $\chi(m)$ is an m-addition chain. The exact result can then be depicted from table 2.6.

	$Q(m)$	$Q(2)$	A
exact costs	$\lambda_m(e) - 1$	$Q_{\gamma(m)}(2)$	$\nu_m(e) - 1 + A_{\gamma(m)}$

Table 2.6: Exact costs of the m-ary method

This result can be refined, if only squarings and multiplications (doublings and additions) are allowed. In this case, the m-steps are replaced by squarings and additions according to lemma 2.13. In the same way as the binary method showed, the number of squarings in the exponentiation step only depends on the length of the input, while the number of multiplications is varying depending on the configuration of $(e)_m$. The refined cost overview is depicted in table 2.7.

the m-ary method cost analysis	
exact costs	
$Q(2) =$	$(\lambda_m(e) - 1) \cdot Q_{\chi(m)}(2) + Q_{\gamma(m)}(2)$
$A =$	$(\lambda_m(e) - 1) \cdot A_{\chi(m)} + \nu_m(e) - 1 + A_{\gamma(m)}$
worst case costs	
$Q(2) =$	$(\lambda_m(e) - 1) \cdot Q_{\chi(m)}(2) + Q_{\gamma(m)}(2)$
$A =$	$(\lambda_m(e) - 1) \cdot A_{\chi(m)} + \lambda_m(e) - 1 + A_{\gamma(m)}$
average case costs	
$Q(2) =$	$(\lambda_m(e) - 1) \cdot Q_{\chi(m)}(2) + Q_{\gamma(m)}(2)$
$A =$	$(\lambda_m(e) - 1) \cdot A_{\chi(m)} + \frac{m-1}{m} \cdot (\lambda_m(e) - 1) + A_{\gamma(m)}$

Table 2.7: Costs of the m-ary method

Proof:

In table 2.6, the result of lemma 2.12 is combined with the general statement about the precomputation step (definition 2.10), while table 2.7 combines the results of lemma 2.12 with the general statements about the precomputation step (definition 2.10) and the exponentiation with m (definition 2.13).

It has still to be shown that by concatenating the three addition chains (for the precomputation, for the m-step and for the exponentiation step) no hidden doublings are created. To show this, assume that the m-step addition chain $\chi(m)$ does not contain a hidden doubling (for example, use the binary method). Then the assumption that $\chi(e)$ contains a hidden doubling contradicts this assumption. It may be assumed that the precomputation step computes a new window value with every step, which prevents hidden doublings (this assumption is justified by the existence of such addition chains shown in section 2.3.1) and it has been shown explicitly for the multiplication in line 5 in lemma 2.12, hence, a possible hidden doubling must origin from the m-step addition chain $\chi(m)$ used for raising the accumulator to its m-th power.

Let $(j(i), k(i))$ with $j(i) \neq k(i)$ and $a_{j(i)} = a_{k(i)}$ be such a hidden doubling. It occurs while raising A to its m-th power. This is achieved by taking the exponent of x represented by A in place of the usual start value 1 of $\chi(m)$ and perform $\chi(m)$ on it. During the m-step, only that value of A and values created within the m-step addition chain are referenced, hence, a hidden doubling corresponds to a hidden doubling in the original $\chi(m)$, which doesn't exist by assumption. Therefore, no hidden doublings occur. This proof becomes very easy, if m is a power of 2 and the m-step only consists of squarings.

□

Example:

If $m = 2$, the m-ary method should give the known results of the binary method. And this is the exact result when looking at the cost analysis for $m = 2$:
In the precomputation for $m = 2$, only $x^0 = 1$ and $x^1 = x$ have to be computed, hence no costs arise, $Q_{\gamma(m)}(2) = A_{\gamma(m)} = 0$.
The exponentiations with $m = 2$ on line 4 of this algorithm are squarings, so the costs of the squarings, measured in squarings and multiplications, are quite trivial:

$$Q_{\chi(2)}(2) = 1, A_{\chi(2)} = 0.$$

Therefore the exact total costs are $(\lambda_2(e) - 1) \cdot 1 = \lambda_2(e) - 1$ squarings and $\nu_2(e) - 1$ multiplications, verifying the result of the analysis of the binary method in the last section.

2.3.5 Brauer's method – a practical restriction

One problem has not been addressed in the analysis yet, the computation of the m-adic expansion $(e)_m$ of e. In practical applications, these costs may be significant, because numbers are usually stored in a computer as a 2^k-ary expansion of e, for some $k \in \mathbb{N}$.
To adapt to this property, which makes the m-ary method very useful in practice, the suggestions by A. Brauer in [Bra39] are used. Brauer restricts the algorithm to bases which are powers of 2, e.g. $m = 2^d$ for some $d \in \mathbb{N}$. In the literature, this restriction is often assumed, while still the name "m-ary method" is used, the m often changes its meaning from the base of the m-ary method to the power of 2 of Brauer's restriction.
This restriction, or Brauer's method, is also very close to the computer's inside reality, because the transformation from the binary expansion into a 2^d-adic expansion can be done free of costs, because simply d consecutive bits have to be viewed together for a digit representation. Modern computers do this anyway, in order to gain speed they're not handling bit after bit but word after word, each word consisting of many bits (32 or 64 bits is today's usual computer word size). In this context, d determines the *window size* when looking at the binary expansion of e.
Although, if the given context of the exponentiation problem Π offers the use of the Frobenius automorphism, it may be a great advantage to choose

a different number representation in order to use that property. In this case, it is useful to choose a suitable base for the representation and to build up windows consisting of d of such digits.
In a correct cost analysis, the costs to create this representation could be included as an additional term in the sum of operations needed. But as this transformation doesn't require arithmetical operations, which are the sole interest of the cost considerations, these costs are omitted.
In the remaining part of the thesis, Brauer's restriction is applied. This leads to a simplified cost analysis:

Corollary 2.15 (cost analysis for the Brauer method)
Let $\gamma(2^d) \in \Gamma(2^d)$ denote any addition chain suitable for the precomputation step. Then for the Brauer restriction, where $m = 2^d$ for some $d \in \mathbb{N}$, the costs depicted in table 2.8 can be assumed.

the Brauer method cost analysis	
	exact costs
$Q(2) =$	$\left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) \cdot d + Q_{\gamma(2^d)}(2)$
$A =$	$\nu_{2^d}(e) - 1 + A_{\gamma(2^d)}$
	worst case costs
$Q(2) =$	$\left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) \cdot d + Q_{\gamma(2^d)}(2)$
$A =$	$\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1 + A_{\gamma(2^d)}$
	average costs
$Q(2) =$	$\left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) \cdot d + Q_{\gamma(2^d)}(2)$
$A =$	$\frac{2^d-1}{2^d} \cdot \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) + A_{\gamma(2^d)}$

Table 2.8: Costs of the Brauer method

Proof:

The replacement of $Q_{\chi(2^d)}(2)$ by d and $A_{\chi(m)}$ by 0 has been shown in lemma 2.13. And it is

$$\lambda_{2^d}(e) = \left\lceil \frac{\lambda_2(e)}{d} \right\rceil .$$

□

2.3.6 Determining an optimal value for the window size d

Obviously, the number of arithmetical operations needed depends on the window size d. For large d, the costs in the exponentiation step are low, while the precomputation requires a lot of operations. If d is small, the ratio is vice versa. In order to reduce the overall costs of the computation of an addition chain to the minimum, this value should be chosen carefully and optimal. The choice has to be done dependent on the bit length $\lambda_2(e)$ of the exponent e.
In the literature, the window length d has always been optimized with respect to the total number of arithmetical operations (squarings plus multiplications). Although examples 2.4 showed significant differences in the real costs of the arithmetical operations, most optimization efforts were spent to optimize applications computing in some $\mathbb{F}_{q^n}$, where $Q = A$ (see example 2.4(1)). It is not the intent of this thesis to extend these results, hence, this starting point for optimization of d is also carried out.
According to the cost analysis and definition 1.2, the overall number of arithmetical operations will be denoted by σ in the following way:

$$\sigma := \big(Q_{\gamma(m)} + A_{\gamma(m)}\big) + (\lambda_m(e) - 1) \cdot \big(Q_{\chi(m)} + A_{\chi(m)}\big) + \nu_m(e) - 1$$

Ordinary addition chains only allow squarings and multiplication, hence the optimization for d should pay respect to that fact and replace all measures by those using only squarings and multiplications. Therefore, it is assumed that the precomputation is implemented by algorithm 2.8, that computes all digits (besides x^2) by successive multiplication with x and it is assumed that the exponentiation with m is implemented by the binary method, using the binary method addition chain for m in order to compute x^m:

$$\Longrightarrow \ \sigma \ = m - 2 + (\lambda_m(e) - 1) \cdot (\lambda_2(m) - 1 + \nu_2(m) - 1) + \nu_m(e) - 1$$

If $m = 2^d$, then $\lambda_{2^d}(e) = \lceil \frac{1}{d} \cdot \lambda_2(e) \rceil$, $\lambda_2(2^d) = d + 1$ and $\nu_2(2^d) = 1$, and therefore

$$\sigma = 2^d - 3 + \left(\left\lceil \tfrac{\lambda_2(e)}{d} \right\rceil - 1\right) \cdot d + \nu_{2^d}(e).$$

In the average case, and this should be the case d should be optimized for, it is $\nu_m(e) = \frac{m-1}{m} \cdot (\lambda_m(e) - 1)$ and hence

$$\begin{aligned} \sigma &= 2^d - 3 + d \cdot \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) + \frac{2^d - 1}{2^d} \cdot \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) \\ &= 2^d - d + \left(\frac{2^d - 1}{2^d} + d\right) \cdot \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) - 3. \end{aligned} \tag{2.3}$$

This function has a unique global minimum, but it is very difficult to compute an explicit analytical solution. Additionally, in practice, integer solutions are needed, which requires an approximation.

$\lambda_2(e)$	**formula (2.5)**	**rounded (2.5)**	**formula (2.4)**
8	1.81261743600	2	0
20	2.36590733400	2	1
33	2.70350182800	3	1
67	3.22038722400	3	1
128	3.73023405800	4	2
155	3.88733092400	4	2
192	4.06629740400	4	2
256	4.31203116400	4	3
307	4.47018933800	4	3
419	4.74609241800	5	3
512	4.92721911400	5	3
555	5.00079525600	5	3
1024	5.57224763000	6	4
1145	5.67875096400	6	4
2048	6.24388051400	6	5

Table 2.9: Comparison of some suggested optimal values for d

Because of this, a good value for d can be found by exhaustive search or using heuristics. The most common choice for d is attributed to Brauer [Bra39] himself and states

$$d = \lfloor \log_2 \lambda_2(e) - 2 \cdot \log_2 \log_2 \lambda_2(e) \rfloor + 1. \tag{2.4}$$

A slightly different value has been proposed by Gordon in [Gor98] as

$$d = \log_2 \log_2 e - 2 \log_2 \log_2 \log_2 e.$$

However, the above values originate in the aim to prove an upper bound to $l(e)$, the lowest number of operations in any addition chain for an integer e, therefore, they do not state the optimal value for d. Although, in practical applications, it is common to determine d according to either of the above formulas and also to try the next two integers to either side to ensure optimal choices.

A new and better approach was suggested by M. Nöcker in [Nöc01], who used Lambert's W function $W(x) = w$, which is defined as the solution w of the equation $w \cdot exp(w) = x$. He shows that the problem of optimizing d can be reduced to an equation of this type. This leads to the global extrema for (2.3) as

$$d' = \frac{2}{\ln 2} \cdot W\left(\frac{1}{2}\sqrt{\frac{\lambda_2(e) \cdot \ln 2}{\frac{c+1}{2}}}\right), \tag{2.5}$$

$\lambda_2(e)$	d_1	operations	d_2	operations	d_{opt}
8	1	10.4952	2	10.2479	2
20	1	28.4965	2	26.7452	2
33	1	47.9852	3	44.7517	3
67	1	98.9879	3	91.2488	3
128	2	175.262	4	167.065	4
155	2	213.748	4	201.627	4
192	2	263.254	4	246.065	4
256	3	335.384	4	325.069	4
307	3	401.251	4	389.24	4
419	3	544.623	5	525.409	5
512	3	664.768	5	638.816	5
555	3	718.997	5	686.561	5
1024	4	1273.07	6	1249.35	(5, 1247.62)
1145	4	1426.13	6	1389.03	6
2048	5	2471.23	6	2443.67	6

Table 2.10: Comparison of the numbers of operations for some values of d

where $c \geq 0$ is a constant describing the ratio

$$c = \frac{c(Q)}{c(A)}.$$

The rounded value of d', e.g. $d = \mathsf{round}(d')$ can then be used for the Brauer method with better results than formula (2.4). Table 2.9 shows some values for d computed with formulas (2.4) and (2.5), table 2.10 shows average results for the number of arithmetical operations for the two suggested choices of d. Note that d_1 is the value computed by formula (2.4), d_2 is the value computed by formula (2.5) with the assumption of $c(A) = c(Q)$ and d_{opt} is the lowest number of operations for choices of $d_1 - 1 \leq d \leq d_2 + 1$ (where $d_1 - 1$ was set to 1 if $d_1 - 1 < 1$).
The experimental data has been created using 10^5 sets of random binary expansions with the stated length, where the leftmost digit has been set to 1 in order to assure that all inputs have the same length. The hardware was a 733MHz PC with Intel architecture, the random bits have been generated with the C++ rand() function as

$$e_i := \mathsf{(int)(2.0 * ((double)rand()/(double)(RAND\_MAX + 1)))}.$$

Chapter 3

Addition-subtraction chains

This chapter will introduce addition-subtraction chains and a number representation, the non-adjacent form (NAF), which is well suited as a base for addition-subtraction chain algorithms. The NAF will be analyzed to great extend, as the analyses of the addition-subtraction variants of the binary method and the Brauer method, which will be presented in chapter 4, require extensive knowledge about properties of this special representation.

All algorithms presented so far deal with additions and doublings only, which lead to multiplications and squarings in the context of exponentiation.
Subtractions, which lead to divisions, always require the explicit computation of a multiplicative inverse, which is a costly operation in general rings (e.g. $O(n^2)$ bit operations for modular inversion using classical arithmetic, see [GG99], p. 232 and §11.1). For that reason, subtractions are hardly usable in a general application of addition-subtraction chains.
One could imagine to introduce subtraction by starting an addition chain not with the value 1 (for x), but with -1 and 1 (for x^{-1} and x). Then all inverses could be computed from the element -1 without computing further inversions. This may help in some cases, where only -1 is needed (algorithm 4.11 is such an example), but this approach leaves out the chances that the new operation offers. It is only helpful if inversion is an expensive operation and should not be used often.
However, in some cases, like elliptic curves, the needed inverses can be computed very fast – in fact, when using elliptic curves, theses inverses are completely free (see definition 2 on page 7 and examples 2.4(2) and (3)). In cases like these, inversion can lead to improved results: addition chains turn to addition-subtraction chains.

Definition 3.1 (addition-subtraction chain $\bar{\chi}(e)$)
An addition-subtraction chain $\bar{\chi}$ is a sequence

$$\bar{\chi} = ((j'(1), k'(1)), \ldots, (j'(r), k'(r))),$$

of pairs of possibly overlined non-negative integers $(j(i), k(i))$ with

$$\begin{array}{ll} j'(i) \in \{\ j(i),\ \overline{j(i)}\ \}, & k'(i) \in \{\ k(i),\ \overline{k(i)}\ \} \\ 0 < k(i) \leq j(i) < i & \forall\ 1 \leq i \leq r. \end{array}$$

The number r of pairs is the length $L(\bar{\chi})$ of the addition chain $\bar{\chi}$.
The semantics of $\bar{\chi}$ is defined to be the set $S(\bar{\chi}) = \{a_0, a_1, \ldots, a_r\}$ of integers such that

$$\begin{array}{rcll} a_0 & = & 1 & \\ a_i & = & a_{j'(i)} + a_{k'(i)} & \forall\ 1 \leq i \leq r. \\ & \textit{with} & & \\ a_{\bar{s}} & = & -a_s & \end{array}$$

The tupel $(j'(i), k'(i))$ is said to produce the element a_i of the semantics. In the case where $j'(i) \neq k'(i)$, the operation is called an addition, in the case where $j'(i) = k'(i)$, the operation is called a doubling. If $j'(i) = \overline{j(i)}$ or $k'(i) = \overline{k(i)}$, the inverse for the addition is taken for the operation, e.g. whenever there's an overlined $j(i)$ or $k(i)$, an inversion has to take place before the addition or doubling can be carried out. The computation of these inverses does not contribute to the length of the addition-subtraction chain. For any set of numbers $E = \{e_0, e_1, \ldots, e_k\} \subseteq S(\bar{\chi})$, $\bar{\chi}$ is called an addition-subtraction chain for E (an addition-subtraction chain for $e_0, \ldots, e_k$) and may be denoted as $\bar{\chi}(e_0, e_1, \ldots, e_k)$. $\bar{\chi}(e)$ denotes an addition-subtraction chain for a single number $e \in S(\bar{\chi}(e))$.

Example:

The following is an addition-subtraction chain for $e = 219$.

$$\begin{array}{rcl} \bar{\chi}(219) & = & ((0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,\overline{5}), \\ & & (9,\overline{2}),(10,\overline{0})) \\ S(\bar{\chi}(219)) & = & \{1,2,4,8,16,32,64,128,256,224,220,219\} \\ L(\bar{\chi}(219)) & = & 11 \end{array}$$

Note that the semantics has been noted in the same order as the elements are created.

Note that unlike most authors (for example [MO90], [KT92], [Len93] and [Gor98]), this thesis includes the computation of the inverse in the definition as an explicit operation, rather than just allowing $+$ and $-$ to be used to combine two elements, because this is what really has to be done for a subtraction and it helps to analyze applications where the computation of inverses may have very different costs. In this, the definition is more general than the usual notation, although the same semantics is created.

In order to count the number of arithmetical operations within an addition-subtraction chain, the following measures are introduced similar to the measures A and $Q(q)$ in section 2.1.

Definition 3.2 (number of operations)
The number of inversion steps that create an additive inverse to be used for the creation of a new element within an addition-subtraction chain $\bar{\chi}(e)$ for a number $e \in \mathbb{N}$ will be denoted as $I_{\bar{\chi}(e)}$. This number is equal to the number of divisions used to solve an exponentiation problem $\Pi = (x \in H, e \in \mathbb{N})$, that presents the task to raise x to its $e-th$ power x^e using this addition-subtraction chain $\bar{\chi}(e)$.
If a whole algorithm is to be analyzed, the total number of inversions is denoted as I.
Analogously to the definition of the measures $Q(q)$ and A, let $Q_{\bar{\chi}(e)}(q)$ denote the number of q-steps and let $A_{\bar{\chi}(e)}$ denote the number of addition steps within the addition-subtraction chain $\bar{\chi}(e)$. Let A and Q denote the total number of such operations within a whole algorithm.

Similarly to definition 2.3, the cost measure for an addition-subtraction chain is defined equivalently to that for addition-chains:

Definition 3.3 (operation dependent cost measure for $\bar{\chi}(e)$)
For an addition-subtraction chain $\bar{\chi}(e_0, e_1, \ldots, e_k)$, let

$$\begin{aligned} c(\bar{\chi}(e_0, \ldots, e_k)) \quad := & \sum_{q \geq 2} \big(c(Q(q)) \cdot Q_{\bar{\chi}(e_0,\ldots,e_k)}(q) \big) + c(A) \cdot A_{\bar{\chi}(e_0,\ldots,e_k)} \\ & + c(I) \cdot I_{\bar{\chi}(e_0,\ldots,e_k)} \end{aligned}$$

denote the costs of the addition-subtraction chain.

3.1 Chances and limitations of addition-subtraction chains

While the computation of shortest addition chains is presumably an NP-complete problem (as explained on page 21), their lengths can be bounded. While a trivial upper bound can be derived using the result of the analysis of the binary method (see section 2.2), there is also a lower bound that has been presented by Arnold Schönhage in [Sch75] (see theorem 1.9). For the length $l(e)$ of a shortest addition chain for a given positive integer e the following equation holds:

$$\log_2 e + \log_2 \nu_2(e) - 2.13 \leq l(e) \leq \lambda_2(e) - 1 + \nu_2(e) - 1 \qquad (3.1)$$

The tightness of the lower bound has been depicted in figure 1.1. Especially for numbers $e = 2^{2^n} - 1$, where addition chains of length $2^n + n - 1 \approx \log_2(e) + \log_2(\nu_2(e)) - 1$ exist, improvements are limited (see [Sch75], p. 2).

There have also been other upper bounds, e.g.

$$l(n) \leq \log_2 e + \frac{\log_2 e}{\log_2 \log_2 e} + o\left(\frac{\log_2 e}{\log_2 \log_2 e}\right),$$

taken from [Bra39], but other upper bounds can also be derived from any of the presented algorithm within this thesis.
The introduction of the subtraction operation now yields the immediate result, that addition-subtraction chains are capable of reducing the optimal length. In the sequence $(l(e))_{n\in\mathbb{N}}$ (see table in appendix A), there are entries e with $l(e) > l(e+1)+1$, see table 3.1 for some examples.

e	$l(e)$	e	$l(e)$	e	$l(e)$
47	8	127	10	159	10
48	6	128	7	160	8

Table 3.1: Some examples of $l(e) > l(e+1)+1$

In any such case, the use of subtraction yields shorter addition-subtraction chain for e using the chain for $e+1$ and subtracting 1 at the end. Therefore, for some numbers, the use of an addition-subtraction chain can yield results which are better than any addition chain.
Example:

Consider $e = 255$ and an addition chain of optimal length 10 producing the semantics shown in appendix A,

$$S(\chi(255)) = \{1, 2, 3, 5, 10, 15, 30, 60, 120, 135, 255\},$$

and the following semantics of an addition-subtraction chain of length 8:

$$S(\bar{\chi}(255)) = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 255\}.$$

The semantics has been noted in this way to show the last step of the addition-subtraction chain to be $(8, \overline{0})$. As one might guess, the most impressive examples are of the form $e = 2^k - 1$ for some $k \in \mathbb{N}$, because doubling is the fastest way to increase the numbers, therefore, powers of 2 have the shortest chains possible.

According to Schönhage, the stated lower bound (3.1) also holds for addition-subtraction chains, if the Hamming weight is replaced by the arithmetic weight $w(e)$ (the minimal possible Hamming weight for signed binary digit representations), that will be defined in definition 3.6. The minimal Hamming weight is necessary, because signed binary digit representations are not unique, hence, by choosing a very long representation, $\log_2(\nu_{\bar{2}}(e))$ could exceed every limit.

For powers of 2, where the binary Hamming weight is equal to the arithmetic weight, no advantage can be expected using addition-subtraction chains. But for other numbers, whose binary expansion has more nonzero digits than a minimal signed binary representation, Schönhage's lower bound decreases, giving hope to find faster algorithms based on signed binary digit representations. This notion is the basic idea to analyze addition-subtraction chains using signed binary digit representations of minimal length. The next sections will present a possible approach.

3.2 Signed binary digit representations

One approach to addition-subtraction chains is the use of so called signed binary digit representations of the exponent e. This approach has been introduced and developed among others by Jedwab and Mitchell in [JM89], Morain and Olivos in [MO90], Koyama and Tsuruoka in [KT92] and Arno and Wheeler in [AW93]. They transform the ordinary binary representation of the exponent e into a binary representation with signed digits using the values -1, 0 and 1.
Using this approach, the authors' aim is to reduce the Hamming weight – and by this the overall number of multiplications needed in an exponentiation algorithm, that uses an addition chain based on this representation. The general algorithm 1.26 used for exponentiation from chapter 1 now has the general form

Algorithm 3.4 (general addition-subtraction chain algorithm)

01 representation of the exponent
02 partitioning of the representation into windows
03 precomputation of window values
04 exponentiation with window values present

The representation of the exponent e is most efficient if it minimizes the operations of the addition-subtraction chain method which is based on that specific representation. By now, authors have mainly concentrated on the approach to compute a canonical signed binary digit representation with the lowest possible Hamming weight, which will be presented in the next sections.
The motivation to have a closer look into addition-subtraction chains was the observation that especially long runs of ones can be represented by one division, hence reducing the Hamming weight significantly, for example 111111 can be transformed into $100000\bar{1}$, using $\bar{1}$ to denote -1. This is most effective in the case where the binary expansion of e only consists of ones – the worst case of the traditional binary method.

At first, some remarks about the notation:

Definition 3.5 (signed digit representation)
A signed digit representation $(e)_{\bar{q}}$ of a given integer $e \in \mathbb{Z}$ is a finite sequence

$$(e)_{\bar{q}} \quad = \quad (e_{k-1}, e_{k-2}, \ldots, e_0)$$

of k digits satisfying

$$e_i \in \{-(q-1), -(q-2), \ldots, -1, 0, 1, \ldots, q-2, q-1\} \qquad \forall\, i$$

with $|e_{k-1}| \neq 0$ for $e > 0$ and

$$e = \sum_{i=0}^{k-1} e_i \cdot q^i.$$

The number k of digits is the length $\lambda_{\bar{q}}(e)$ of the signed binary digit representation.
It is sometimes helpful to interpret $(e)_{\bar{q}}$ as an infinite sum of digits with all but finite many being zero. In this case, $\lambda_{\bar{q}}(e)$ can be formally defined as

$$\lambda_{\bar{q}}(e) := \begin{cases} \max\{i \in \mathbb{N}_0 | e_i \neq 0\} + 1 & for\ e > 0 \\ 1 & for\ e = 0. \end{cases}$$

For the sake of a better notation, any value for $e_i = n < 0$ will be written as $\bar{\mathsf{n}}$, especially -1 will be denoted as the digit $\bar{1}$.

In generalization to definition 1.8, the Hamming weight for signed digit representations is similarly defined. Additionally, as signed binary digit representations are not unique, the minimal Hamming weight, the arithmetical weight, is introduced.

Definition 3.6 (Hamming weight and arithmetic weight)
(following A. Schönhage in [Sch75] and J.H. van Lint in [Lin98])
The Hamming weight of a given signed q-ary digit representation $(e)_{\bar{q}}$ of a given integer e is defined as the number of nonzero digits in $(e)_{\bar{q}}$, e.g.

$$\nu_{\bar{q}}(e) \quad = \quad \sum_{e_k \neq 0} 1.$$

The minimal Hamming weight of any possible signed q-ary digit representation for e is called the arithmetic weight of e and it is denoted and defined as

$$w_q(e) := min \left\{ \sum_{e_k \neq 0} 1 \;\middle|\; e = \sum_k e_k \cdot q^k, -(q-1) \leq e_k \leq (q-1) \right\}.$$

It can easily be seen that $w_q(e) \leq \nu_{\bar{q}}(e)$. $w(e)$ denotes $w_2(e)$.

Example:

We have

$$(15)_2 = 1111 = 1000\bar{1} = 16 - 1,$$

but it is also true that

$$(15)_2 = 1111 = 10\bar{1}11 = 16 - 4 + 2 + 1.$$

Additionally, consider

$$(15)_2 = 1000\bar{1} = 1\bar{1}^*000\bar{1} = 2^n - 2^{n-1} - \ldots - 16 - 1, \tag{3.3}$$

which gives a valid signed binary digit representation for any number of digits $\bar{1}$ at the $*$.

This example demonstrates that a signed binary digit representation is no longer unique like the ordinary binary expansion. In fact, there are always infinitely many possible signed binary digit representations for any chosen number $e \neq 0$. The pattern (3.3) used for 15 in the example above can be applied to any signed binary digit representation creating infinitely many valid representations.

There is also no unique shortest signed digit representation in general. While for some values of e there is only one shortest signed digit representation, like 7, which has the unique shortest representation

$$(7)_2 = 111,$$

for many other values there are more than just one shortest representation, for example

$$(19)_2 = 10011 = 1010\bar{1}.$$

Fortunately, by putting some restrictions on the signed digit representation, it is possible to define a representation uniquely. This unique representation has the minimal possible Hamming weight, but it doesn't necessarily have the shortest possible length. The latter is not too great of a trade-off, because it will be shown that the length will increase by at most one digit. The next section will introduce the non-adjacent form (NAF), which gives this unique special signed binary digit representation.

A signed binary digit representation can always be interpreted as the subtraction of two binary numbers, the first argument containing all positive digits, the second argument containing all negative digits (see [MO90]). This means that the value of any signed digit representation can be computed using just one subtraction in $\mathbb{Z}$.

Example:

$(15)_2 = 10\bar{1}11 = (15)_{\bar{2}}^{+} - (15)_{\bar{2}}^{-} = 10011 - 100 \mathrel{\widehat{=}} 19 - 4.$

3.3 The non-adjacent form (NAF)

The need to have a uniquely defined signed binary representation with minimal Hamming weight for any given number e arises from the need to construct algorithms, which work with such a representation. Those algorithms should work efficient and, of course, deterministic.
The following definition introduces the non-adjacent form (NAF) of a given integer. It shows both of the demanded properties, uniqueness and minimal Hamming weight. The NAF has been rediscovered many times by several authors (see [Gor98], p.138) and is sometimes referred to as a special recoded binary representation (*Booth-recoding*, referencing [Boo51]) or as a *canonical signed-digit vector* (see [EgK90], p.189).

Definition 3.7 (Non-adjacent form (NAF))
For $e > 0$, any signed binary digit representation

$$(e)_{\bar{2}} = (e_{\lambda_{\bar{2}}(e)-1}, e_{\lambda_{\bar{2}}(e)-2}, \ldots, e_0),$$

$e_i \in \{-1, 0, 1\}$ for all i, with no two adjacent nonzero digits, hence

$$e_i \cdot e_{i+1} = 0 \quad \forall\, i \geq 0,$$

is called a non-adjacent form (NAF) of e. The NAF of $e = 0$ is defined as (0).
In this case, any such $(e)_{\bar{2}}$ will be denoted as $(e)_{\mathcal{NAF}}$, with length $\lambda_{\mathcal{NAF}}(e) = \lambda_{\bar{2}}(e)$.

As mentioned in definition 3.5, many authors (like [AW93], [Lin98]) explicitly use an infinite sum to describe the NAF, assuming that all but finite many values are 0. The two notations are equivalent.
Several algorithms have been suggested to create unique signed binary digit representations, most of which create the NAF, so most of them are equivalent – at least for the binary case. Such algorithms have been suggested by Jedwab and Mitchell in [JM89] Morain and Olivos in [MO90], and Arno and Wheeler in [AW93]. This thesis will present these three algorithms in the following sections.

3.3.1 Creating the NAF from the binary expansion

F. Morain and J. Olivos developed in [MO90] an algorithm to create a signed binary digit representation from the ordinary binary expansion, replacing sequences of two or more consecutive ones, also regarding isolated zeros.

The algorithm creates a representation with no two adjacent nonzero digits, hence it creates a NAF of the given number.
The binary representation $(e)_2$ of a given integer e is scanned from right to left and sequences of consecutive ones, denoted as 1^a, are transformed into a sequence of lower (or at most of the same) Hamming weight according to the following rule

$$1^a \longmapsto 10^{a-1}\bar{1}. \tag{3.4}$$

If a is chosen to create the sequence 1^a of maximal length, the transformation doesn't interfere with the rest of the representation, because the leading one is added to a zero, hence no problems with carried bits arise. But this transformation is only optimal with a slight enhancement: While the transformation is applied from right to left, the newly created leading 1 of a replaced sequence must be taken into account when searching for the next sequence of consecutive ones. In the case of isolated zeros, this enhancement produces one nonzero digit less than rule (3.4) alone. While Morain and Olivos themselves saw this problem, they tried to solve it using the extra rule

$$1^a01^b \longmapsto 10^a\bar{1}0^{b-1}\bar{1}. \tag{3.5}$$

However, this rule would only correct the case of one isolated zero, therefore, taking the leading 1 into account for the next step is more suitable to deal with isolated zeros.
The NAF created by this algorithm can then be used for exponentiation algorithms, which will be examined in chapter 4.
Example:

Let $(e)_2 = 111101111$, then the application of the transition rule (3.4) leads to the signed binary digit representation

$$(e)_2 = 111101111 \longmapsto 1000\bar{1}1000\bar{1}$$

if applied to all sequences of consecuting ones at once.
Using the enhancement of determining the sequences to be replaced on the fly, the transformation works as follows:

$$(e)_2 = 111101111 \longmapsto 11111000\bar{1} \longmapsto 10000\bar{1}000\bar{1}.$$

The rule (3.4) can be easily written as an algorithm that creates the NAF. This algorithm will search the input for consecutive ones. Note that although the Hamming weight is only reduced when there are three or more consecutive ones, the transformation rule must be applied for all sequences of consecutive ones ($a \geq 2$), because otherwise the algorithm would fail in the case of isolated zeros (like 11011). Hence minimal weight can only achieved if the NAF is completely produced.

The total length of the resulting NAF is at most one digit longer than that of the ordinary binary expansion of e, e.g.

$$\lambda_{\mathcal{NAF}}(e) \in \{\lambda_2(e), \lambda_2(e)+1\},$$

a fact that will be proved in theorem 3.16. This observation has helped to construct the following algorithm. Note that it is assumed that all $e_i = 0$ for $i \geq \lambda_2(e)$.

Algorithm 3.8 (NAF transformation by Morain and Olivos)
(following F. Morain and J.Olivos in [MO90])

Input: *The binary expansion* $(e)_2 = (e_{\lambda_2(e)-1}, e_{\lambda_2(e)-2}, \ldots, e_0)$, $e > 0$
Output: *The NAF* $(e)_{\mathcal{NAF}} = (f_{\lambda_{\mathcal{NAF}}(e)-1}, f_{\lambda_{\mathcal{NAF}}(e)-2}, \ldots, f_0)$

```
Set i := 0
while ((i < λ2(e) − 1)) do
    if (e_i == 1 AND e_i+1 == 1) then
        Set e_i := 1̄
        Set i := i + 1
        while (e_i == 1) do
            Set e_i := 0
            Set i := i + 1
        Set e_i := 1
return (e)_2
```

The proof that the above algorithm really computes the NAF is part of the following theorem.

Theorem 3.9 (existence and uniqueness of the NAF)
Every integer e has a unique NAF.

Proof:

1) existence

For $e = 0$, the NAF is defined as (0).

For $e > 0$, let $(e)_2 = (e_{\lambda_2(e)-1}, e_{\lambda_2(e)-2}, \ldots, e_0)$ be the binary expansion of e, which is uniquely defined. Assume that $(e)_2$ is not yet in non-adjacent form (otherwise, there is nothing to prove). The existence of the NAF can be proven by showing that algorithm 3.8 produces the NAF from $(e)_2$. This proof is carried out using induction. Note that a transformation of the form

$$01^a \longmapsto 10^{a-1}\bar{1}$$

preserves the value of that substring.

start of induction (induction over the length of $(e)_2$):
Let $i \in \mathbb{N}$ be the minimal number such that the if-condition in line 3 of algorithm 3.8 yields true, e.g. the minimal i such that $e_i = e_{i+1} = 1$. If $(e)_2$ is not a NAF, there must be at least two adjacent nonzero digits, hence such a pair $(i, i+1)$ exists. Then the first i digits are in non-adjacent form.

assumption:
Let $k \in \mathbb{N}$ be the number of the current digit, at the time algorithm 3.8 evaluates the if-condition in line 3. Then the sequence $(e_k, e_{k-1}, \ldots, e_0)$ is in non-adjacent form. Now, $(e)_2$ can be transformed further into a sequence with a non-adjacent tail with at least $(e_{k+1}, e_k, \ldots, e_0)$ being in non-adjacent form.

inductive proof:
If the if-condition yields false, $(e_{k+1}, e_k, \ldots, e_0)$ is already in non-adjacent form and nothing needs to be shown. Otherwise, the if-condition yields true and $e_k = e_{k+1} = 1$.

The latter implies that $e_{k-1} = 0$. Therefore, with line 4 of algorithm 3.8 setting $e_k := \bar{1}$, the assumption still holds. Now let $j \geq k+1$ be the value of i at the end of the while-loop in lines 6-8, hence the maximal number $j \geq k+1$ such that $e_i = 1$ for $k+1 \leq i \leq j$ in the input. By setting all these values to zero in line 7, $e_i := 0$ for $k+1 \leq i \leq j$, the resulting sequence $(e_j, e_{j-1}, \ldots, e_{k+1}, e_k, e_{k-1})$ is in non-adjacent form.

Now as j was chosen maximal, $e_{j+1} = 0$, hence, with line 9 setting $e_{j+1} := 1$, and the assumption, it is $(e_{j+1}, e_j, \ldots, e_{k+1}, e_k, \ldots, e_0)$ in non-adjacent form.

Note that e_k was nonzero before the transformation and after the transformation. That means that if $j = k+1$, which is the minimal value for j, the Hamming weight is unchanged, as e_{k+2} was zero and changed the value with e_{k+1}. But if $j > k+1$, the Hamming weight is clearly reduced, as all values between e_{j+1} and e_k are set to zero.

2) uniqueness
Suppose e can be written as two different NAFs,

$$e = \sum_{i=0}^{a} e_i \cdot 2^i = \sum_{i=0}^{b} e'_i \cdot 2^i \tag{3.6}$$

with $(e_a, e_{a-1}, \ldots, e_0) \neq (e'_b, e'_{b-1}, \ldots, e'_0)$. Assume without loss of generality that $e_0 \neq e'_0$ (otherwise subtract e_0 and divide both representations by 2, e.g. delete e_0 and e'_0. Continue until $e_0 \neq e'_0$.)

If $e_0 = 0$ then e is divided by 2 and because $e'_0 \neq e_0$, it also means that e is not divided by 2. Therefore $e_0 \neq 0$ and $e'_0 \neq 0$.

Without loss of generality, let $e_0 = 1$ and $e'_0 = -1$ (otherwise switch). Then e is not divided by 2. But because both sums in (3.6) are NAFs, $e_1 = e'_1 = 0$, so in

$$\sum_{i=0}^{a} e_i \cdot 2^i + \sum_{i=0}^{b} e'_i \cdot 2^i = e + e = 2 \cdot e,$$

the last two digits are 0, therefore 4 divides $2 \cdot e$ and so 2 divides e in contradiction to the assumption. The assumption must be wrong, both NAFs must be the same, so it is justified to speak of *the* NAF of a given number. □

Example 3.10:

1. The NAF can reduce the Hamming weight of a binary expansion dramatically: Consider $e = 2,097,151$. Then it is

$$\begin{array}{rcr} (e)_2 = & 11111111111111111111 & \nu_2(e) = 20 \\ & \downarrow & \\ (e)_{\mathcal{NAF}} = & 10000000000000000000\bar{1} & \nu_{\bar{2}}(e) = 2 \end{array}$$

2. On the other hand, there is not always a gain. In some cases, the NAF preserves the Hamming weight (when it is already minimal), but still increases the length. Consider $e = 699,051$.

$$\begin{array}{rcrr} (e)_2 = & 10101010101010101011 & \nu_2(e) = 11 & \lambda_2(e) = 20 \\ & \downarrow & & \\ (e)_{\mathcal{NAF}} = & 10\bar{1}0\bar{1}0\bar{1}0\bar{1}0\bar{1}0\bar{1}0\bar{1}0\bar{1}0\bar{1}0\bar{1} & \nu_{\bar{2}}(e) = 11 & \lambda_{\mathcal{NAF}}(e) = 21 \end{array}$$

3. And again, if the input is already in non-adjacent form, the NAF-transformation will neither change weight nor length. But there are also some other cases, when the NAF looks differently from the binary expansion, but it has the same length and the same weight. Consider $e = 1,258,291$.

$$\begin{array}{rcrr} (e)_2 = & 100110011001100110011 & \nu_2(e) = 11 & \lambda_2(e) = 21 \\ & \downarrow & & \\ (e)_{\mathcal{NAF}} = & 1010\bar{1}010\bar{1}010\bar{1}010\bar{1}010\bar{1} & \nu_{\bar{2}}(e) = 11 & \lambda_{\mathcal{NAF}}(e) = 21 \end{array}$$

Theorem 3.9 showed that the NAF exists for every number $e \in \mathbb{N}_0$. The facts that it also has the minimum possible Hamming weight and that the length of the NAF is at most one digit longer than the binary expansion of e will be the results of the following section.

3.3.2 Creating the NAF from a signed binary representation

Although in many applications, the NAF will be build from the usual binary expansion, every signed binary digit representation of some number e can be used to build the NAF. The following two subsections will introduce two algorithms, which provide such approaches. The first one will be used to prove the minimal Hamming weight of the NAF, the second one, an efficient on-line version, shows that the NAF can be computed in $O(\lambda_{\bar{2}}(e))$ steps and has at most one digit more than the binary expansion.

The algorithm of Jedwab and Mitchell

In [JM89], J. Jedwab and C. J. Mitchell also proposed an algorithm to create a signed binary digit representation (called MSD representation, modified signed digit representation) of a given integer from a given signed binary representation. The algorithm transforms the given signed binary representation from right to left by transforming consecutive pairs (or sequences) of nonzero digits. The resulting representation is sparse and therefore the uniquely defined NAF of the given integer.
For this reason, the algorithm produces the same result as the algorithm of Morain and Olivos presented as algorithm 3.8. However, this algorithm is slightly more general than algorithm 3.8, because it can handle signed binary digit representations as inputs. That makes it also slightly more difficult to understand. For this reason, the algorithm of Morain and Olivos has been used to introduce NAF transformation algorithms.

Algorithm 3.11 (NAF transformation by Jedwab and Mitchell)
(following J. Jedwab and C. J. Mitchell in [JM89])

Input: *A signed binary digit representation* $(e)_{\bar{2}} = (e_{\lambda_{\bar{2}}(e)-1}, e_{\lambda_{\bar{2}}(e)-2}, \ldots, e_0)$ *of a given number* $e > 0$

Output: *The NAF of e,* $(e)_{\mathcal{NAF}} = (f_{\lambda_{\mathcal{NAF}}(e)-1}, f_{\lambda_{\mathcal{NAF}}(e)-2}, \ldots, f_0)$

Set $\mathsf{e}_{\lambda_{\bar{2}}(\mathsf{e})} := 0$
Set $\mathsf{i} := 0$
while $(\mathsf{i} < \lambda_{\bar{2}}(\mathsf{e}))$ do
    increase i until e_i AND $\mathsf{e}_{\mathsf{i}+1}$ are both $\neq 0$ OR $\mathsf{i} == \lambda_{\bar{2}}(\mathsf{e})$
    if $(\mathsf{e}_\mathsf{i} \neq \mathsf{e}_{\mathsf{i}+1})$ then
        Set $\mathsf{e}_\mathsf{i} := \mathsf{e}_{\mathsf{i}+1}$
        Set $\mathsf{e}_{\mathsf{i}+1} := 0$
    else
        Set $\mathsf{j} := \mathsf{i} + 1$
        increase j until $\mathsf{e}_\mathsf{j} \neq \mathsf{e}_\mathsf{i}$
        if $(\mathsf{e}_\mathsf{j} = 0)$ then
            Set $\mathsf{e}_\mathsf{j} := \mathsf{e}_\mathsf{i}$
        else

```
14           Set e_j := 0
15        Set e_i := -e_i
16        for k from i+1 to j-1 do
17           Set e_k := 0
18  return (e)_2̄
```

Lemma 3.12 (Jedwab and Mitchell produce the NAF)
The algorithm 3.11 by Jedwab and Mitchell produces the NAF. In this, it is equivalent to the Algorithm 3.8 of Morain and Olivos for ordinary binary expansions.

Proof:

For $e > 0$, let $(e)_{\bar{2}} = (e_{\lambda_{\bar{2}}(e)-1}, e_{\lambda_{\bar{2}}(e)-2}, \ldots, e_0)$ be any signed binary representation of e which is not in non-adjacent form (otherwise, there is nothing to prove). Then algorithm 3.11 works as follows. Note that transformations of the form

$$\begin{array}{rclrcl} 1\bar{1} & \longmapsto & 01, & \bar{1}1 & \longmapsto & 0\bar{1}, \\ 01^a & \longmapsto & 10^{a-1}\bar{1}, & 0\bar{1}^a & \longmapsto & \bar{1}0^{a-1}1, \\ \bar{1}1^a & \longmapsto & 0^a\bar{1}, & 1\bar{1}^a & \longmapsto & 0^a1, \end{array}$$

do not change the values of these substrings.

start of induction (induction over the length of $(e)_{\bar{2}}$):
Let $i \in \mathbb{N}$ be the minimal number such that e_i and e_{i+1} are both nonzero digits (line 4). If $(e)_{\bar{2}}$ is not a NAF, there must be at least two adjacent nonzero digits, hence a pair $(i, i+1)$ exists. Then the first i digits are in non-adjacent form.

assumption:
Let $k \in \mathbb{N}$ be the maximal number such that $(e_k, e_{k-1}, \ldots, e_0)$ are in non-adjacent form. $(e)_{\bar{2}}$ can now be transformed into a sequence with a longer non-adjacent tail, with at least $(e_{k+1}, e_k, \ldots, e_0)$ being in non-adjacent form.

inductive proof:
$(e)_{\bar{2}}$ is either a NAF already or $e_{k+1} \neq 0$ and $e_k \neq 0$ (line 4 yields true).
The latter implies that $e_{k-1} = 0$ and it leads to one of the following cases:

case 1: $e_{k+1} \neq e_k$

This case includes $1\bar{1}$ and $\bar{1}1$. The pattern can be transformed into $e_{k+1} := 0$ and $e_k := -e_{k+1}$. Because of the assumption

that $(e_k, e_{k-1}, \dots, e_0)$ are already in non-adjacent form, the same now holds up to e_{k+1}.
Note that the Hamming weight is reduced by 1.

case 2: $e_{k+1} = e_k$
This case covers $1 \dots 1$ and $\bar{1} \dots \bar{1}$.
Let $j \geq 1$ be the minimal number, such that $e_{k+j} \neq e_k$ (line 10). Then the whole substring $(e_{k+j}, e_{k+j-1}, \dots, e_{k+1}, e_k)$ can be transformed according to the following:
The substring $(e_{k+j-1}, e_{k+j-2}, \dots, e_{k+1}, e_k)$ is transformed into

$$(e_{k+j-1}, e_{k+j-2}, \dots, e_{k+1}, e_k) \longmapsto (0, 0, \dots, 0, -e_k).$$

If $e_{k+j} \neq 0$, it must be $e_{k+j} = -e_k$ due to the maximum property of j, so $e_{k+j} + e_k = 0$. Hence, if $e_{k+j} \neq 0$, let $e_{k+j} :=$ 0, otherwise let $e_{k+j} := e_k$.
Note that if $j = 2$, then the Hamming weight is reduced if $e_{k+j} \neq 0$, otherwise it stays the same. If $j > 2$, the Hamming weight is clearly reduced because all digits between e_k and e_{k+j} become 0.

Note that throughout the transformation, the Hamming weight of the given signed binary digit representation does not increase.

□

The algorithm 3.11 provides an easy way to determine the Hamming weight of the NAF.

Theorem 3.13 (minimal Hamming weight of the NAF)
The NAF has the minimal Hamming weight $\nu_{\bar{2}}(e) = w(e)$.

Proof:

It has been shown in lemma 3.12 that during the application of algorithm 3.11, where every signed binary digit representation for a given number e can be transformed into its NAF, the Hamming weight is never increased. Because this also holds for a signed binary digit representation with minimal weight, the NAF itself must have minimal weight. □

The algorithm of Arno und Wheeler

In [AW93], Steven Arno and Ferrell S. Wheeler introduce another algorithm to create the NAF from signed binary digit representations. It is presented here, because it is an efficient on-line algorithm requiring $O(\lambda_{\bar{2}}(e))$ steps and because it is also suitable to handle signed digit representations in other radices than 2. See section 3.5 for a brief view on a possible generalization. It will also be used in theorem 3.16, where the length of the NAF is determined. The algorithm scans through the input once using only the set of data $(e_{i+1}, e_i, \epsilon_i)$, where $\epsilon_i \in \{-1, 0, 1\}$ is used to represent the carry and the function $\mathsf{sgn(x)} \in \{-1, 0, 1\}$ computes the sign of an integer.

Algorithm 3.14 (NAF transformation by Arno and Wheeler)
(following S. Arno and F. S. Wheller in [AW93])

Input: *A signed binary digit representation* $(e)_{\bar{2}} = (e_{\lambda_{\bar{2}}(e)-1}, e_{\lambda_{\bar{2}}(e)-2}, \ldots, e_0)$ *of a given number* $e > 0$, *the radix or base* $r = 2$ *of* $(e)_{\bar{r}}$.
Output: *The NAF of* e, $(e)_{\mathcal{NAF}} = (f_{\lambda_{\mathcal{NAF}}(e)-1}, f_{\lambda_{\mathcal{NAF}}(e)-2}, \ldots, f_0)$

```
Set i := 0
Set ϵ0 := 0
while (i < λ2̄(e)) do
    if ((ei + ϵi) · (ei+1 + sgn(ei + ϵi)) == 0(mod r)) then
        Set ϵi+1 := sgn(ei + ϵi)
        Set fi := ei + ϵi − sgn(ei + ϵi) · r
    else
        Set ϵi+1 := 0
        Set fi := ei + ϵi
    Set i := i + 1
if (ϵλ2̄(e) == 1) then
    Set fλ2̄(e) := 1
return (f)
```

Theorem 3.15 (Arno and Wheeler produce the NAF)
The algorithm 3.14 by Arno and Wheeler produces the NAF for $r = 2$. *In this, it is equal to the algorithms by Morain and Olivos and by Jedwab and Mitchell for ordinary binary expansion or signed binary digit representations respectively. The algorithm needs* $O(\lambda_{\bar{2}}(e))$ *steps.*

Proof:

The proof of theorem 3.15 will be done using formal verification of the algorithm 3.14. While the termination of the algorithm is trivial due to i being strictly monotonous increasing with a start value below $\lambda_{\bar{2}}$ ($e > 0$ is always assumed), the proof of the correctness will need a more thorough analysis.

The correctness of the algorithm is equivalent to the proof of the correctness of the main while-loop by using a suitable invariant IV and a loop condition C.

It is obvious that the loop condition C should be

$$C = \{0 \leq i < \lambda_{\bar{2}}(e)\}.$$

The invariant IV should demand that the created sequence (f_{i-1}, f_{i-2}, …, f_0) should always be a sparse sequence. This can be expressed by forcing that for all $0 \leq t < i-1$ the equation $f_t \cdot f_{t+1} = 0$ must hold. The invariant IV should also require the sum of all values to be e. During the transformation, the sequence

$$(e_{\lambda_{\bar{2}}(e)-1}, e_{\lambda_{\bar{2}}(e)-2}, \ldots, e_{i+1}, e_i + \epsilon_i, f_{i-1}, f_{i-2}, \ldots, f_0)$$

should always be a representation for e.

Therefore, consider

$$\begin{aligned} IV &= IV_1 \wedge IV_2 \\ &\quad \text{with} \\ IV_1 &= \{\forall\, 0 \leq t < i-1 : f_t \cdot f_{t+1} = 0\} \\ IV_2 &= \left\{ e = \sum_{k=i+1}^{\lambda_{\bar{2}}(e)-1} e_k \cdot 2^k + (e_i + \epsilon_i) \cdot 2^i + \sum_{k=0}^{i-1} f_k \cdot 2^k \right\} \end{aligned}$$

To prove that the algorithm is correct, the following scheme has to be proven:

$P_1 : \{i = 0 \wedge \epsilon_i = 0 \wedge IV \wedge C\}$
03 while $(\mathsf{i} < \lambda_{\bar{2}}(\mathsf{e}))$ do
04 if $((\mathsf{e_i} + \epsilon_\mathsf{i}) \cdot (\mathsf{e_{i+1}} + \mathrm{sgn}(\mathsf{e_i} + \epsilon_\mathsf{i})) == 0 (\mathrm{mod}\, \mathsf{r}))$ then
05 Set $\epsilon_{\mathsf{i+1}} := \mathrm{sgn}(\mathsf{e_i} + \epsilon_\mathsf{i})$
06 Set $\mathsf{f_i} := \mathsf{e_i} + \epsilon_\mathsf{i} - \mathrm{sgn}(\mathsf{e_i} + \epsilon_\mathsf{i}) \cdot \mathsf{r}$
07 else
08 Set $\epsilon_{\mathsf{i+1}} := 0$
09 Set $\mathsf{f_i} := \mathsf{e_i} + \epsilon_\mathsf{i}$
10 Set $\mathsf{i} := \mathsf{i} + 1$
$P_2 : \{IV \wedge C\} \vee \{IV \wedge \neg C\}$
$P_3 : \{IV \wedge \neg C\}$
11 if $(\epsilon_{\lambda_{\bar{2}}(\mathsf{e})} == 1)$ then
12 Set $\mathsf{f}_{\lambda_{\bar{2}}(\mathsf{e})} := 1$

part 1: prove P_1

According to lines 1 and 2 of algorithm 3.14, $i = 0$ and $\epsilon_0 = \epsilon_i = 0$. Because $i = 0$, IV_1 is trivial, because nothing has to be shown. For the same reason and because $\epsilon_0 = 0$, IV_2 just sums up all digits $e_{\lambda_{\bar{2}}(e)-1}, e_{\lambda_{\bar{2}}(e)-2}, \ldots e_0$, which add up to e by default.

part 2: prove P_2

P_2 has to be true after every run trough the while-loop. IV_1 can be shown in an inductive manner itself. If $i = 0$, no digit of $(e)_{\mathcal{NAF}}$ has been constructed, hence all digits are sparse. Similarly, if $i = 1$, only the single digit f_0 has been constructed, which is always sparse, too.

part 2.1: prove IV_1 to be true

In the case where $i > 1$, we need to differ on how the last digit f_{i-1} has been created in order to show that $f_{i-1} \cdot f_i = 0$ holds. In order to do this, the five values for $e_{i-1} + \epsilon_{i-1} \in \{-2, -1, 0, 1, 2\}$ will be reviewed. Because $r = 2$, these are the only possible values for the sum $e_{i-1} + \epsilon_{i-1}$. They can be combined into three cases, where $e_{i-1} + \epsilon_{i-1} = 1$, $e_{i-1} + \epsilon_{i-1} = -1$ or $e_{i-1} + \epsilon_{i-1} \equiv 0 \pmod 2$.

case 1: $e_{i-1} + \epsilon_{i-1} \equiv 0 \pmod 2$

If $e_{i-1} + \epsilon_{i-1} \equiv 0 \pmod 2$, then the condition of the if-statement in line 4 of algorithm 3.14 will always be true, because the product will be 0 or a multiple of $r = 2$. Hence

$$\begin{aligned} f_{i-1} &:= e_{i-1} + \epsilon_{i-1} - \operatorname{sgn}(e_{i-1} + \epsilon_{i-1}) \cdot r \\ &= \left\{ \begin{array}{ll} -2 - (-1) \cdot 2 & if\ e_{i-1} + \epsilon_{i-1} = -2 \\ 0 - 0 \cdot 2 & if\ e_{i-1} + \epsilon_{i-1} = 0 \\ 2 - (1) \cdot 2 & if\ e_{i-1} + \epsilon_{i-1} = +2 \end{array} \right\} = 0 \end{aligned}$$

If $f_{i-1} = 0$, then $f_i \cdot f_{i-1} = 0$ regardless of the value of f_i. Therefore, IV_1 holds.

case 2: $e_{i-1} + \epsilon_{i-1} = -1$

If $e_{i-1} + \epsilon_{i-1} = -1$, then the if-condition of line 4 can be either true or false depending on the value of e_i:

$e_i = -1$ With this value, the if-condition evaluates to $(-1) \cdot (-1 + -1) \equiv 0 \pmod 2$. Hence $f_{i-1} := e_{i-1} + \epsilon_{i-1} - \operatorname{sgn}(e_{i-1} + \epsilon_{i-1}) \cdot 2 = -1 - (-1) \cdot 2 = 1$ and $\epsilon_i := \operatorname{sgn}(e_{i-1} + \epsilon_{i-1}) = -1$. Now, the next step (the actual step to be examined) has the configuration $e_i + \epsilon_i = -2$, which leads to $f_i := 0$ according to case 1. Therefore, $f_i \cdot f_{i-1} = 0$ and IV_1 is true.

$e_i = 0$ With e_i being 0, the if-condition evaluates to -1 and the else-branch is being executed. There, $f_{i-1} = e_{i-1} + \epsilon_{i-1} = -1$ and $\epsilon_i := 0$ are computed. With these values, $e_i + \epsilon_i = 0$. This results in $f_i := 0$ according to case 1 and again, IV_1 is true.

$e_i = 1$ If $e_i = 1$, the if-condition's left side gives 2 and therefore, f_{i-1} and e_i are computed in the then-branch of the conditional, $f_{i-1} := -1 - (-1) \cdot 2 = 1$ and $\epsilon_i := -1$. This leads to the next step's sum being $e_i + \epsilon_i = 1 + -1 = 0$ which makes the algorithm set $f_i := 0$ according to case 1. But then, $f_i \cdot f_{i-1} = 0$ and IV_1 is proven.

case 3: $e_{i-1} + \epsilon_{i-1} = 1$
This case behaves very similar to case 2, but with respect to clarity, cases 2 and 3 have not been combined. Again, the value of e_i leads to different cases:

$e_i = -1$ In this case, the if-condition evaluates to 0 and the then-branch leaves $f_{i-1} := -1$ and $\epsilon_i := 1$. This sets $e_i + \epsilon_i$ to $-1 + 1 = 0$ and therefore, according to case 1, $f_i := 0$ and with that the first part of the invariant, IV_1, is true.

$e_i = 0$ This case makes the if-condition result in 1 and the else-branch to compute $f_{i-1} := 1$ and $\epsilon_i := 0$. Again, with $e_i = 0$ and $\epsilon_i = 0$, the sum is zero which proves IV_1 due to the result of case 1.

$e_i = 1$ In analogy to $e_i = -1$, this case's if-condition is true and $f_{i-1} := -1$, $\epsilon_i := 1$. This makes $e_i + \epsilon_i = 2 \equiv 0 \pmod 2$, hence case 1 allows to conclude that $f_i := 0$ and therefore, IV_1 to be true.

part 2.2: prove IV_2 to be true
To show that IV_2 is true after each run though the while-loop, one has to show that while before the loop started, IV_2 was true for i, it is now true for $i+1$. This can be expressed by the statement

$$\left\{ \sum_{k=i+2}^{\lambda_2(e)-1} e_k \cdot 2^k + e_{i+1} \cdot 2^{i+1} + (e_i + \epsilon_i) \cdot 2^i + \sum_{k=0}^{i-1} f_k \cdot 2^k \right.$$

$$= e =$$

$$\left. \sum_{k=i+2}^{\lambda_2(e)-1} e_k \cdot 2^k + (e_{i+1} + \epsilon_{i+1}) \cdot 2^{i+1} + \sum_{k=0}^{i} f_k \cdot 2^k \right\}.$$

It can be simplified, because only for the indices i and $i+1$ any changes appear, so it has to be shown that

$$(e_{i+1} + \epsilon_{i+1}) \cdot 2^{i+1} + f_i \cdot 2^i = e_{i+1} \cdot 2^{i+1} + (e_i + \epsilon_i) \cdot 2^i.$$

This can easily be done by inserting the computed values for f_i and ϵ_{i+1} depending on whether the then-branch or the else-branch of the conditional has been chosen.

then: In this case, inserting the definition for f_i and ϵ_{i+1} from lines 5 and 6 of algorithm 3.14 leads to

$$\begin{aligned}
& (e_{i+1} + \epsilon_{i+1}) \cdot 2^{i+1} + f_i \cdot 2^i - e_{i+1} \cdot 2^{i+1} - (e_i + \epsilon_i) \cdot 2^i \\
= \; & (e_{i+1} + \operatorname{sgn}(e_i + \epsilon_i)) \cdot 2^{i+1} + (e_i + \epsilon_i - \operatorname{sgn}(e_i + \epsilon_i) \cdot 2) \cdot 2^i \\
& - e_{i+1} \cdot 2^{i+1} - (e_i + \epsilon_i) \cdot 2^i \\
= \; & 2^{i+1} \cdot (e_{i+1} + \operatorname{sgn}(e_i + \epsilon_i) - \operatorname{sgn}(e_i + \epsilon_i) - e_{i+1}) \\
& + 2^i \cdot (e_i + \epsilon_i - (e_i + \epsilon_i))) \\
= \; & 0.
\end{aligned}$$

else: For this analysis, the definitions of lines 8 and 9 have to be inserted:

$$\begin{aligned}
& (e_{i+1} + \epsilon_{i+1}) \cdot 2^{i+1} + f_i \cdot 2^i - e_{i+1} \cdot 2^{i+1} - (e_i + \epsilon_i) \cdot 2^i \\
= \; & (e_{i+1} + 0) \cdot 2^{i+1} + (e_i + \epsilon_i) \cdot 2^i - e_{i+1} \cdot 2^{i+1} - (e_i + \epsilon_i) \cdot 2^i \\
= \; & 0.
\end{aligned}$$

Therefore, IV_1 and IV_2 are true at position P_2.

part 3: prove P_3

At the point of P_3, the invariant IV is true, because it is true after each step through the loop. If the while-loop ended, $i = \lambda_{\bar{2}}(e)$. It is obvious, that if there's still a carry left, it has to be added to the next digit, which is a zero. It is left to show that the resulting sequence $(f_{\lambda_{\bar{2}}(e)}, f_{\lambda_{\bar{2}}(e)-1}, \ldots, f_0)$ is sparse even if the carry is 1. If it is 0, no further action has to be taken, therefore, IV stays true and a sparse sequence representing e has been constructed.

But if $\epsilon_{\lambda_{\bar{2}}(e)} = 1$, then in the last application of the if-conditional, the true-branch must have been chosen, because in the else-branch, ϵ_i is always set to 0.

Knowing this, the if-condition reveals that $(e_{\lambda_{\bar{2}}(e)-1} + \epsilon_{\lambda_{\bar{2}}(e)-1}) \cdot (e_{\lambda_{\bar{2}}(e)} + \operatorname{sgn}(e_{\lambda_{\bar{2}}(e)-1} + \epsilon_{\lambda_{\bar{2}}(e)-1})) \equiv 0 \pmod 2$. Because $e_{\lambda_{\bar{2}}(e)} = 0$, this means that $(e_{\lambda_{\bar{2}}(e)-1} + \epsilon_{\lambda_{\bar{2}}(e)-1}) \cdot \operatorname{sgn}(e_{\lambda_{\bar{2}}(e)-1} + \epsilon_{\lambda_{\bar{2}}(e)-1}) \equiv 0 \pmod 2$, which means that $e_{\lambda_{\bar{2}}(e)-1} + \epsilon_{\lambda_{\bar{2}}(e)-1} \equiv 0 \pmod 2$. According to case 1 from part 2.1 of this proof, $f_{\lambda_{\bar{2}}(e)-1}$ must have been set to 0, hence, $f_{\lambda_{\bar{2}}(e)} \cdot f_{\lambda_{\bar{2}}(e)-1} = 0$.

Therefore, the constructed sequence is a sparse signed binary representation for e, which means that it is the NAF, because it is uniquely defined according to theorem 3.9.

The claimed costs can be easily deducted from the fact that the loop is exactly executed $\lambda_{\bar{2}}(e) - 1$ times. □

After these different algorithms, that all produce the NAF starting at slightly different points, it still remains to prove that the length of the NAF lies in very narrow bounds, in fact, it cannot increase the binary expansion by more than one digit.

Corollary 3.16 (The length of the NAF)
Let $(e)_2 = (e_{\lambda_2(e)-1}, e_{\lambda_2(e)-2}, \ldots, e_0)$ with $e_{\lambda_2(e)-1} = 1$ be the binary expansion of a given integer e, and let $(e)_{\mathcal{NAF}} = (f_{\lambda_{\mathcal{NAF}}(e)-1}, f_{\lambda_{\mathcal{NAF}}(e)-2}, \ldots, f_0)$ be the NAF of e. Then the length $\lambda_{\mathcal{NAF}}(e)$ can take one of two values:

$$\lambda_{\mathcal{NAF}}(e) \quad \in \quad \{\lambda_2(e),\ \lambda_2(e)+1\} \tag{3.7}$$

Proof:

The lower bound $\lambda_{\mathcal{NAF}}(e) \geq \lambda_2(e)$ arises by the fact, that the NAF transformation of the binary expansion (algorithm 3.8) can only resolve runs of adjacent ones by adding a carry bit to the remaining left part of the binary expansion, while possibly changing the current digit. Hence it must be shown that the leading digit cannot be eliminated without gain in length. Algorithm 3.8 shows that a 1 is only changed if there are adjacent ones next to each other. If the algorithm examines the bit $e_{\lambda_2(e)-2}$, two situations may arise, 11 and 10. In the first case, the resulting string starts with $10\bar{1}$ or 100 (depending on the history), in the second case, the string is left unchanged at 10. Therefore, the length cannot decrease.

The upper bound follows equally, or with a reference to algorithm 3.14, where it can easily be seen that the highest digit the algorithm may modify is $f_{\lambda_2(e)}$. □

As a result of this section, it is clear that the non-adjacent form can be computed from either the binary expansion or any signed binary digit representation previously gathered in an efficient way. Especially the on-line algorithm 3.14 by Arno and Wheeler shows that the NAF can be computed in $\mathrm{O}(\lambda_2(\mathrm{e}))$ steps. This is important in practical applications, because although the model used in this thesis only counts operations in the underlying field of the exponentiation problem, practical applications will face the need to compute the NAF. However, field operations are usually very expensive compared to the binary changes that NAF transformation algorithms apply (in a general ring, the basic operations, like multiplication, require $O(\lambda_2(e)^2)$ steps using classical arithmetic (see [GG99], p. 757 and §11)), hence the creation of the NAF is neglectable.
In this sense, the NAF provides a good basis for introducing the subtraction, because of the uniqueness, the minimality of its Hamming weight and the minimal increase in length. The next chapter will analyze, how this representation may be used to enhance exponentiation algorithms. In order

to do so, knowledge about some more properties of the NAF is needed and will be provided in the following section.

3.4 Properties of the NAF

In order to be able to analyze the use of the non-adjacent form for exponentiation algorithms and the creation of addition-subtraction chains, some special properties of this special signed digit representation have to be known. This section will state results about the total number of sparse signed digit strings for a certain length n, about the expected length of the NAF compared to the length of the binary expansion and about the occurrence and distribution of the digits. Some parts of the results have not been published yet.
First, the examined properties are defined.

Definition 3.17 (number of and average length of sparse strings)

(1) *For any positive integer $n \in \mathbb{N}$, let $N_q(n)$ be the set of all non-adjacent signed q-ary digit strings with* **at most n digits** *(leading zeros being acceptable). Formally, it is*

$$N_q(n) \;:=\; \{w = (w_{n-1}, \ldots, w_0) \in \{-q+1, \ldots, q-1\}^n \mid w_i \cdot w_{i+1} = 0 \quad \forall\, 0 \le i < n-1\}.$$

Let $s_q(n) \in \mathbb{N}$ denote the total number of sparse signed q-ary digit strings with **at most n digits** *(leading zeros being acceptable). This means that $s_q(n)$ is the cardinality of $N_q(n)$,*

$$s_q(n) := \#N_q(n).$$

Let $Val(N_q(n))$ denote the set of the integer values of all signed q-ary words contained in $N_q(n)$,

$$Val(N_q(n)) := \{e \in \mathbb{N} \mid (e)_{\mathcal{NAF}} \in N_q(n)\}.$$

For $w \in N_q(n)$, let $Val(w) := e$ iff $(e)_{\mathcal{NAF}} = w$.

(2) *For any positive integer $n \in \mathbb{N}$, let $N_q^{(-1)}(n)$ be the set of all non-adjacent signed q-ary digit strings with* **exactly n digits** *(hence leading 1 mandatory), that encode a number $e \in \mathbb{N}$, whose binary expansion needs one digit less,*

$$N_q^{(-1)}(n) \;:=\; \{w = (1, w_{n-2}, \ldots, w_0) \in \{-q+1, \ldots, q-1\}^n \mid \exists\, e \in \mathbb{N} :\; (e)_{\mathcal{NAF}} = w \;\wedge\; \lambda_2(e) = n-1\}.$$

Again, let $s_q^{(-1)}(n)$ *denote the number of such strings, e.g. let* $s_q^{(-1)}(n)$ *denote the cardinality of* $N_q^{(-1)}(n)$,

$$s_q^{(-1)}(n) := \# N_q^{(-1)}(n).$$

Clearly, it is $N_q^{(-1)}(n) \subset N_q(n)$, $s_q^{(-1)}(n) < s_q(n)$ *(consider* $e = 2^{n-1}$*).*

(3) *For any positive integer* $n \in \mathbb{N}$, *let* $N_q^+(n)$ *be the set of all non-adjacent* **unsigned** *q-ary digit strings with* **at most n digits** *(leading zeros being acceptable). Elements* $w \in N_q^+(n)$ *only consist of the digits* 0, ..., q-1. *It is*

$$N_q^+(n) \quad := \quad \{w = (w_{n-1}, \ldots, w_0) \in \{0, \ldots, q-1\}^n \mid w_i \cdot w_{i+1} = 0 \\ \forall\, 0 \leq i < n-1\}.$$

Again, let $s_q^+(n)$ *denote the number of such words* w,

$$s_q^+(n) := \# N_q^+(n).$$

3.4.1 Determining the number of NAF strings with length n

Especially for those exponentiation algorithms that precompute possible window values, it is important to know how many different window values may appear. This is the problem to determine the total number of different sparse signed digit strings for a certain number of digits (leading zeros being acceptable). Some of these numbers are depicted in table 3.6. It shows that there are more non-adjacent strings with a fixed number of digits than binary expansions. The following theorem counts these numbers exactly. The special case $q = 2$ has been shown independently from the recursive formula in part 1 in [EgK90] as theorem 2. The recursive formula (3.8), which gives a good understanding about how algorithms may build these sparse strings (see algorithm 4.11), and the explicit formula (3.9) for general q seem not to have been published yet.

n	$s_2(n)$	n	$s_2(n)$	n	$s_2(n)$
1	3	6	85	11	2731
2	5	7	171	12	5461
3	11	8	341	13	10923
4	21	9	683	14	21845
5	43	10	1365	15	43691

Table 3.6: Some numbers of sparse signed binary n-digit strings

Theorem 3.18 (number of sparse signed q-ary digit strings)

1. *The total number of sparse signed q-ary digit representations of integers using exactly n bits (leading zeros being acceptable) can be described by the following linear recurrence:*

$$\begin{aligned} s_q(0) &= 1 \\ s_q(1) &= 2q-1 \\ s_q(n) &= s_q(n-1) + (2q-2)\cdot s_q(n-2) \qquad (3.8) \\ &\qquad\qquad\qquad \forall\, n \geq 2 \end{aligned}$$

For all $n \in \mathbb{N}$, $s_q(n)$ is odd.

2. *The generic linear recurrence (3.8) from part 1 evaluates to the following explicit formula:*

$$s_q(n) = \frac{1}{2}\cdot(\lambda_+^n + \lambda_-^n) + \frac{1}{2}\cdot(\lambda_+^n - \lambda_-^n)\cdot\frac{4q-3}{\sqrt{8q-7}} \qquad (3.9)$$

$$\text{with} \qquad \lambda_+ := \frac{1}{2} + \frac{1}{2}\cdot\sqrt{8q-7}$$

$$\text{and} \qquad \lambda_- := \frac{1}{2} - \frac{1}{2}\cdot\sqrt{8q-7}$$

In the case where $q = 2$, this formula is equal to

$$s_2(n) = \frac{1}{3}\cdot(2^{n+2} + (-1)^{n+1}). \qquad (3.10)$$

Formula (3.10) has also been shown by Ö. Eğecioğlu and Ç. K. Koç in [EgK90].

3. *The set $Val(N_q(n))$ of the integer values corresponding to all elements $w \in N_q(n)$ is for all $n \in \mathbb{N}$ and $q \geq 2$ a subset of the following interval,*

$$\begin{aligned} Val(N_q(n)) \subseteq \Bigg\{ & -\frac{1}{q+1}\cdot\left(q^{n+1} - \frac{q+1}{2} + (-1)^{n+1}\cdot\frac{q-1}{2}\right), \ldots, \\ & \frac{1}{q+1}\cdot\left(q^{n+1} - \frac{q+1}{2} + (-1)^{n+1}\cdot\frac{q-1}{2}\right)\Bigg\}. \qquad (3.11) \end{aligned}$$

For $q = 2$, the set $Val(N_q(n))$ is equal to the interval, hence $N_q(n)$ contains the non-adjacent forms of all integers between the bounds of the interval, giving the simplified equation

$$\begin{aligned} Val(N_2(n)) &= \left\{ -\frac{1}{3}\cdot\left(2^{n+1} + \frac{(-1)^{n+1}-3}{2}\right), \ldots, \right. \qquad (3.12) \\ &\qquad \left. \frac{1}{3}\cdot\left(2^{n+1} + \frac{(-1)^{n+1}-3}{2}\right)\right\} \\ &= \left\{ \left\lceil -\frac{2^{n+1}}{3} \right\rceil, \ldots, \left\lfloor \frac{2^{n+1}}{3} \right\rfloor \right\}. \end{aligned}$$

For $q > 2$, $Val(N_q(n))$ is a real subset of the interval, the smallest positive integer without a NAF in $N_q(n)$ is $q + 1$.

Proof:

proof of part 1

For the first two start values, the value of $s_q(0)$ is a definition. It has to be one, because there are no non-adjacent words with zero digits and the value is used as a factor.

For $s_q(1)$, the number of different sparse signed q-ary digit strings with one digit can be counted easily. In order to do that, note that for any signed q-ary representation, there are $2q-1$ possible digits, consisting of the 0, the positive digits 1, 2, ..., (q-1) and the corresponding negative digits $\bar{1}$, $\bar{2}$, ..., $(\overline{\mathsf{q}-1})$. This also implies that there are $2 \cdot (q-1) = 2q-2$ nonzero digits.

A sparse signed digit string of length 1 can consist of any of the $2q-1$ possible digits, hence $s_q(1) = 2q-1$.

For any other value of $s_q(n)$ with $n \geq 2$, consider the structure of the possible strings. There are two possible cases: Such a string $w \in N_q(n)$ may either start with a zero or with a nonzero digit. If it starts with a zero, the remaining $n-1$ digits may form any sparse signed q-ary digit string of length $n-1$, which gives $s_q(n-1)$ possibilities (see figure 3.1).

0	$N_q(n-1)$

Figure 3.1: Configurations of $w \in N_q(n)$ starting with 0

If on the other hand w starts with any nonzero digit, the next digit must be a zero in order to guarantee the resulting string to be sparse. The following $n-2$ digits may again form any sparse signed q-ary digit string of length $n-2$. With the possible $2q-2$ nonzero digits at the position of the first digit, there are $(2q-2) \cdot s_q(n-2)$ possible configurations (see figure 3.2).

$\star$	0	$N_q(n-2)$

Figure 3.2: Configurations of $w \in N_q(n)$ not starting with 0

Because both cases are disjoint, these results prove the claimed recursion $s_q(n) = s_q(n-1) + (2q-2) \cdot s_q(n-2)$.

$s_q(n)$ is odd for all $n \in \mathbb{N}$, $q \geq 2$, because for every positive element $w \in N_q(n)$, the element encoding the negative value of w is also in $N_q(n)$ (negate all digits). This gives an even number of elements and together with 0, which has no corresponding element, $s_q(n)$ is always odd.

proof of part 2

The explicit formula is best proven using induction over the number of digits n. Within the following proof, assume

$$\mathcal{Z} \;:=\; \lambda_+ - \lambda_- = \sqrt{8q-7} \tag{3.13}$$

then it is obviously $\mathcal{Z}^2 = 8q - 7$. This will be used for abbreviations within the equations.

start of induction over n

According to the claimed formula (3.9), it is

$$\begin{aligned} s_q(0) &= \frac{1}{2} \cdot (\lambda_+^0 + \lambda_-^0) + \frac{1}{2} \cdot (\lambda_+^0 - \lambda_-^0) \cdot \frac{4q-3}{\mathcal{Z}} \\ &= \frac{1}{2} \cdot 2 = 1, \end{aligned}$$

which gives the correct result for $s_q(0)$ according to part 1 of this theorem.

Similarly, it is

$$\begin{aligned} s_q(1) &= \frac{1}{2} \cdot (\lambda_+ + \lambda_-) + \frac{1}{2} \cdot (\lambda_+ - \lambda_-) \cdot \frac{4q-3}{\mathcal{Z}} \\ &= \frac{1}{2} \cdot \left(\frac{1}{2} + \frac{\mathcal{Z}}{2} + \frac{1}{2} - \frac{\mathcal{Z}}{2} \right) + \frac{1}{2} \cdot (4q-3) \\ &= \frac{1}{2} \cdot (1 + 4q - 3) \\ &= 2q - 1 \end{aligned}$$

which is again proven right by the first part of this theorem.

assumption

Assume formula (3.9) to be true for all $0 \leq k \leq n$. Then it can also shown to be true for $k = n + 1$.

inductive proof for $k = n + 1$

For $s_q(n+1)$, the linear recurrence (3.8) will be used:

$$\begin{aligned}
s_q(n+1) &\overset{(3.8)}{=} s_q(n) + (2q-2) \cdot s_q(n-1) \\
&= \frac{1}{2} \cdot \Big((\lambda_+^n + \lambda_-^n) + (\lambda_+^n - \lambda_-^n) \cdot \frac{4q-3}{\mathcal{Z}} \\
&\quad +(2q-2) \cdot (\lambda_+^{n-1} + \lambda_-^{n-1}) \\
&\quad +(2q-2) \cdot (\lambda_+^{n-1} - \lambda_-^{n-1}) \cdot \frac{4q-3}{\mathcal{Z}} \Big) \\
&= \frac{1}{2} \cdot \big((\lambda_+^n + \lambda_-^n) + (2q-2) \cdot (\lambda_+^{n-1} + \lambda_-^{n-1}) \big) \\
&\quad + \frac{4q-3}{2\mathcal{Z}} \cdot \big((\lambda_+^n - \lambda_-^n) \\
&\qquad\qquad +(2q-2) \cdot (\lambda_+^{n-1} - \lambda_-^{n-1}) \big)
\end{aligned} \tag{3.14}$$

The right side of this equation can now be transformed into the claimed formula by using the equations

$$(\lambda_+^k + \lambda_-^k) + (2q-2) \cdot (\lambda_+^{k-1} + \lambda_-^{k-1}) = \lambda_+^{k+1} + \lambda_-^{k+1}, \tag{3.15}$$

$$(\lambda_+^k - \lambda_-^k) + (2q-2) \cdot (\lambda_+^{k-1} - \lambda_-^{k-1}) = \lambda_+^{k+1} - \lambda_-^{k+1}. \tag{3.16}$$

The following steps will prove this formula starting from the form found in (3.14). Some transformation steps may become clearer if they're read from bottom to top:

$$\begin{aligned}
&(\lambda_+^k + \lambda_-^k) + (2q-2) \cdot (\lambda_+^{k-1} + \lambda_-^{k-1}) \\
&= (\lambda_+^k + \lambda_-^k) + \frac{1}{4} \cdot (8q-8) \cdot (\lambda_+^{k-1} + \lambda_-^{k-1}) \\
&= (\lambda_+^k + \lambda_-^k) + \frac{1}{2} \cdot \left(\frac{8q-7}{2} - \frac{1}{2} \right) \cdot (\lambda_+^{k-1} + \lambda_-^{k-1}) \\
&= (\lambda_+^k + \lambda_-^k) + \frac{1}{2} \cdot \left(\frac{\mathcal{Z}^2}{2} + \frac{\mathcal{Z}}{2} - \frac{\mathcal{Z}}{2} - \frac{1}{2} \right) \cdot (\lambda_+^{k-1} + \lambda_-^{k-1}) \\
&= (\lambda_+^k + \lambda_-^k) + \frac{1}{2} \cdot \left[\left(\frac{\mathcal{Z}}{2} + \frac{\mathcal{Z}^2}{2} \right) \cdot \lambda_+^{k-1} - \left(\frac{1}{2} + \frac{\mathcal{Z}}{2} \right) \cdot \lambda_+^{k-1} \right. \\
&\qquad \left. - \left(\frac{\mathcal{Z}}{2} - \frac{\mathcal{Z}^2}{2} \right) \cdot \lambda_-^{k-1} - \left(\frac{1}{2} - \frac{\mathcal{Z}}{2} \right) \cdot \lambda_-^{k-1} \right] \\
&= (\lambda_+^k + \lambda_-^k) + \frac{1}{2} \cdot \left(\mathcal{Z} \cdot \lambda_+^k - \lambda_+^k - \mathcal{Z} \cdot \lambda_-^k - \lambda_-^k \right) \\
&= (\lambda_+^k + \lambda_-^k) + \frac{\mathcal{Z}}{2} \cdot (\lambda_+^k - \lambda_-^k) - \frac{1}{2} \cdot (\lambda_+^k + \lambda_-^k) \\
&= \frac{1}{2} \cdot (\lambda_+^k + \lambda_-^k) + \frac{\mathcal{Z}}{2} \cdot (\lambda_+^k - \lambda_-^k) \\
&= \left(\frac{1}{2} + \frac{\mathcal{Z}}{2} \right) \cdot \lambda_+^k + \left(\frac{1}{2} - \frac{\mathcal{Z}}{2} \right) \cdot \lambda_-^k = (\lambda_+^{k+1} + \lambda_-^{k+1})
\end{aligned}$$

Equation (3.16) follows completely analogously. Now (3.14) can be easily transformed into

$$s_q(n+1) \quad = \quad \frac{1}{2} \cdot \left(\lambda_+^{n+1} + \lambda_-^{n+1}\right) + \frac{1}{2} \cdot \left(\lambda_+^{n+1} - \lambda_-^{n+1}\right) \cdot \frac{4q-3}{\sqrt{8q-7}},$$

proving the claimed explicit formula.

The formula of Eğecioğlu and Koç follows by setting $q = 2$. Note that

$$\begin{aligned} \mathcal{Z} &= \sqrt{8 \cdot 2 - 7} = 3 \\ \lambda_+ &= \frac{1}{2} + \frac{\mathcal{Z}}{2} = 2 \\ \lambda_- &= \frac{1}{2} - \frac{\mathcal{Z}}{2} = -1 \end{aligned}$$

and hence

$$\begin{aligned} s_2(n) &= \frac{1}{2} \cdot (2^n + (-1)^n) + \frac{1}{2} \cdot (2^n - (-1)^n) \cdot \frac{5}{3} \\ &= \frac{3}{6} \cdot 2^n + \frac{5}{6} \cdot 2^n + \frac{3}{6} \cdot (-1)^n - \frac{5}{6} \cdot (-1)^n \\ &= \frac{4}{3} \cdot 2^n - \frac{1}{3} \cdot (-1)^n \\ &= \frac{1}{3} \cdot \left(2^{n+2} + (-1)^{n+1}\right). \end{aligned}$$

proof of part 3

Let m:=q-1. Then the highest number to be represented with a sparse q-ary n-digit string is obviously m0m0m0…m0 if n is even and if n is odd m0m0m0…m. The lowest possible number can be obtained by switching all occurrences of m into $\overline{\mathsf{m}}$. Therefore, all values of $Val(N_q(n))$ must lie between these bounds. The equality for $q = 2$ will be proven by counting the possible values between the minimal and the maximal possible value to appear. With the results of part 1, the equality will follow. For $q > 2$, a counterexample will be given. To get the bounds first, the two cases of n being even or odd will be analyzed separately.

case 1: Let n be odd.
Then the highest number A that can be displayed with n sparse q-ary digits is

$$\begin{aligned}
A &= Val(\overbrace{\mathsf{m0m0m0}\ldots\mathsf{m}}^{n\ digits}) = \sum_{\substack{k\ odd\\ n\geq k>0}} (q-1)\cdot q^{k-1} \\
&= \sum_{n\geq k>0} (q-1)\cdot q^{k-1} - \sum_{\substack{k\ even\\ n>k>0}} (q-1)\cdot q^{k-1} \\
&= (q-1)\cdot\frac{q^n-1}{q-1} - \frac{1}{q}\cdot(A-(q-1)) \\
\Rightarrow A+\frac{1}{q}\cdot A &= q^n - 1 + \frac{q-1}{q} \\
\Rightarrow \qquad A &= \frac{1}{q+1}\cdot\left(q^{n+1}-q+q-1\right) \\
&= \frac{1}{q+1}\cdot\left(q^{n+1}-1\right).
\end{aligned}$$

The lowest number B that can be displayed with n sparse q-ary digits is now due to symmetry

$$B = -\frac{1}{q+1}\cdot\left(q^{n+1}-1\right).$$

case 2: Let n be even.
Then the highest number A that can be displayed with n sparse q-ary digits is

$$\begin{aligned}
A &= Val(\overbrace{\mathsf{m0m0}\ldots\mathsf{m0}}^{n\ digits}) = \sum_{\substack{k\ even\\ n\geq k>0}} (q-1)\cdot q^{k-1} \\
&= \sum_{n\geq k>0} (q-1)\cdot q^{k-1} - \sum_{\substack{k\ odd\\ n>k>0}} (q-1)\cdot q^{k-1} \\
&= (q-1)\cdot\frac{q^n-1}{q-1} - \left(\frac{1}{q}\cdot A\right) \\
\Rightarrow A+\frac{1}{q}\cdot A &= q^n - 1 \\
\Rightarrow \qquad A &= \frac{1}{q+1}\cdot(q^n-q).
\end{aligned}$$

Again, the lowest number B that can be displayed with n sparse digits is now due to symmetry

$$B = -\frac{1}{q+1}\cdot(q^{n+1}-q).$$

To combine both cases, the following alternating term is used:

$$-\frac{q+1}{2} + (-1)^{n+1} \cdot \frac{q-1}{2} \quad = \quad \begin{cases} -1 & if\ n\ is\ odd \\ -q & if\ n\ is\ even, \end{cases}$$

and because all other values of the elements of $N_q(n)$ must be located between the bounds, the claimed formula (3.11) follows.

For $q = 2$, the bounds are equal to

$$\pm\ \frac{1}{3} \cdot \left(2^{n+1} - \frac{3}{2} + (-1)^{n+1} \cdot \frac{1}{2}\right),$$

giving formula (3.12). As the alternating term always makes sure that the bound is an integer and the alternating term's value is < 1, it will always make sure that the result is corrected towards the next lower integer (in case of the upper border) or towards the next higher integer (in case of the lower border). Hence, the Gaussian brackets notation follows immediately.

The equality of the set of values and the stated interval can be shown by comparing the cardinalities of both sets. The set $Val(N_q(n))$ contains as many elements as $N_q(n)$, which has been defined as $s_q(n)$. The interval contains the following number of elements (including zero):

$$\begin{aligned} & 2 \cdot \frac{1}{3} \cdot \left(2^{n+1} + \frac{(-1)^{n+1} - 3}{2}\right) + 1 \\ = \quad & \frac{1}{3} \cdot \left(2^{n+2} + (-1)^{n+1} - 3 + 3\right) \\ = \quad & \frac{1}{3} \cdot \left(2^{n+2} + (-1)^{n+1}\right) = s_q(n) \end{aligned}$$

Therefore, both sets must be equal.

For $q > 2$, consider the following integer $e = q+1$: It is $(e)_q = (\mathsf{11})$ and assume that there was a sparse signed q-ary digit string representing e. In order to do that, it must consist of at least three digits. If the last digit was $\mathsf{0}$, then $q+1$ would be a multiple of q, which it isn't. Therefore, the last digit (the rightmost) must be nonzero. Then the second last digit is bound to be zero.

As e is positive, the value represented by the leftmost $n-2$ digits must be positive. The lowest possible positive integer that can possibly be represented by these digits is $(\mathsf{100}) = (q^2)_q$. If e can be represented sparsely, then the last digit must be low enough to reduce that value to $q+1$. But as the lowest possible value of a signed q-ary digit is $\overline{\mathsf{q-1}} = 1 - q$, this means that it must be

$$\begin{aligned} & q^2 - q + 1 \ \leq \ q + 1 \mathrel{\hat{=}} (\mathsf{11}) \\ \Leftrightarrow \quad & q^2 \ \leq \ 2q. \end{aligned}$$

But for $q \geq 3$, it is $q^2 > 2q$ and this proves the assumption to be wrong. Therefore, there are numbers in the interval that do not have a non-adjacent q-ary digit string in $N_q(n)$, hence, the set $Val(N_q(n))$ must be a real subset of the interval. □

Knowledge about the value of $s_2(n)$ and about the words contained in $N_2(n)$ is vital for the precomputation step of any addition-subtraction chain method using the NAF as the input representation, because all these values have to be precomputed. An efficient way to compute all x^e for $(e)_{\mathcal{NAF}} \in N_2(n)$ will be given in lemma 4.10, where the precomputation step of the NAF-based Brauer method will be examined. The number $s_2(n)$ will be needed for comparisons with the ordinary Brauer method in chapter 4, too.

3.4.2 Determining the average length of the NAF

It has been shown in chapter 2 that the length of the chosen q-ary representation determines the number of q-steps necessary for exponentiation. Therefore, analyses require exact knowledge about the average length of the representations the algorithms are based on. It has been shown in corollary 3.16 that the NAF is either equal to or at most one digit longer than the binary expansion. For further analyses, it is of interest how often the NAF increases the length, e.g. what is the average length $\lambda_{\mathcal{NAF}}(e)$ of the NAF compared to the binary expansion length $\lambda_2(e)$?
The possible increase is only very slim, hence, it will only be able to account for a small constant amount of additional operations. It is of interest though, because in some cases, with Brauer's method, the additional digit may lead to an additional window, which requires some additional operations.
Additionally, the question has a nice solution as the following theorem shows, which seems not to have been published anywhere yet. Table 3.7 shows the first 16 values of the constructed sequence $s_q^{(-1)}(n)$ for $q = 2$.

$\boldsymbol{n}$	1	2	3	4	5	6	7	8
$\boldsymbol{s_2^{(-1)}(n)}$	0	0	1	2	5	10	21	42
$\boldsymbol{n}$	9	10	11	12	13	14	15	16
$\boldsymbol{s_2^{(-1)}(n)}$	85	170	341	682	1365	2730	5461	10922

Table 3.7: The first 16 values of the sequence $s_2^{(-1)}(n)$

It has been shown in part 3 of theorem 3.18, that for a general choice of q, the set $N_q(n)$ does not contain the non-adjacent forms of all integers in a certain range, which means that there are values for e that cannot be expressed by a sparse signed q-ary digit string with all nonzero digits being separated. It is the main reason, why it is not feasible to generalize the NAF in this way. See section 3.5 for more details about possible generalizations. Due to this fact, thoughts about $s_q^{(-1)}(n)$ are only interesting for $q = 2$. Therefore, the following theorem only gives answers for this case needed within further analyses.

Theorem 3.19 (Expected length of the NAF)
Let $(e)_2 = (e_{\lambda_2(e)-1}, e_{\lambda_2(e)-2}, \ldots, e_0)$ with $e_{\lambda_2(e)-1} = 1$ be the binary expansion of a random integer e, and let $n := \lambda_2(e) + 1$. Let $s_2^{(-1)}(n)$ denote the number of sparse binary digit strings that encode a number whose binary expansion is one digit shorter (as defined in definition 3.17).
Then the following equations hold for the the numbers $s_2^{(-1)}(n)$:

$$s_2^{(-1)}(n) \quad = \quad \frac{1}{6} \cdot \left(2^n - 3 + (-1)^{n+1}\right), \tag{3.17}$$

with the alternating term evaluating to -4 if n is even and -2 if n is odd. An alternative formulation of formula (3.17) is

$$s_2^{(-1)}(n) \quad = \quad \left\lfloor \frac{2^{n-1}}{3} \right\rfloor. \tag{3.18}$$

The expected length of the NAF is $\lambda_2(e)$ or $\lambda_2(e) + 1$ according to theorem 3.16. The higher value appears in the following number of cases:

$$\begin{aligned} \text{Prob}[\lambda_{\mathcal{NAF}}(e) = n | \lambda_2(e) = n - 1] \quad &= \quad \frac{2}{3} - \frac{(-1)^n + 3}{3 \cdot 2^{n-1}} \\ &\overset{n\to\infty}{\longrightarrow} \quad \frac{2}{3} \end{aligned} \tag{3.19}$$

Proof:

proof of formula (3.17)

This formula is best proven using induction over n, the length of the NAF of e.

start of induction over the length n of the NAF

Table 3.8 shows the transformation of all binary expansion with $n - 1$ digits and the leading digit 1 into their NAF for $n \in \{1, 2, 3\}$. It can be seen, that for $n = 1$ and $n = 2$, there are no binary expansions that lead to a NAF with more digits, for $n = 3$, there is only one binary expansion of length 2, (11), which counts for $s_2^{(-1)}(3)$. Together, it is

$$\begin{aligned} s_2^{(-1)}(1) \quad &= 0 \quad = \frac{1}{3} \cdot (1 - 1) = \frac{1}{6} \cdot (2^n - 2) \\ s_2^{(-1)}(2) \quad &= 0 \quad = \frac{1}{3} \cdot (2 - 2) = \frac{1}{6} \cdot (2^n - 4) \\ s_2^{(-1)}(3) \quad &= 1 \quad = \frac{1}{3} \cdot (4 - 1) = \frac{1}{6} \cdot (2^n - 2) . \end{aligned}$$

n-1 = 0			**n-1 = 1**			**n-1 = 2**	
$(e)_2$	$(e)_{\mathcal{NAF}}$	$(e)_2$		$(e)_{\mathcal{NAF}}$	$(e)_2$		$(e)_{\mathcal{NAF}}$
		1	$\longrightarrow$	1	10	$\longrightarrow$	10
					11	$\longrightarrow$	$10\bar{1}$

Table 3.8: The first NAF transformations for exact $n-1$ digit numbers for $n-1 = 0,1,2$

assumption

Assume the formula (3.17) to be proven for all $1 \leq k \leq n-1$. Then it also holds for $k = n$.

inductive proof for $k = n$

To be able to apply the induction technique, one has to find a recursive formula, so the value for $k = n$ can in some way be traced back to known values. To find that recursive formula for the sequence $(s_2^{(-1)}(n))_{n\in\mathbb{N}}$, consider counting the elements of $s_2^{(-1)}(n)$ in the following manner:

If n is given, then we first have to determine the number of binary expansion with length $n-1$ that start with a 1, which are 2^{n-2}. Because we're only interested in bit strings that increase the length, we may leave out those strings starting with zero, because according to theorem 3.16, their NAF can have at most $n-1$ digits.

Then we need to know how many of those 2^{n-2} elements have a NAF, that is one digit longer than the binary expansion. We have half of the above number of strings that start with 11, where the NAF transformation will increase the number of digits by one, because all presented algorithms can only solve adjacent nonzeros by moving a carry value to the left. Hence, we have 2^{n-3} many elements already.

In addition to those, we need to look for elements starting with 10, where the NAF transformation produces a 1-carry at the position $e_{\lambda_2(e)-3}$, at the bit directly right of 10. But these are exactly all bit strings of length $n-3$, starting with a 1, whose NAF requires an additional digit on the left, which is already known to be $s_2^{(-1)}(n-2)$ many. Therefore, it is

$$s_2^{(-1)}(n) \quad = \quad 2^{n-3} + s_2^{(-1)}(n-2).$$

Now we have the recursion we need to apply the induction technique.

We have

$$\begin{aligned} s_2^{(-1)}(n) &\overset{a.s.}{=} 2^{n-3} + \frac{1}{6} \cdot \left(2^{n-2} - 3 + (-1)^{n-1}\right) \\ &= \frac{1}{6} \cdot \left(3 \cdot 2^{n-2} + 2^{n-2} - 3 + (-1)^{n+1}\right) \\ &= \frac{1}{6} \cdot \left(2^n - 3 + (-1)^{n+1}\right). \end{aligned}$$

proof of formula (3.18)

As in formula (3.17), the last term evaluates to either $-\frac{1}{3}$ or $-\frac{2}{3}$ and $s_2^{(-1)}(n) \in \mathbb{N}$, the value of the first term is always corrected towards the next lower integer. This is expressed by the Gaussian bracket notation in formula (3.18).

proof of formula (3.19)

The stated probability can be shown by a counting argument, finding the ratio between the set of binary expansion with length $n-1$, whose length is increased when transformed into the NAF – counted by $s_2^{(-1)}(n)$ – and the total number of $(n-1)$-bit strings starting with 1, which is 2^{n-2}. Hence it is

$$\begin{aligned} \text{Prob}[\lambda_{\mathcal{NAF}}(e) = n \mid \lambda_2(e) = n-1] &= \frac{s_2^{(-1)}(n)}{2^{n-2}} \\ &\overset{(3.17)}{=} \frac{2^n - 3 + (-1)^{n+1}}{6 \cdot 2^{n-2}} \\ &= \frac{2}{3} - \frac{(-1)^n + 3}{3 \cdot 2^{n-1}} \\ &\overset{n \to \infty}{\longrightarrow} \frac{2}{3}. \end{aligned}$$

□

The third counter of a set of sparse strings defined in definition 3.17, $s_2^{+}(n)$, that counts the number of unsigned non-adjacent binary digit strings, will be needed for the NAF-based binary method, where it will be necessary to know how many of the possible inputs do not require an inversion. The answer to this question states the following theorem. Due to the fact that the sparseness of single nonzero digits is not a feasible way to define a general NAF, the results are only of interest for the binary digit case.

Theorem 3.20 (determining $s_2^+(n)$)
The number of sparse signed binary digit stings (NAFs) with n digits, which do not contain the negative digit $\bar{1}$, which is the same as the number of sparse unsigned binary digit strings, is the $(n+2)$nd Fibonacci number,

$$s_2^+(n) = F_{n+2} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+2}}{\sqrt{5}}. \tag{3.20}$$

Proof:

According to literature (see for example [BSMM95], p. 708), the numbering of the Fibonacci sequence is assumed to be $F_0 = 0,\ F_1 = 1,\ F_2 = 1,\ F_3 = 2,\ F_4 = 3, \ldots$

The only 1-digit sparse unsigned binary digit strings are 0 and 1, hence $s_2^+(1) = 2 = F_3$.

The only 2-digit sparse unsigned binary digit strings are 00, 01 and 10, and therefore $s_2^+(2) = 3 = F_4$.

For other values of n, it is clear that with the same considerations as in the proof of formula (3.8) in theorem 3.18, a sparse unsigned binary digit string must either start with 0 and may be continued with any sparse unsigned binary digit string of length $n-1$ or it must start with 10 and may be continued with any sparse unsigned binary digit string with $n-2$ digits. This leads to the well known Fibonacci recurrence

$$s_2^+(n) = s_2^+(n-1) + s_2^+(n-2),$$

with the same start values as the traditional Fibonacci sequence, shifted by two positions and hence

$$s_2^+(n) = F_{n+2} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+2}}{\sqrt{5}}.$$

The last equation can be found in many standard books on analysis, for example in [Heu94], p. 378. □

3.4.3 Determining occurrence and distribution of digits

For the generalization of the binary expansion for exponentiation, it is always crucial to know about the Hamming weight of the string that is to be partitioned and used and about the expected number of nonzero or zero windows. These analyses depend on a good knowledge of the expected occurrence of the possible digits and their distribution over the whole string. The following theorems will answer these questions.

Theorem 3.21 (occurrences of the letter 0 in the NAF)
(from Ö. Eğecioğlu and Ç. K. Koç in [EgK90], p.191)
The total number of occurrences of the letter 0 *over all signed binary digit strings of length* n *is*

$$\zeta_n = \frac{(24n+56)\cdot 2^n + (8-3n)\cdot(-1)^n}{27}. \tag{3.21}$$

Proof:

(following Ö. Eğecioğlu and Ç. K. Koç in [EgK90], p.191)
Let $\mathcal{L}$ denote the formal language of all words w over the alphabet $\{\bar{1}, 0, 1\}$ in which none of the patterns $11, 1\bar{1}, \bar{1}1, \bar{1}\bar{1}$ appears. With the same thoughts that build the recursive formula (3.8) in theorem 3.18, it is easy to see that $\mathcal{L}$ satisfies the relation

$$\mathcal{L} \quad = \quad \varepsilon + 1 + \bar{1} + 10\mathcal{L} + \bar{1}0\mathcal{L} + 0\mathcal{L}, \tag{3.22}$$

where ε denotes the empty word and $+$ denotes disjoint union. Let $\lambda(w)$ denote the length of w and $\lambda(w)|_0$ denote the length of w only regarding zeros, hence the number of zeros in w. Consider the generating function

$$f_{\mathcal{L}}(t,x) = \sum_{w\in\mathcal{L}} t^{\lambda(w)} x^{\lambda(w)|_0}.$$

It follows from (3.22) that $f_{\mathcal{L}}$ satisfies

$$f_{\mathcal{L}}(t,x) = 1 + 2t + 2t^2 x f_{\mathcal{L}}(t,x) + tx f_{\mathcal{L}}(t,x),$$

and therefore

$$f_{\mathcal{L}}(t,x) \quad = \quad \frac{1+2t}{1-tx-2t^2x} \tag{3.23}$$

With this generating function $f_{\mathcal{L}}$, the generation function of the sequence ζ_n can easily be found from $f_{\mathcal{L}}(t,x)$ as

$$\sum_{n=0}^{\infty} \zeta_n t^n = \frac{\partial}{\partial x} f_{\mathcal{L}}(t,1),$$

where the substitution $x = 1$ is carried out after the differentiation with respect to x. From the expression (3.23) we obtain the formula

$$\begin{aligned}
\frac{\partial}{\partial x} f_{\mathcal{L}}(t,1) \quad &= \quad \frac{t\cdot(1+2t)^2}{(1-t-2t^2)^2} \\
&= \quad \frac{8}{9}\cdot\frac{1}{(1-2t)^2} - \frac{32}{27}\cdot\frac{1}{1-2t} + \frac{11}{27}\cdot\frac{1}{1+t} - \frac{1}{9}\cdot\frac{1}{(1+t)^2}
\end{aligned}$$

and therefore

$$\zeta_n = \frac{(24n+56)\cdot 2^n + (8-3n)\cdot(-1)^n}{27}.$$

□

Now we can take a closer look an the average Hamming weight of the NAF. The NAF has minimal possible Hamming weight, as it was proved in the last section, while only needing at most one digit more than the ordinary binary expansion. It is of great interest, what the average Hamming weight is and how it compares to the average Hamming weight of the ordinary binary expansion. For this case, there have been interesting results. They state that the binary Hamming weight is greatly reduced compared to the average binary expansion, leading to great improvements when using the classical binary method on top of the NAF (see section 4.1).

Theorem 3.22 (average Hamming weight of the binary NAF)
(from Ö. Eğecioğlu and Ç. K. Koç in [EgK90], pp. 191-192)
The average Hamming weight of the NAF of a given integer is

$$\frac{1}{n}\cdot E\left(\nu_{\mathcal{NAF}}(e)\right) = 1 - \frac{\zeta_n}{n\cdot s_2(n)} \quad\longrightarrow\quad \frac{1}{3}.$$

Proof:

(following Ö. Eğecioğlu and Ç. K. Koç in [EgK90], pp. 191-192)
The fraction

$$\frac{\zeta_n}{n\cdot s_2(n)}$$

describes the average ratio of zeros on n digits in every element in $s_2(n)$. From the formulas (3.10) for $s_2(n)$ and (3.21) for ζ_n, it follows directly that

$$\begin{aligned}
\frac{\zeta_n}{n\cdot s_2(n)} &= \frac{(24n+8)\cdot 2^n + (8-3n)\cdot(-1)^n}{9n\cdot(2^{n+2}+(-1)^{n+1})} \\
&= \frac{2^n\cdot(24n+8)}{2^{n+2}\cdot(9n+(-1)^{n+1}\cdot\frac{9n}{2^{n+2}})} + \frac{(8-3n)\cdot(-1)^n}{9n\cdot(2^{n+2}+(-1)^{n+1})} \\
&\overset{n\to\infty}{\longrightarrow} \frac{24n}{36n} \\
&\Longrightarrow \lim_{n\to\infty}\frac{\zeta_n}{n\cdot s_2(n)} = \frac{2}{3}.
\end{aligned}$$

□

The last result of this section aims to give an idea about the probability distribution of the Hamming weight. It allows to compute the expected values of a certain Hamming weight to appear as the Hamming weight of a random exponent e. The theorem is mentioned to complete the results about the probabilistic behaviour of the NAF.

Theorem 3.23 (probability distribution of $\nu_{\bar{q}}(e)$)
(from S. Arno and F. S. Wheeler in [AW93], p. 1009)
Let ε be a random variable on the space of positive integers, whose q-adic expansion requires exactly n digits. Then for $1 \leq i \leq n$ we have

$$\begin{aligned}
&\mathrm{Prob}(\nu_{\bar{q}}(e) = k) \\
&\quad = \frac{1}{q^n - q^{n-1}} \cdot \\
&\qquad \left(\sum_{b=1}^{k} 2^{b-1}(q-1)^b(q-2)^{k-b} \cdot \binom{n-k}{b-1}\binom{k-1}{b-1} \right. \\
&\qquad \left. + \sum_{b=1}^{k-1} 2^{b-1}(q-1)^b(q-2)^{k-b-1} \cdot \binom{n-k}{b-1}\binom{k-2}{b-1} \right).
\end{aligned}$$

Proof:

Because this theorem does not contribute to the main objective of this thesis, the proof for theorem 3.23 is omitted. It can be found as the proof to theorem 4 in [AW93](p. 1009). □

3.5 Generalizations of the NAF

As a result of theorem 3.18(3), it has been seen that the NAF cannot be generalized to a signed q-adic digit representation (or representation in radix q) in the same manner as the NAF was originally designed. The reason is, that some values cannot be displayed if all nonzero digits of a general NAF have to be non-adjacent, e.g. surrounded by zeros. The proof to theorem 3.18(3) showed that for $e = q + 1$ in a q-adic representation. Therefore, a possible generalization must weaken that requirement.
A number of solutions have been suggested, all of which provide unique minimal representations for any input e. Examples can be found amongst others in [AW93], [Lin98], [Par90] or [CL73].
The representation suggested by J. H. van Lint in [Lin98] is briefly described. It provides a generalized NAF for arbitrary bases $q \in \mathbb{N}$, which is achieved by requiring adjacent digits to be admissible according to the following definition.

Definition 3.24 (admissible pairs of integers)
(from J. H. van Lint in [Lin98], 12.2.1)
Let $b, c \in \mathbb{Z}, |b| < q, |c| < q$. *The pair* (b, c) *is called admissible if one of the following holds*

(i) $b \cdot c = 0$,
(ii) $b \cdot c > 0$ *and* $|b + c| < q$,
(iii) $b \cdot c < 0$ *and* $|b| > |c|$.

For $q = 2$, this definition is equal to the definition of "non-adjacent" (see definition 3.7), because cases *ii* and *iii* cannot appear. Using definition 3.24, the general NAF can be defined as follows:

Definition 3.25 (a general NAF: $\mathcal{NAF}_q$)
(following J. H. van Lint in [Lin98], 12.2.2)
For a given number $e \in \mathbb{Z}$, *a representation of the form*

$$e = \sum_{i=0}^{\infty} e_i \cdot q^i,$$

with $e_i \in \mathbb{Z}, |e_i| < q \ \forall i, e_i = 0$ *for all but finite many* $i's$, $q \in \mathbb{N}, q \geq 2$ *the base of the representation, is called a general NAF for e to the base q if for every* $i \geq 0$, *the pair* (e_{i+1}, e_i) *is admissible.*
The general NAF for e to base q is denoted as $\mathcal{NAF}_q(e)$ *and the length of the representation is defined as*

$$\lambda_{\mathcal{NAF}_q}(e) = \max\{i \in \mathbb{N} \mid e_i \neq 0\} + 1.$$

The above definition provides a unique general NAF for every $e \in \mathbb{Z}$ with minimal Hamming weight equal to the arithmetic weight. For more information about this general NAF see [Lin98], §12.2.

Chapter 4

Exponentiation using the NAF

This last chapter will now use the NAF introduced and analyzed in the last chapter to form the addition-subtraction variants of the binary method and of the m-ary method. Both new variants are analyzed in detail and the results will be compared to those of the traditional variants introduced in chapter 2. As the result of this chapter and of this thesis, the cost comparisons will lead to inequalities, which can be used to determine for every application scenario, whether the binary expansion is superior or inferior to the non-adjacent form as a basis for a chosen exponentiation algorithm.

Given the non-adjacent form, a good basis for exponentiation algorithms seems to be found. The result of theorem 3.22 shows that the Hamming weight of the NAF can be reduced by 33% on average if compared to the ordinary binary expansion. Although, the resulting sequence is sparse, which decreases the chance of long zero runs. Additionally, inversions have to be performed. Therefore, it has to be analyzed, whether the replacement of the binary expansion by the NAF leads to significant improvements in the costs of exponentiation algorithms or not. The following sections will review the binary method presented in section 2.2 and Brauer's method presented in section 2.3 and analyze them assuming the NAF as input. The analyses will show in which cases the methods should be based on the ordinary binary expansion and when they should be based on the NAF. This decision will depend on the ratio of the three operations addition, doubling and inversion. Especially if inversions are free or substantially cheaper than multiplications and squarings – as in the situation of elliptic curve arithmetic – the NAF leads to a substantially improved performance.

4.1 The NAF and the binary method

The analysis of the binary method based on the NAF instead of on the ordinary binary expansion introduces the possibilities of this special representation. Although the binary method has been shown to be inferior to the m-ary method (because it is a special case of the m-ary method), analyzing the NAF based binary method has a justification beyond the purpose of serving as an easy example. There are still applications where memory is limited (like FPGAs), which may suggest not to use an exponentiation algorithm that needs precomputation. In these cases, the binary method will still be the method of choice.
Because the operational costs of this method depends mainly on the Hamming weight of the input, the weight reductions proved in the last section take full effect. Recall algorithm 2.6, which is enhanced here to deal with the digit $\bar{1}$, that indicates the need to compute x^{-1}:

Algorithm 4.1 (NAF based binary method, left to right)

Input: $x,(e)_{\mathcal{NAF}} = (e_{\lambda_{\mathcal{NAF}}(e)-1}, e_{\lambda_{\mathcal{NAF}}(e)-2}, \ldots, e_0),\ e > 0$
Output: x^e

```
Pre-compute x^{-1}
Set A := x                    # e > 0 ensures the most significant bit to be 1
for i from λ_NAF(e) − 2 downto 0 do
    Set A := A^2
    if e_i == 1 then A := A · x
    else if e_i == 1̄ then A := A · x^{-1}
return A
```

4.1.1 Cost analysis

Following the analysis of the traditional binary method (see section 2.2), the operational costs can easily be determined. The following three lemmas will state the results about the three arithmetical operations involved.

Lemma 4.2 (exact and average number of squarings)
For the NAF-based binary method, the measure Q(2) indicating the number of squarings takes the following values if the input number e is i.i.d.:

squarings

exact number	$Q(2) =$	$\lambda_{\mathcal{NAF}}(e) - 1$
worst case number	$Q(2) =$	$\lambda_2(e)$
average number	$Q(2) =$	$\lambda_2(e) - \frac{1}{3}$

Proof:

The exact number is easily determined from the algorithm. The main loop requires one squaring for each round and the for-loop requires $\lambda_{\mathcal{NAF}}(e) - 1$ many rounds.

The worst case number of squarings shows up for an input with the longest possible NAF. As it is the objective to formulate these cases in comparison to the traditional binary method, this happens if the NAF of e takes one digit more than the binary expansion of e, hence in the case where $\lambda_{\mathcal{NAF}}(e) - 1 = \lambda_2(e) + 1 - 1$. Corollary 3.16 showed that $\lambda_{\mathcal{NAF}}(e) \in \{ \lambda_2(e), \lambda_2(e) + 1 \}$.

The average case number of squarings can be determined using theorem 3.19, formula (3.19), which states that the average length of the NAF of a random integer e is $\lambda_2(e) + \frac{2}{3}$ for $\lambda_{\mathcal{NAF}}(e)$ big enough, and therefore $\lambda_{\mathcal{NAF}}(e) - 1 = \lambda_2(e) - \frac{1}{3}$. □

The lemma shows that the NAF-based binary method requires the same or the same plus one number of squarings as the traditional binary method, which requires $\lambda_2(e) - 1$ squarings (see table 2.2 on page 58). But while there is this slight increase in squarings, the following lemma will show that multiplications are reduced dramatically.

Lemma 4.3 (exact and average number of multiplications)
For the NAF-based binary method, the measure A indicating the number of multiplications takes the following values if the input number e is i.i.d.:

	multiplications	
exact number	$A =$	$\nu_{\mathcal{NAF}}(e) - 1$
worst case number	$A =$	$\left\lfloor \frac{\lambda_2(e)}{2} \right\rfloor$
average number	$A =$	$\frac{1}{3} \cdot \lambda_2(e) - \frac{7}{9}$

Proof:

The exact number of multiplications is the same as the number of nonzero digits within the NAF, with the first digit being the initialization value. Therefore, there are always $\nu_{\mathcal{NAF}}(e) - 1$ many multiplications.

This number contains no hidden doublings, as the accumulator is initialized with $(x^2)^2$ (as the NAF starts with 10 unless it contains only one digit), hence, a hidden doubling in lines 5 or 6 of algorithm 4.1 may only occur, if the accumulator becomes x or x^{-1}. Both cases cannot occur, as only multiplications with x^{-1} may reduce the current power of x the accumulator contains, and those reduce the power by

1. This means that if the accumulator represents a power of x greater or equal to 4 (the initial value) before line 5, it is greater or equal to 3 after the multiplication (which therefore cannot be a hidden doubling). After that, the accumulator is squared again, the power doubled, and it again represents a power of x greater or equal to 4. Therefore, no hidden doublings may occur.

The maximal Hamming weight of a NAF can be reached if every second digit is nonzero. As the first digit is bound to be 1, a NAF with maximal Hamming weight follows the pattern

$$(e)_{\mathcal{NAF}} = \begin{cases} 1(0\star)^{\mathsf{a}} & \text{if } \lambda_{\mathcal{NAF}}(e) \text{ is odd} \\ \left\{\begin{array}{c} 1(0\star)^{\mathsf{a}} \text{ plus another zero} \\ \text{inserted anywhere right of} \\ \text{the most significant digit} \end{array}\right\} & \text{if } \lambda_{\mathcal{NAF}}(e) \text{ is even,} \end{cases}$$

with $\star$ denoting any nonzero digit and

$$\mathsf{a} = \left\lfloor \frac{\lambda_{\mathcal{NAF}} - 1}{2} \right\rfloor .$$

The claimed costs follow with the fact that in the worst case, $\lambda_{\mathcal{NAF}}(e) = \lambda_2(e) + 1$. See example 3.10(2) for an example of such a worst case.

The average Hamming weight of the NAF (for a random e) has been determined in theorem 3.22, which states that an expected $\frac{1}{3} \cdot \lambda_{\mathcal{NAF}}(e)$ digits are nonzero. Because the leading digit requires no addition (it is the initial value), there is an expected number of $\frac{1}{3} \cdot \lambda_{\mathcal{NAF}}(e) - 1$ multiplications on lines 5 and 6 of algorithm 4.1. Theorem 3.19 states that the average value for $\lambda_{\mathcal{NAF}}(e)$ is $\frac{2}{3} + \lambda_2(e)$, hence

$$\begin{aligned} E(\nu_{\mathcal{NAF}}(e) - 1) &= \frac{1}{3} \cdot (\frac{2}{3} + \lambda_2(e)) - 1 \\ &= \frac{1}{3} \cdot \lambda_2(e) - \frac{7}{9} \end{aligned}$$

□

Recall that the traditional binary method requires $\lambda_2(e) - 1$ multiplications in the worst case and $\frac{1}{2} \cdot (\lambda_2(e) - 1)$ multiplications in the average case (according to table 2.2 on page 58). It can be seen that the worst case of the NAF-based binary method is about the same as the former average case, while the new average case requires 33% less the number of multiplications expected with the traditional binary method. If the inversion doesn't level out this advantage, the NAF-based binary method will be much faster.

Lemma 4.4 (exact and average number of inversions)
For the NAF-based binary method, the measure I indicating the number of inversions takes the following values if the input number e is i.i.d.:

inversions		
exact number	$I =$	*1*
worst case number	$I =$	*1*
average number	$I =$	*1*

Proof:

The algorithm always computes x^{-1} once in the beginning as a precomputation step independently from the input. There are no other inversions necessary (if x^{-1} is requested within the loop, the precomputed value is assumed to be present through a table lookup). $\square$

One could be led to think of a possible improvement of the costs of inversions by noting that there are a number of inputs that do not contain the digit $\bar{1}$ and, hence, do not need the computation of x^{-1}. The number of these inputs can be easily determined as $s_2^+(n-2)$, as the leading digit is bound to be 1, requiring the second one to be 0, which may be followed be any unsigned binary digit string with at most $n-2$ digits, which has been defined as $s_2^+(n-2)$.
Algorithm 4.1 could now replace the mandatory computation of x^{-1} in line 1 by a scan through the NAF and a computation of x^{-1} if and only if the digit $\bar{1}$ appears. As the implementation would surely realize the availability of the value of x^{-1} through table lookup, it is no problem to include that precomputation step in the main loop, computing x^{-1} when it is first requested. In this way, the scan through the NAF would require no additional costs. Additionally, as the analyses within this thesis only count arithmetical operations, the scan would be negligible anyway.
Although this feature of an input sensitive computation of x^{-1} could be implemented with very little additional costs, it is not feasible to include it in the analyzed algorithm, because for practical applications the gain is diminishingly small. As the number of inputs without the digit $\bar{1}$ is $s_2^+(n-2)$, which has been shown to be equal to the n-th Fibonacci-number F_n in theorem 3.20, the ratio between these numbers, where the inversion could be saved, and the number of all possible inputs with exactly n digits, which is $s_2(n-2)$ (due to the fact that they start with 10 followed by any NAF of at most $n-2$ digits), tends towards 0 very quickly:

Corollary 4.5 (ratio between $s_2^+(n-2)$ and $s_2(n-2)$)
The number of NAFs without the digit $\bar{1}$ is a diminishing part of the number

of all possible NAFs with at most n digits:

$$\frac{s_2^+(n)}{s_2(n)} = \frac{F_{n+2}}{s_2(n)} \longrightarrow 0$$

Proof:

It is

$$\begin{aligned}
\frac{F_{n+2}}{s_2(n)} &\overset{(3.20),(3.10)}{=} \frac{3}{\sqrt{5}} \cdot \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+2}}{2^{n+2} + (-1)^{n+1}} \\
&\leq \frac{3}{\sqrt{5}} \cdot \left[\frac{(0.80902)^{n+2}}{2^{n+2} \pm 1} - \frac{\left(\frac{1-\sqrt{5}}{4}\right)^{n+2}}{1 \pm \frac{1}{2^{n+2}}} \right] \\
&\longrightarrow 0
\end{aligned}$$

as the first fraction clearly tends towards 0 and the second one's numerator is $\sim (-1)^{n+2} \cdot (0.309)^{n+2}$. □

For the lengths of NAFs used in practical applications, such as $n = 512$ (RSA), the average saving is less than 10^{-47} of one inversion. Due to this reason, the input sensitive inversion is not analyzed.
Combining the three lemmas above, the analysis of the NAF-based binary method yields the following results:

Theorem 4.6 (analysis of the NAF-based binary method)
The NAF-based binary method performed on a random input number $e \in \mathbb{N}$ requires the following costs:

measure	**exact costs**	**average costs** *for $\lambda_2(e)$ big enough*	**worst case costs**
I	1	1	1
$Q(2)$	$\lambda_{\mathcal{NAF}}(e) - 1$	$\lambda_2(e) - \frac{1}{3}$	$\lambda_2(e)$
A	$\nu_{\mathcal{NAF}}(e) - 1$	$\frac{1}{3} \cdot \lambda_2(e) - \frac{7}{9}$	$\left\lfloor \frac{\lambda_2(e)}{2} \right\rfloor$

Proof:

The results have been shown in lemma 4.2, lemma 4.3 and lemma 4.4. □

With these costs known, the NAF-based binary method can be compared to the traditional binary method. This thesis examines only the number of operations in the underlying field of the exponentiation problem, therefore

the creation of the NAF is not regarded, the NAF is assumed to be given on input. In spite of this restriction, the comparison of the two methods is not trivial, because the main difference is in substituting a number of multiplications by an inversion. To be able to analyze this substitution, the costs of the operations have to be regarded. Because they differ depending on the underlying field, a comparison must depend on them.
If the cost measure functions are known, it is possible to give a cost comparison, which can be the basis for a fast algorithm that chooses whether to apply the NAF-based binary method or the traditional binary method. This will be done together with Brauer's method, which will be examined first in the next section.

4.2 The NAF and Brauer's approach

The m-ary method may as well be based upon the NAF instead of on the traditional binary expansion. A general m-adic representation is not common in practical applications, because the binary or 2^k-ary representation is natural to computers. As powers of 2 are used as bases, this method will, similar to section 2.3.5, be referenced as the NAF-based Brauer method.
As a first step, the NAF is partitioned into windows, which are all of length d digits, for some input value $d \in \mathbb{N}_{>0}$. As practical applications may either use the optimal value determined in section 2.3.6 or the computer word size, which only means that the computer has to interpret several successive digits as one number, the partitioning does not require substantial time and as the analyses within this thesis only regard arithmetical costs, the partitioning step will not be included within the algorithm. Instead, the prepartitioned NAF will be assumed to be present and it will be referred to according to the following definition.

Definition and theorem 4.7 (*d*-digit partitioned NAF: *d*-NAF)
Given the NAF

$$(e)_{\mathcal{NAF}} = (e_{\lambda_{\mathcal{NAF}}(e)-1}, e_{\lambda_{\mathcal{NAF}}(e)-2}, \ldots, e_1, e_0)$$

of a given integer $e \in \mathbb{N}$ and an integer $d \in \mathbb{N}_{>0}$, the d-NAF is defined as the d-digit partitioned NAF, where from the rightmost digit to the leftmost digit every d digits are combined into one window. The leftmost window may include less than d digits. Each window can be interpreted as a signed 2^d-ary digit:

$$(e)_{d-\mathcal{NAF}} = (E_{\lambda_{d-\mathcal{NAF}}(e)-1}, E_{\lambda_{d-\mathcal{NAF}}(e)-2}, \ldots, E_1, E_0)$$

Where

$$\begin{aligned} E_i &= (e_{(i+1)\cdot d-1}, e_{(i+1)\cdot d-2}, \ldots, e_{i\cdot d+1}, e_{i\cdot d}) && \text{for } 0 \le i < \lambda_{d-\mathcal{NAF}}(e) - 1 \\ &= (e_{\lambda_{\mathcal{NAF}}(e)-1}, e_{\lambda_{\mathcal{NAF}}(e)-2}, \ldots, e_{i\cdot d+1}, e_{i\cdot d}) && \text{for } i = \lambda_{d-\mathcal{NAF}}(e) - 1. \end{aligned}$$

$$\lambda_{d-\mathcal{NAF}}(e) = \left\lceil \frac{\lambda_{\mathcal{NAF}}(e)}{d} \right\rceil \quad (4.1)$$

$$\text{length}(E_{\lambda_{d-\mathcal{NAF}}(e)-1}) = \lambda_{\mathcal{NAF}}(e) \text{ rem } d \quad (4.2)$$

Proof:

Division with remainder of $\lambda_{\mathcal{NAF}}(e)$ by d gives $\lambda_{\mathcal{NAF}}(e) = k \cdot d + r$ for some $k, r \in \mathbb{N}$, $r < d$. Hence, there are k windows with exactly d digits and one window with only r digits. If $r = 0$, it is $\lambda_{d-\mathcal{NAF}}(e) = \frac{\lambda_{\mathcal{NAF}}(e)}{d} = \lceil \frac{\lambda_{\mathcal{NAF}}(e)}{d} \rceil$ and if $0 < r < d$, there are $\lambda_{d-\mathcal{NAF}}(e) = k + 1 = \lceil \frac{\lambda_{\mathcal{NAF}}(e)}{d} \rceil$ windows, proving the first claimed property. As rem is the operation returning the remainder r, the second claimed property is also proven. □

Example:

Given a NAF

$$(e)_{\mathcal{NAF}} = 10010\bar{1}00\bar{1}01010000010\bar{1},$$

the 2-NAF is

$$(e)_{2-\mathcal{NAF}} = 10\ 01\ 0\bar{1}\ 00\ \bar{1}0\ 10\ 10\ 00\ 01\ 0\bar{1}.$$

The following algorithm enhances algorithm 2.7 to deal with the NAF as input.

Algorithm 4.8 (exponentiation, Brauer-method with NAF)

Input: x, $(e)_{d-\mathcal{NAF}} = (E_{\lambda_{d-\mathcal{NAF}}(e)-1}, E_{\lambda_{d-\mathcal{NAF}}(e)-2}, \ldots, E_0)$, $e > 0$, *window length* $d \in \mathbb{N}$

Output: x^e

```
Pre-compute all needed digits.
Set A := x^(E_{λ_{d-NAF}(e)-1})
for i from λ_{d-NAF}(e) - 2 downto 0 do
    Set A := A^(2^d)
    if E_i ≠ 0 then A := A · x^(E_i)
return A
```

Cost analysis

The analysis of the operational costs of this algorithm will be presented in two parts: the analysis of the precomputation and the analysis of the costs in the main part of the algorithm.

4.2.1 Analysis of the precomputation

The traditional Brauer method gains superiority over the binary method by performing precomputation steps in order to save steps within the main loop. This concept is also the basic concept of the NAF-based Brauer method. Here, different values may appear as window values, because now also negative values are possible. An optimal algorithm may only compute those window values which appear as windows within the partitioned NAF. But as the probability that some windows won't be needed decreases rapidly with larger NAFs (and practical applications use large NAFs of 128 - 4096 binary digits), the extended overhead of an input sensitive precomputation is not justified. Hence, all possible values should be computed.

The first task is to examine how that can be done. Differently from the traditional Brauer method, where all possible window values may be computed by successive multiplication with x, the special structure of the NAF windows, which are themselves sparse binary strings, requires a more specialized approach. The next algorithm will present an approach to compute and store all these values. This can be done using a ternary tree structure, where the three possible branches are used for the three possible digits 1, 0 and $\bar{1}$. The window values are stored within the tree in the manner depicted in figure 4.1.

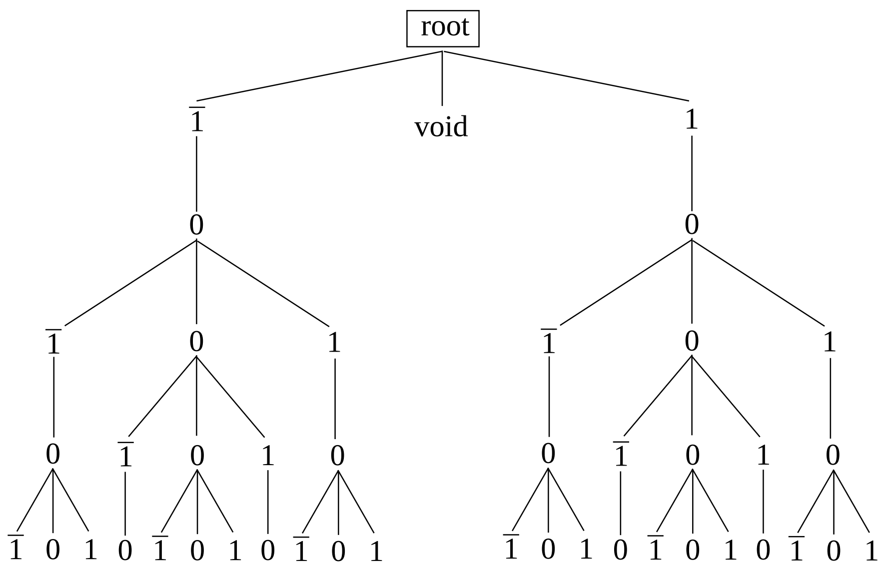

Figure 4.1: The first 5 levels of the NAF-tree computation

All possible window values have a sparse signed binary representation using at most d digits, leading zeros being acceptable. The tree stores all possible sparse signed binary strings with exact lengths from 1 to d, which start with a nonzero digit. Hence, windows with leading zeros have their values stored at the end of the path from the root to a knot determined by their digit sequence starting at the leftmost nonzero digit. For example, the value of `001101` can be found at the last knot of the path `1101` from the root.

Using this approach, no unneeded intermediate values are computed, every computed value gives a new possible window value. This is not true for all addition-subtraction algorithms, for example the sliding window algorithms need to precompute x^2 in order to precompute all possible window values efficiently, although x^2 never appears as a window value. They precompute all odd window values, but need x^2 to skip the even numbers between them (see section 1.2.3).

Of course, the knowledge about the set of all possible NAF strings from theorem 3.18(3) could also lead to the precomputation of all these values without regarding representation. But then every time a value needs to be looked up, the NAF of the value must be computed.

During the analysis of the precomputation steps of the traditional Brauer method in section 2.3.1, it has been shown that there are different ways to perform the precomputation, especially by exchanging different kinds of operations, e.g. $x^4 = (x^2)^2 = (x^3) \cdot x$. The same possibility arises with the NAF-based Brauer approach, where the inversion plays the major role.

One of the two different ways to build up the tree depicted in figure 4.1 is to compute x^{-1} and then compute level after level by first squaring (and producing the child with digit `0`) and then, if the current digit is zero, multiplying the produced `0`-knot with x and x^{-1}, hence producing the other two childs $\bar{1}$ and 1. On the other hand, if the first digit is $\bar{1}$, one could create only half the tree (starting with 1) and produce the other half by inverting the corresponding knot's value on the first half. Both approaches have different costs and the cost of the inversion is crucial to the decision about which algorithm to use.

To give exact results about the precomputation, the task to be performed is defined within the next definition:

Definition 4.9 (precomputation AS chain $\bar{\gamma}(n)$)

In the precomputation step, all values between $-(n-1)$ and $n-1$ for some maximal value $n \in \mathbb{N}$ have to be precomputed in one addition-subtraction chain. The window values x^k for $-n < k < n$ are then computed according to that addition-subtraction chain.

Let

$$\bar{\Gamma}(n) \quad := \quad \{ \; \bar{\chi}(-(n-1), \dots, -1, 0, 1, \dots, (n-1)) \; \}$$

denote the set of all possible addition-subtraction chains, that compute all

values between $-(n-1)$ *and* $n-1$. *Elements from* $\bar{\Gamma}(n)$ *will usually be denoted as* $\bar{\gamma}(n)$.
Analogously to definition 2.10, let $A_{\bar{\gamma}(n)}$ *denote the number of additions, let* $Q_{\bar{\gamma}(n)}(q)$ *denote the number of q-steps and let* $I_{\bar{\gamma}(n)}$ *denote the number of inversions in an addition-subtraction chain* $\bar{\gamma}(n) \in \bar{\Gamma}(n)$ *and let*

$$c(\bar{\gamma}(n)) \quad := \quad c(Q(q)) \cdot Q_{\bar{\gamma}(n)}(q) + c(A) \cdot A_{\bar{\gamma}(n)} + c(I) \cdot I_{\bar{\gamma}(n)}$$

denote the costs of the addition-subtraction chain $\bar{\gamma}(n) \in \bar{\Gamma}(n)$.
For the precomputation of the NAF-based Brauer method, all values from the ternary tree in the above described algorithm must be computed. For

$$\hat{n}(d) := \frac{1}{3} \cdot \left(2^{d+1} + \frac{(-1)^{d+1} - 3}{2} \right), \tag{4.3}$$

$\bar{\Gamma}(\hat{n}(d)+1)$ *is the set of all addition-subtraction chains that performs the precomputation step of the NAF-based Brauer method according to the described algorithm, because the set of all sparse signed binary digit strings with at most d digits has been defined as* $N_2(d)$ *and it has been shown to be* $N_2(d) = \{-\hat{n}(d), \ldots, \hat{n}(d)\}$ *(see theorem 3.18(3)).*

The following lemma will now analyze the two most important ways to compute a $\bar{\gamma}(\hat{n}(d)+1) \in \bar{\Gamma}(\hat{n}(d)+1)$ using mainly additions and doublings and either only one inversion or 50% inversions.

Lemma 4.10 (cost analysis: NAF-based Brauer precomputation)

1. **choosing minimal inversions:**
   The NAF-based Brauer precomputation can be done with the following costs, e.g. there exists an addition-subtraction chain $\bar{\gamma}_1(\hat{n}(d)+1) \in \bar{\Gamma}(\hat{n}(d)+1)$ *such that*

$$\begin{aligned}
I_{\bar{\gamma}_1(\hat{n}(d)+1)} &= 1 \\
Q_{\bar{\gamma}_1(\hat{n}(d)+1)}(2) &= \begin{cases} 0 & for\ d=1 \\ s_2(d-1)-1 & for\ d>1 \end{cases} && (4.4) \\
&= \frac{1}{3} \cdot \left(2^{d+1} + (-1)^d \right) - 1 && (4.5) \\
A_{\bar{\gamma}_1(\hat{n}(d)+1)} &= 2 \cdot (s_2(d-2)-1) && (4.6) \\
&= \frac{2}{3} \cdot \left(2^d + (-1)^{d-1} \right) - 2 && (4.7)
\end{aligned}$$

2. **choosing maximal inversions:**
   The NAF-based Brauer precomputation can be done with the following

costs, e.g. there exists an addition-subtraction chain $\bar{\gamma}_2(\hat{n}(d)+1) \in \bar{\Gamma}(\hat{n}(d)+1)$ *such that*

$$
\begin{aligned}
I_{\bar{\gamma}_2(\hat{n}(d)+1)} &= \frac{1}{3}\cdot\left(2^{d+1}+\frac{(-1)^{d+1}}{2}\right)-\frac{1}{2} && (4.8)\\
&= \frac{1}{2}\cdot s_2(d)-\frac{1}{2} && (4.9)\\
Q_{\bar{\gamma}_2(\hat{n}(d)+1)}(2) &= \begin{cases} 0 & for\ d=1\\ \frac{1}{2}\cdot(s_2(d-1)-1) & for\ d>1\end{cases} && (4.10)\\
&= \frac{1}{6}\cdot\left(2^{d+1}+(-1)^{d}\right)-\frac{1}{2} && (4.11)\\
A_{\bar{\gamma}_2(\hat{n}(d)+1)} &= s_2(d-2)-1 && (4.12)\\
&= \frac{1}{3}\cdot\left(2^{d}+(-1)^{d-1}\right)-1 && (4.13)
\end{aligned}
$$

Proof:

proof of part 1

As the described algorithm is an algorithm constructing addition chains, the chain constructed by the first version of the algorithm proves the first claim if it shows the claimed costs. In this version of the algorithm, it is obvious that only one inversion is needed, because only x^{-1} must be computed and stored, no other inversion operation takes place as long as x^{-1} is still known.

To prove the other formulas, formula (4.4) is shown using induction over the number of digits. The other formulas will follow. Note that the formulas are derived from the formulas shown to be true in theorem 3.18 with respect to the fact that the values for x^0, x^1 and x^{-1} don't have to be computed using squarings or multiplications (or doublings and additions respectively).

start of induction over the digit length d

Figure 4.1 shows the first five levels of the tree that is computed in algorithm 4.11. Every zero is created by squaring the value of the parent knot, every nonzero digit is created by multiplying the neighboring zero value with x or x^{-1} respectively. By counting all zeros up to tree level d, the value of $Q_{\bar{\gamma}_1(\hat{n}(d)+1)}(2)$ can be found. We have

$$
\begin{aligned}
Q_{\bar{\gamma}_1(\hat{n}(1)+1)}(1) &= 0\\
Q_{\bar{\gamma}_1(\hat{n}(1)+1)}(2) &= 2\\
&= s_2(1)-1 = 2\cdot 2-1-1 = 2
\end{aligned}
$$

with formula (3.8) stating that $s_q(1) = 2q-1$.

assumption

Assume formula (4.4) to be true for all $1 < k \leq d$. Then it can also be shown to be true for $k = d + 1$.

inductive proof for $k = d + 1$

The number of squarings needed to construct the tree up to level $d+1$ is equal to the number of squarings needed up to level d plus the number of squarings that are needed within the $(d+1)$st level. The first value is known from the assumption, the latter can be found by noting that due to the construction of the tree, every sparse signed binary digit string with exactly d digits is the basis for a sparse signed binary digit string with exactly $d + 1$ digits ending in zero. This can be done by simply adding a 0 to the representation, which equals the doubling of the value in the addition-subtraction chain or the squaring of the value in the exponentiation algorithm. As the number of leaves in a tree with depth d is equal to the number of sparse signed binary digit strings which need exactly d digits, it can be computed by subtracting $s_2(d-1)$, the total number of sparse signed binary digit strings with at most $d - 1$ digits from $s_2(d)$, the total number of sparse signed binary digit strings with at most d digits. Therefore, the following formula leads to the proof:

$$\begin{aligned}
Q_{\bar{\gamma}_1(\hat{n}(d+1)+1)}(2) &= Q_{\bar{\gamma}_1(\hat{n}(d)+1)}(2) + s_2(d) - s_2(d-1) \\
&= s_2(d-1) - 1 + s_2(d) - s_2(d-1) \\
&= s_2(d) - 1
\end{aligned}$$

The other formula can now easily be derived. Using the equation (3.10) from theorem 3.18 for $s_2(d)$,

$$\begin{aligned}
Q_{\bar{\gamma}_1(\hat{n}(d)+1)}(2) &= s_2(d-1) - 1 \\
&= \frac{1}{3} \cdot \left(2^{d+1} + (-1)^d\right) - 1,
\end{aligned}$$

which proves formula (4.5).

The formulas for $A_{\bar{\gamma}_1(\hat{n}(d)+1)}$ are derived noting that it is the same as the number on nonzero knots within the ternary tree up to level d, without the knots corresponsing to x and x^{-1}. As the tree is created, every zero up to level $d - 1$ leads to the creation of two nonzero knots in the next level. Hence, the searched value is

$$\begin{aligned}
A_{\bar{\gamma}_1(\hat{n}(d)+1)} &= 2 \cdot Q_{\bar{\gamma}_1(\hat{n}(d-1)1+)}(2) \\
&= 2 \cdot (s_2(d-2) - 1) \\
&\overset{(3.10)}{=} \frac{2}{3} \cdot \left(2^d + (-1)^{d-1}\right) - 2,
\end{aligned}$$

proving formulas (4.6) and (4.7). Note that this number cannot contain hidden doublings, as every step creates a new element (see section 1.1.1).

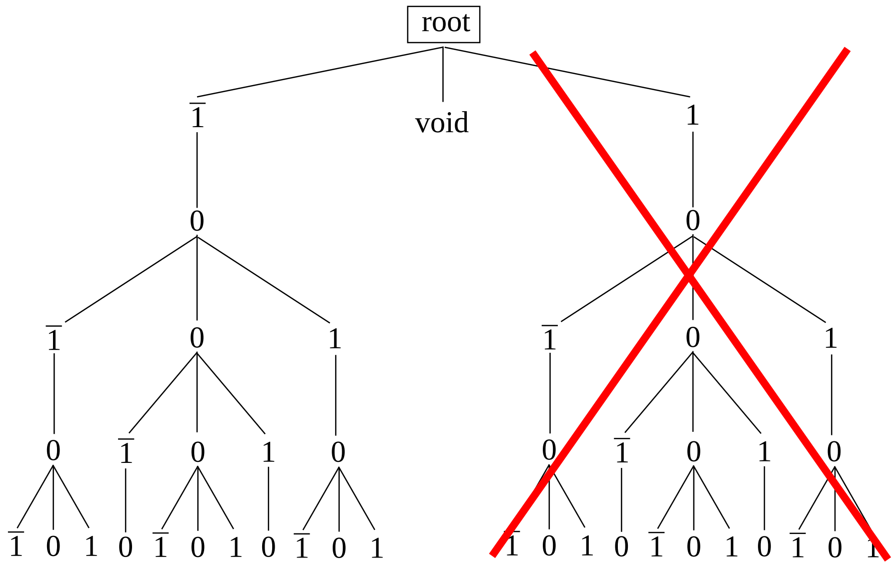

Figure 4.2: The first 5 levels of the pruned NAF-tree computation

<u>**proof of part 2:**</u>

As the ternary tree is completely symmetrical, pruning one of the two major branches will result in exactly half of the costs for both squarings and multiplications (see figure 4.2). Therefore, formulas (4.10), (4.11), (4.12) and (4.13) are derived by simply dividing the corresponding formulas of part 1 by 2.

For the claimed result for $I_{\bar{\gamma}_2(\hat{n}(d)+1)}$, all values on the right side of the tree depicted in figure 4.2 are computed using inverses of the corresponding values of the left side. The corresponding value can be found by following the path of the negated digit sequence, which is the additive inverse of the right hand side digit sequence. As the value for x is not computed using any operation, the value of x^{-1} must be computed explicitly, for it is needed for the creation of the left hand side. All other right hand side values can be counted by adding the number of multiplications and squarings needed to build up the left hand side. The result gives the total number of inversions. Therefore, we have

$$\begin{aligned}
I_{\bar{\gamma}_2(\hat{n}(d)+1)} &= 1 + Q_{\bar{\gamma}_2(\hat{n}(d)+1)}(2) + A_{\bar{\gamma}_2(\hat{n}(d)+1)} \\
&= 1 + \frac{1}{6} \cdot \left(2^{d+1} + (-1)^d\right) - \frac{1}{2} + \frac{1}{3} \cdot \left(2^d + (-1)^{d-1}\right) - 1 \\
&= 1 + \frac{1}{3} \cdot \left(2^d + 2^d + \frac{(-1)^d + 2 \cdot (-1)^{d-1}}{2}\right) - \frac{1}{2} - 1 \\
&= 1 + \frac{1}{3} \cdot \left(2^{d+1} + \frac{-(-1)^{d+1} + 2 \cdot (-1)^{d+1}}{2}\right) - \frac{3}{2} \\
&= \frac{1}{3} \cdot \left(2^{d+1} + \frac{(-1)^{d+1}}{2}\right) - \frac{1}{2} \\
&\text{(proving formula (4.8))} \\
&= \frac{1}{2} \cdot \frac{1}{3} \cdot \left(2^{d+2} + (-1)^{d+1}\right) - \frac{1}{2} \\
&\overset{(3.10)}{=} \frac{1}{2} \cdot s_2(d) - \frac{1}{2} \\
&\text{(proving formula (4.9))}
\end{aligned}$$

□

Note that the costs of the precomputation step do not depend on the input, therefore, these values apply for the exact analysis, the average and the worst case. Both approaches can easily be implemented as the following two algorithms show.
The algorithms are noted in a C++-like manner, assuming an abstract data structure that implements the ternary tree as an object containing nodes with the member value that stores the value of the knot. void is used to denote an unused branch (like the keyword NULL in C++). If a value is set, the creation of the corresponding data object is assumed, unset knots are assumed to be void. The left branch is used to care for a leading $\bar{1}$, the right branch handles the leading 1 and the middle branch handles the 0.
Algorithm 4.11 implements the first part of lemma 4.10, algorithm 4.12 implements the second part of lemma 4.10.

Algorithm 4.11 (NAF-Brauer precomputation, few inversions)

Input: *x, window width d*

Output: *T, a tree containing all precomputed window values for windows with length d*

```
Initialize the ternary tree structure T
Compute and store x^(-1)
Set T.left.value := x^(-1)
Set T.middle := void
Set T.right.value := x
```

```
For i from 2 to d do
    For all leaves L of the tree T do
        Set L.middle.value := (L.value)^2
        If (L.value == 0) then
            Set L.left.value := L.middle.value · x^-1
            Set L.right.value := L.middle.value · x
return T
```

Algorithm 4.12 (NAF-Brauer precomputation, many inversions)

Input: *x, window width d*

Output: *T, a tree containing all precomputed window values for windows with length d*

```
Initialize the ternary tree structure T
Compute and store x^-1
Set T.left.value := x^-1
Set T.middle := void
Set T.right.value := x
For i from 2 to d do
    For all leaves L on the left hand side of the tree T do
        Set L.middle.value := (L.value)^2
        If (L.value == 0) then
            Set L.left.value := L.middle.value · x^-1
            Set L.right.value := L.middle.value · x
    For all leaves L on the right hand side of the tree T do
        Set L' := corresponding knot on the left side of the tree
        Set L.middle.value := inverse(L'.middle.value)
        If (L.value == 0) then
            Set L.left.value := inverse(L'.left.value)
            Set L.right.value := inverse(L'.right.value)
return T
```

The two different approaches to perform the precomputation presented here introduce upper bounds on the arithmetical operations. The total number of operations is fixed and it cannot be reduced, as every operation creates a new window value.

The decision whether to choose algorithm 4.11 or algorithm 4.12 will depend on the individual costs of the arithmetical operations. Beforehand, the analysis of the main part of the NAF-based Brauer method is still to be presented.

4.2.2 Analysis of the main part

Similarly to the binary method, the arithmetical costs will be presented for each operation in the following three lemmas:

Lemma 4.13 (exact and average number of squarings)
For the NAF-based Brauer method, the measure Q(2) indicating the number of squarings takes the following values if the input number e is i.i.d.:

	squarings		
exact number	$Q(2) =$	$d \cdot (\lambda_{d-\mathcal{NAF}}(e) - 1)$	$= d \cdot \left\lceil \frac{\lambda_{\mathcal{NAF}}(e)}{d} \right\rceil - d$
worst case number	$Q(2) =$	$d \cdot (\lambda_{d-\mathcal{NAF}}(e) - 1)$	$= d \cdot \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor$
average number	$Q(2) =$	$\frac{d}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \frac{2d}{3} \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - d$	

Proof:

The total number of exponentiations to a 2^dth power in line 4 of algorithm 4.8 is $\lambda_{d-\mathcal{NAF}}(e) - 1$. As every one of these exponentiations requires d squarings (which is optimal, because 2^d is a power of two, where the binary method is optimal), the exact number of squarings necessary is $d \cdot \lambda_{d-\mathcal{NAF}}(e) - d$. Formula (4.1) traces $\lambda_{d-\mathcal{NAF}}(e)$ back to $\lambda_{\mathcal{NAF}}(e)$ and therefore

$$Q(2) \overset{(4.1)}{=} d \cdot \left\lceil \frac{\lambda_{\mathcal{NAF}}(e)}{d} \right\rceil - d.$$

The above formula for the exact case also applies to the worst case, where it may be assumed that for the input e, we have $\lambda_{\mathcal{NAF}}(e) = \lambda_2(e) + 1$, which gives the following number of squarings:

$$Q(2) = d \cdot \left\lceil \frac{\lambda_2(e) + 1}{d} \right\rceil - d = d \cdot \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor \qquad (4.14)$$

To show this equality, consider the division with remainder $\lambda_2(e) = k \cdot d + r$, $0 \le r < d$. The right hand side of formula (4.14) always evaluates to $d \cdot k$, as the lower Gaussian brackets always result in the highest integer below or equal to the fraction, which is always k. For the left hand side, the additional 1 has the same effect as if $0 < r \le d$ was given, which causes the upper Gaussian brackets to result in $k+1$ always, as this is the lowest integer above or equal to the fraction.

In the average case, the value of $\lambda_{d-\mathcal{NAF}}(e)$ takes the same value as $\lambda_{2^d}(e)$, if d doesn't divide $\lambda_2(e)$ and hence the leftmost window does not contains d digits. Any possible increase in length has no effect on the number of windows.

However, if $\lambda_2(e)$ is a multiple of d, the number of squarings is different. As the NAF may require an additional digit, and the leftmost window already contains d digits, a new window might be created. In this case, the former leftmost window can no longer be used for initialization of the accumulator, but has to be handled as an ordinary window, hence, it requires d more squarings. The probability of the NAF requiring an additional digit has been determined in theorem 3.19. It states that this happens in

$$\frac{2}{3} - \frac{(-1)^{\lambda_2(e)} + 3}{3 \cdot 2^{\lambda_2(e)-1}} \quad \longrightarrow \quad \frac{2}{3}$$

of all cases. The limit may be taken for the formula as the correction term tends to zero exponentially. As the expected additional $\frac{2}{3} \cdot d$ squarings only have to be added in the case where d divides $\lambda_2(e)$, the following factor is used. It evaluates to 1 iff $\lambda_2(e)$ is a multiple of d and to zero otherwise:

$$\left(1 - \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor\right)$$

With this factor, the following formula corrects the expected number of squarings if d divides $\lambda_2(e)$ and leaves it to the usual case otherwise:

$$\begin{aligned}
&d \cdot \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) + \left(1 - \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor\right) \cdot \frac{2}{3} \cdot d \\
&\quad = \quad \frac{1}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil - \frac{d}{3} + \frac{2}{3} \cdot d \cdot \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor \\
&\quad \overset{(4.14)}{=} \quad \frac{d}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil - \frac{d}{3} + \frac{2}{3} \cdot d \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - \frac{2}{3} \cdot d \\
&\quad = \quad \frac{d}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \frac{2d}{3} \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - d
\end{aligned}$$

□

Recall that the average number of squarings for the traditional Brauer method (see results in table 2.7) was

$$Q(2) \quad = \quad d \cdot \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor - d.$$

This reveals that the additional digit the NAF may have compared to the binary expansion might increase the number of squarings needed by d. This

is the case whenever $\lambda_2(e)$ is divisible by d and hence the leftmost window of the traditional partitioning has d digits. If this is not the case, the number of squarings is equal to the number of the traditional Brauer method. This means that for applications with numbers of a certain length, the choice of the right method may vary not only depending on the operational costs but also on the binary length of the inputs (see example 4.21(1)).
The fact that the number of squarings is mostly the same for both Brauer methods is not surprising, as the number of squarings is not likely reduced by any addition-subtraction chain method. The main sources for savings are the precomputation and the multiplications.

Lemma 4.14 (exact and average number of multiplications)
For the NAF-based Brauer method, the measure A indicating the number of multiplications takes the following values if the input number e is i.i.d.:

	multiplications	
exact number	$A = \nu_{d-\mathcal{NAF}}(e) - 1$	
worst case number	$A = \lambda_{d-\mathcal{NAF}}(e) - 1$	$= \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor$
average case number	$A = \frac{1}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d}\right) \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - \frac{2^d - \frac{5}{3}}{2^d}$	

The average number of zero windows is about 33% higher than with the traditional Brauer method. This generalizes the results from the binary method (theorem 3.22).

Proof:

Line 5 of algorithm 4.8 accounts for one multiplication whenever a window value is nonzero. Hence the Hamming weight of the d-NAF without the leftmost initializing window determines the number of multiplications. As the main step references elements prior created in the precomputation, the proof that no hidden doublings are miscounted as additions will be given in theorem 4.16, when the addition-subtraction chains for precomputation and main exponentiation step are combined.

In the worst case, the Hamming-weight will be as high as the number of windows, hence $\lambda_{d-\mathcal{NAF}}(e) - 1$ multiplications. As formula (4.1) states that

$$\lambda_{d-\mathcal{NAF}}(e) = \left\lceil \frac{\lambda_{\mathcal{NAF}}(e)}{d} \right\rceil,$$

and in the worst case, $\lambda_{\mathcal{NAF}}(e) = \lambda_2(e)+1$, the claimed formula follows with

$$A \;=\; \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - 1 \overset{(4.14)}{=} \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor .$$

Note that $\lambda_{\mathcal{NAF}}(e) \in \{\lambda_2(e),\ \lambda_2(e)+1\}$ according to theorem 3.16.

In the average case, it is feasible to determine the average number of zero windows in the d-NAF and subtract that from the overall number of windows in order to get the average number of nonzero windows, which indicates the number of multiplications.

Let W_i denote the i-th window in the d-digit partitioned binary expansion of the input exponent e (Brauer partition) and let W_i' denote the same window after the binary expansion has been transformed into the NAF. As the d-NAF is partitioned just like the binary expansion, W_i' denotes the i-th window in the NAF. Note that the NAF may have an additional digit and therefore also an additional window. It is the purpose to give results comparable to those of the traditional Brauer method, hence, the number of multiplications will be counted by determining how the number of zero windows changes during the transformation into the NAF. This poses the task to compute the probabilities

$$\begin{aligned} \mathrm{Prob}(Val(W_i')=0) &= \mathrm{Prob}(Val(W_i')=0 \mid Val(W_i)=0) \\ &+ \mathrm{Prob}(Val(W_i')=0 \mid 1 \le Val(W_i) \le 2^d-2) \\ &+ \mathrm{Prob}(Val(W_i')=0 \mid Val(W_i)=2^d-1) \end{aligned} \tag{4.15}$$

for all windows W_i' in the NAF.

That task will be carried out in three steps: First, the general case will be examined, then the two special cases W_0' and $W_{\lambda_{2^d}(e)-1}'$ will be analyzed. The combined result will give the claimed formula.

For any window W_i, T_i denotes the tail of that specific window, hence, as $W_i = (e_{(i+1)\cdot d-1}, \dots, e_{id})$, $T_i := (e_{id-1}, \dots, e_0)$. The length of the tail of window W_i will be denoted as $t_i := \#T_i$.

case 1: the general case

Let W_i' with $1 \le i < \lambda_{2^d}(e) - 1$ be any inner window of the d-NAF. Then the three probabilities from formula (4.15) can be determined as follows:

case 1.1 : Prob($Val(W_i') = 0 \mid Val(W_i) = 0$)

The tail T_i of W_i may take any of the 2^{t_i} values $\{0, 1, \dots, 2^{t_i}-1\}$. From theorem 3.18, formula (3.12) on page 94, it is known that only the values up to (and including)

$$\hat{n}(t_i) \overset{(4.3)}{:=} \frac{1}{3} \cdot \left(2^{t_i+1} + \frac{(-1)^{t_i+1}-3}{2}\right)$$

may be displayed with their NAF using the same space, the other values between $\hat{n}(t_i)+1$ and $2^{t_i}-1$ require an additional digit, therefore, the NAF transformation of these tails propagates a carry bit into the window W_i. For the lower values, the transformation of T_i into its NAF has no effect on W_i, hence, when W_i' is created, it will also become a zero window. Note that zero has to be included, hence, this happens in

$$\begin{aligned}\frac{\hat{n}(t_i)+1}{2^{t_i}} &= \frac{\frac{1}{3}\cdot\left(2^{t_i+1}+\frac{(-1)^{t_i+1}-3}{2}\right)+1}{2^{t_i}} \\ &= \frac{2}{3}+\frac{(-1)^{t_i+1}+3}{3\cdot 2^{t_i+1}} \\ &\overset{t_i\to\infty}{\longrightarrow} \frac{2}{3} \end{aligned} \tag{4.16}$$

many of such cases, which gives the searched probability. Note that this probability indicates the fraction of all possible tails not propagating a carry bit.

case 1.2 : Prob($Val(W_i') = 0 \mid 1 \leq Val(W_i) \leq 2^d - 2$)

This is an easy case as a window W_i can never be changed into a zero window with any of these values, no matter if the tail T_i propagates a zero or not. It is $\text{Prob}(Val(W_i') = 0 \mid 1 \leq Val(W_i) \leq 2^d - 2) = 0$.

case 1.3 : Prob($Val(W_i') = 0 \mid Val(W_i) = 2^d - 1$)

In the case where W_i only consists of ones, a propagated 1 will change the window W_i to zero, hence, W_i' will be zero, too. Additionally, although the NAF-transformation of T_i might not require an additional digit, there is also a change to zero, if the leftmost digit of the NAF of T_i is one, giving a run of at least $d+1$ adjacent ones. That leftmost digit of the NAF of T_i becomes $\bar{1}$, while the run of ones in W_i is resolved at a zero left of W_i.

The first of the two described scenarios, where a carry bit is propagated, may arise in about $1 - \frac{2}{3} = \frac{1}{3}$ of all occurrences of $2^d - 1$ as a window value according to formula (4.16). The second described scenario can be computed if the number of all possible NAFs with exactly t_i digits (leading one required) is known. As any such NAF must start with 10 and may be followed by any NAF string of length $t_i - 2$, we have $s_2(t_i - 2)$ many such tails. As this is a fraction of

$$\begin{aligned}\frac{s_2(t_i-2)}{2^{t_i}} &\overset{(3.10)}{=} \frac{\frac{1}{3}\cdot\left(2^{t_i}+(-1)^{t_i-1}\right)}{2^{t_i}} \\ &= \frac{1}{3}+\frac{(-1)^{t_i-1}}{3\cdot 2^{t_i}} \overset{t_i\to\infty}{\longrightarrow} \frac{1}{3},\end{aligned}$$

we have

$$\text{Prob}(Val(W_i') = 0 \mid Val(W_i) = 2^d - 1) = \frac{1}{3}+\frac{1}{3}=\frac{2}{3} \tag{4.17}$$

of such windows W_i.

Altogether, as the probability of a single window W_i to be of any certain value is 2^{-d}, we have

$$\begin{aligned}\mathrm{Prob}(Val(W_i') = 0) &= \frac{2}{3 \cdot 2^d} + \frac{2}{3 \cdot 2^d} \\ &= \frac{4}{3 \cdot 2^d}.\end{aligned}$$

Therefore, as the fraction is a fraction of the windows in the Brauer partitioned binary expansion, this gives a total of

$$\frac{4}{3 \cdot 2^d} \cdot \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 2 \right)$$

many zero windows already.

case 2: the rightmost window

The rightmost window W_0' is zero iff W_i was, therefore, this case gives a total of

$$\frac{1}{2^d}$$

many additional zero windows.

case 3: the leftmost window

This last case is concerned with the leftmost window. It is usually not a source for a multiplication, because it initializes the accumulator. But in some seldom cases, the NAF has the right length to create an additional new leftmost window, which requires the former leftmost window to be handled as an ordinary window requiring a multiplication. This may only happen if $\lambda_2(e)$ is divided by the window width d and the NAF requires an additional digit. For this case, it is easier to determine the additional multiplications, which differs from cases 1 and 2, because here, mostly additional nonzero windows are created. Similarly to the average number of squarings in lemma 4.13, the following factor will be used to make sure that the derived costs only apply to those cases, where d divides $\lambda_2(e)$, e.g. the following formula evaluates to 1 if $\lambda_2(e)$ is a multiple of d and to 0 otherwise:

$$\left(1 - \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor \right) \overset{(4.14)}{=} \left(\left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - \left\lceil \frac{\lambda_2(e)}{d} \right\rceil \right)$$

As d divides $\lambda_2(e)$, the leftmost window consists of d digits, with the leftmost digit being $\mathtt{1}$. Therefore, the leftmost window has the 2^{d-1} possible window values $\{\mathtt{10}^{d-1}, \ldots, \mathtt{1}^d\}$. The number of those binary expansions with exactly $\lambda_2(e)$ digits, whose NAF requires an additional digit has been defined to be $s_2^{(-1)}(\lambda_2(e)+1)$, which evaluates to

$$\frac{2}{3} - \frac{(-1)^{\lambda_2(e)+1}+3}{3\cdot 2^{\lambda_2(e)}} \longrightarrow \frac{2}{3}$$

according to formula (3.19) on page 102. As the subtracted term diminishes rapidly, the limit is used within the formula.

However, not all of those window values, that create an additional leftmost window also cause an additional multiplication. In the case where $W_{\lambda_2(e)-1} = 2^d - 1$, the NAF transformation may either propagate a carry bit or leave the tail with a leftmost 1. The same case has already been studied in case 1.3 and the number of occurrences of this scenario is according to formula (4.17)

$$\frac{2}{3}\cdot\frac{1}{2^{d-1}} \quad = \quad \frac{4}{3\cdot 2^d},$$

giving an additional number of multiplications of

$$\frac{2}{3} - \frac{4}{3\cdot 2^d}.$$

Altogether, the claimed costs follow with the subtraction of the average number of zero windows found in case 1 and 2 from the overall number of windows in the binary expansion (minus 1, as there's still one multiplication saved by initialization of the accumulator) and by adding the correction term found in case 3. We have

$$\begin{aligned}
A &= \left(\left\lceil\frac{\lambda_2(e)}{d}\right\rceil - 1\right) - \frac{4}{3\cdot 2^d}\cdot\left(\left\lceil\frac{\lambda_2(e)}{d}\right\rceil - 2\right) - \frac{1}{2^d}\\
&\quad + \left(\frac{2}{3} - \frac{4}{3\cdot 2^d}\right)\cdot\left(\left\lceil\frac{\lambda_2(e)+1}{d}\right\rceil - \left\lceil\frac{\lambda_2(e)}{d}\right\rceil\right)\\
&= \frac{2^d - \frac{4}{3}}{2^d}\cdot\left\lceil\frac{\lambda_2(e)}{d}\right\rceil - \frac{2^d+\frac{5}{3}}{2^d}\\
&\quad + \left(\frac{2}{3} - \frac{4}{3\cdot 2^d}\right)\cdot\left(\left\lceil\frac{\lambda_2(e)+1}{d}\right\rceil - \left\lceil\frac{\lambda_2(e)}{d}\right\rceil\right)\\
&= \frac{2^d - \frac{4}{3}}{2^d}\cdot\left(\left\lceil\frac{\lambda_2(e)}{d}\right\rceil - 1\right) + \frac{1}{3\cdot 2^d} \qquad (4.18)\\
&\quad + \left(\frac{2}{3} - \frac{4}{3\cdot 2^d}\right)\cdot\left(\left\lceil\frac{\lambda_2(e)+1}{d}\right\rceil - \left\lceil\frac{\lambda_2(e)}{d}\right\rceil\right)
\end{aligned}$$

$$= \left\lceil \frac{\lambda_2(e)}{d} \right\rceil \cdot \left(\frac{2^d - \frac{4}{3}}{2^d} - \frac{2}{3} + \frac{4}{3 \cdot 2^d} \right) + \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil \cdot \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right)$$

$$+ \frac{1}{3 \cdot 2^d} - \frac{2^d - \frac{4}{3}}{2^d}$$

$$= \left\lceil \frac{\lambda_2(e)}{d} \right\rceil \cdot \frac{3 \cdot 2^d - 4 - 2 \cdot 2^d + 4}{3 \cdot 2^d} + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil$$

$$+ \frac{1 - 3 \cdot 2^d + 4}{3 \cdot 2^d}$$

$$= \frac{1}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - \frac{2^d - \frac{5}{3}}{2^d}$$

The average increase of the number of zero windows can be determined from above formula (4.18) by noting that the average fraction of nonzero windows in the Brauer partition of the binary expansion was (see table 2.7)

$$\frac{2^d - 1}{2^d} \cdot (\lambda_{2^d}(e) - 1)\,,$$

while the average fraction of nonzero windows in the d-NAF is about

$$\frac{2^d - \frac{4}{3}}{2^d} \cdot (\lambda_{2^d}(e) - 1)\,,$$

with the other terms left out, because their influence on the percentage diminishes fast compared to this one. This shows that the d-NAF has an expected 33% increase in zero windows. It also shows that the overall fraction of zero windows diminishes fast with increase in window length. □

Recall the result of the traditional Brauer method, which stated in table 2.7 the number of the multiplications in the main part to be (for $m = 2^d$) $\frac{2^d-1}{2^d} \cdot (\lambda_{2^d}(e) - 1)$. This means that the number of multiplications in the NAF-based Brauer method is slightly lower than with the binary expansion as a base for Brauer's approach. This result is not surprising, because although the digits of the NAF tend to be better distributed over the length of the chosen representation, the 33% weight reduction on average (see theorem 3.22) cannot be outweighed. On average, less multiplications and more zero windows result. Example 4.21(2) will show this effect.

Lemma 4.15 (exact and average number of inversions)
For the NAF-based Brauer method, the measure I indicating the number of inversions takes the following values if the input number e is i.i.d.:

inversions	
exact number	$I = 0$
worst case number	$I = 0$
average number	$I = 0$

Proof:

In the main loop, no inversions occur whatsoever. □

Theorem 4.16 (analysis of the NAF-based Brauer method)
The NAF-based Brauer method performed on a random input number $e \in \mathbb{N}$ requires the following costs depending on the chosen precomputation:

overview: the NAF-based Brauer cost analysis	
exact costs	
$I =$	$I_{\bar{\gamma}(\hat{n}(d)+1)}$
$Q(2) =$	$Q_{\bar{\gamma}(\hat{n}(d)+1)}(2) + d \cdot \left\lfloor \frac{\lambda_{\mathcal{NAF}}(e)}{d} \right\rfloor - d$
$A =$	$A_{\bar{\gamma}(\hat{n}(d)+1)} + \nu_{d-\mathcal{NAF}}(e) - 1$
worst case costs	
$I =$	$I_{\bar{\gamma}(\hat{n}(d)+1)}$
$Q(2) =$	$Q_{\bar{\gamma}(\hat{n}(d)+1)}(2) + d \cdot \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor$
$A =$	$A_{\bar{\gamma}(\hat{n}(d)+1)} + \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor$
average costs *(for d large enough)*	
$I =$	$I_{\bar{\gamma}(\hat{n}(d)+1)}$
$Q(2) =$	$Q_{\bar{\gamma}(\hat{n}(d)+1)}(2) + \frac{d}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \frac{2d}{3} \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - d$
$A =$	$A_{\bar{\gamma}(\hat{n}(d)+1)} + \frac{1}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d}\right) \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - \frac{2^d - \frac{5}{3}}{2^d}$

For the precomputation, there exist addition chains

$$\bar{\gamma}_1, \bar{\gamma}_2, \bar{\gamma}_3, \bar{\gamma}_4 \in \bar{\Gamma}(\hat{n}(d) + 1),$$

such that the following costs for the precomputation may be used in the formulas of the above overview. Note that $\bar{\gamma}_1$ requires the minimal number of *inversions, while $\bar{\gamma}_2$ requires the maximal number of inversions, $\bar{\gamma}_3$ and $\bar{\gamma}_4$ replace the maximal number of squarings or additions respectively by inversions.*

overview: important possible values for the precomputation			
	$\boldsymbol{I}_{\bar{\gamma}(\hat{n}(d)+1)}$	$\boldsymbol{Q}_{\bar{\gamma}(\hat{n}(d)+1)}$	$\boldsymbol{A}_{\bar{\gamma}(\hat{n}(d)+1)}$
$\bar{\gamma}(\hat{n}(d)+1) = \bar{\gamma}_1$	1	$s_2(d-1)-1$	$2 \cdot s_2(d-2)-2$
$\bar{\gamma}(\hat{n}(d)+1) = \bar{\gamma}_2$	$\frac{1}{2} \cdot s_2(d) - \frac{1}{2}$	$\frac{1}{2} \cdot s_2(d-1) - \frac{1}{2}$	$s_2(d-2)-1$
$\bar{\gamma}(\hat{n}(d)+1) = \bar{\gamma}_3$	$\frac{1}{2} + \frac{1}{2} \cdot s_2(d-1)$	$\frac{1}{2} \cdot s_2(d-1) - \frac{1}{2}$	$2 \cdot s_2(d-2)-2$
$\bar{\gamma}(\hat{n}(d)+1) = \bar{\gamma}_4$	$s_2(d-2)$	$s_2(d-1)-1$	$s_2(d-2)-1$

Proof:

The results have been shown in lemma 4.10, lemma 4.13, lemma 4.14 and lemma 4.15.

By concatenating the addition chains for the precomputation step and for the main step, no hidden doublings are created. While the precomputation step computes window values, no hidden doublings occur as it has been shown in lemma 4.10. After that, in the main exponentiation step, the exponentiation with 2^d is assumed to be performed by an optimal addition chain using only doublings, hence, only line 5 could miscount hidden doublings. Assume $x \notin \{0,1\}$ and $d > 1$ ($d = 1$ has been proven in the NAF-based binary method's analysis). The multiplication in this line never adds the same element to itself, if the accumulator is different from any possible window value. This can be shown easily: First, the accumulator is initialized with the leftmost window value, which is nonzero, e.g. $A = x^i$ with $i \geq 1$. After that, in the first execution of the main loop, A is squared d times, giving a value of $A = x^j$ with $j \geq 2^d$. But in line 5, the possible window values are powers x^k of x with $-2^d + 1 \leq k \leq 2^d - 1$. Hence, in the first execution of line 5, no hidden doubling occurs. Assuming the lower bound for the exponent, x^{-2^d+1}, for the window value, it still leaves $A = x^m$ with $m \geq 2^d - 2^d + 1 = 1$. But as A is squared d times again after that multiplication in the following execution of the loop, $A = x^n$ with $n \geq 2^d$ again before executing the next multiplication of line 5. Thus, the same situation as before occurs, which therefore proves the claim inductively. □

4.2.3 Experimental results

Tables 4.5 and 4.6 show the results of some experiments compared to the results of the theorems in the last section. The experimental data has been created using 10^5 sets of random binary expansions with a certain length, where the leftmost digit has been set to 1 in order to assure that all inputs have the same length. The binary expansion has been transformed into the d-NAF and the numbers of zero and nonzero windows have been counted. The hardware was a 733MHz PC with Intel architecture, the random bits have been generated with the C++ rand() function as

$$e_i := (\mathsf{int})(2.0 * ((\mathsf{double})\mathsf{rand}()/(\mathsf{double})(\mathsf{RAND\_MAX} + 1))).$$

$(\lambda_2(e), d)$	**experimental $Q(2)$ main part only**	**$Q(2)$ from lemma 4.13 main part only**
(20, 1)	19.6698	19.6666
(67, 1)	66.6673	66.6666
(128, 2)	127.3290	127.3333
(155, 2)	154.0000	154.0000
(192, 2)	191.3360	191.3333
(256, 3)	255.0000	255.0000
(307, 3)	306.0000	306.0000
(419, 3)	417.0000	417.0000
(512, 3)	510.0000	510.0000
(555, 3)	554.0060	554.0000
(1024, 4)	1022.6700	1022.6666
(2048, 5)	2045.0000	2045.0000
(4096, 5)	4096.0000	4095.0000
(8192, 6)	8190.0000	8190.0000

Table 4.5: Experimental and theoretical results for $Q(2)$

The experimental data follows the already proved formulas.
The results of the analysis in theorem 4.16 can now be used to determine if and when the NAF-based Brauer method is better than the traditional approach. This leads to a cost comparison suitable for every application in the next section. There is no general result about which variant of the Brauer method is superior. The reason for this fact can be seen in looking at the analysis of the precomputation part. If inversions are expensive, the traditional Brauer method is determined to be the method of choice. If inversions are cheap, the savings in the precomputation step take full effect. At first sight, there are more operations altogether, e.g. about $\frac{4}{3}$ the number of the traditional Brauer method as the total number of arithmetical

$(\lambda_2(e), d)$	experimental A main part only	A from lemma 4.14 main part only
(20, 1)	6.446	6.500
(67, 1)	22.120	22.166
(128, 2)	42.448	42.416
(155, 2)	51.444	51.416
(192, 2)	63.779	63.750
(256, 3)	70.855	70.875
(307, 3)	85.031	85.041
(419, 3)	115.839	115.875
(512, 3)	141.684	141.708
(555, 3)	153.887	153.875
(1024, 4)	234.373	234.354
(2048, 5)	391.970	391.968
(4096, 5)	784.865	784.885
(8192, 6)	1336.55	1336.567

Table 4.6: Experimental and theoretical results for A

operations found in lemma 4.10 is for $\bar{\gamma}(\hat{n}(d)+1) \in \{ \bar{\gamma}_1(\hat{n}(d)+1), \bar{\gamma}_2(\hat{n}(d)+1) \}$

$$\begin{aligned} I_{\bar{\gamma}(\hat{n}(d)+1)} + Q_{\bar{\gamma}(\hat{n}(d)+1)}(2) + A_{\bar{\gamma}(\hat{n}(d)+1)} &= \frac{1}{3} \cdot \left(2^{d+2} + (-1)^{d+1}\right) - 2 \\ &\approx \frac{4}{3} \cdot 2^d, \end{aligned}$$

while the total number of operations of the traditional Brauer method, according to lemma 2.11, sums up to

$$Q_{\gamma(m)}(2) + A_{\gamma(m)} = 2^d - 2,$$

for $\gamma(m) \in \{\gamma_1(m), \gamma_2(m)\}$ of lemma 2.11.
But if inversions are very cheap, the costs will be lowered as half of the additions and half of the squarings can be replaced using inversions. If inversions are completely free, there are only about $\frac{2}{3}$ of the operations of the traditional precomputation, hence, a substantial saving.
The savings in the main part are not that impressive. In the contrary, while there are slightly less multiplications, the number of squarings is slightly increased. The ratio between the costs of both Brauer variants in the main part is closer to 1, although, as the savings in multiplications scale with the input length, while the slight increase in the squarings only scales with the window length, the overall costs in the main part are reduced. The next section will give examples to illustrate this behaviour.

4.3 Cost comparisons

It has been mentioned many times within this thesis that different applications involve very different costs. Hence, the decision whether to use a NAF-based method or a traditional method must depend on the actual implementational costs of the application. It is assumed that the choice about the method itself has been already made – the binary method should only be considered if the implementation needs to be simple or there is only a very limited amount of memory, otherwise the Brauer method should be used. This is because the Brauer method can adapt to the input size by choosing an optimal window length. The following two sections can then give an answer to the question which number representation is most suitable. The theorems presented can also be used to build hybrid algorithms, that can decide (on average or for every input), which variant of the chosen method to apply. In doing this, one can combine the advantages of both variants.

4.3.1 Cost comparison for the binary method

In the following, two theorems are presented. The first gives the cost comparison in the case, the input can be examined and knowledge about length and configuration (especially Hamming weight) can be used for the comparison. The costs of the underlying operations, squaring, multiplication and inversion, must be provided in any case.

Theorem 4.17 (exact cost comparison for the binary method)
Let $\Pi = (x \in G, e \in \mathbb{N})$ be a given exponentiation problem with random but fixed values. Assume that $\lambda_2(e), \lambda_{\mathcal{NAF}}(e), \nu_2(e)$ and $\nu_{\mathcal{NAF}}(e)$ are known, as well as $c(I), c(Q(2))$ and $c(A)$, the costs of the arithmetical operations for the computation of x^e.
Then the following inequality can be used to decide whether the traditional binary method or the NAF-based binary method is faster to compute x^e.
If

$$R := c(I) + c(Q(2)) \cdot (\lambda_{\mathcal{NAF}}(e) - \lambda_2(e)) + c(A) \cdot (\nu_{\mathcal{NAF}}(e) - \nu_2(e)) > 0$$

then the ordinary binary method is faster,
otherwise the NAF-based binary method is faster.

Proof:

The results of the analyses of both variants of the binary method are subtracted. The results are stated in table 2.2 and in theorem 4.6. □

If the method should not be chosen depending on the actual input, but based on general considerations, the cost comparison for the average case gives the answer. Here, the Hamming weight of the input is no longer known. The binary length has to be known, hence the average application case should be known (for example the RSA system, where a 512 bit integer is expected).

Theorem 4.18 (average cost comparison for the binary method) *Let $n := \lambda_2(e)$, $c(I)$, $c(Q(2))$ and $c(A)$ be declared as in theorem 4.17.*

Then for n large enough, the following inequality can be used to decide whether to use the traditional binary method or the NAF-based binary method in order to compute x^e.

If

$$R := c(I) + c(Q(2)) \cdot \frac{2}{3} - c(A) \cdot \frac{3n+5}{18} > 0 \tag{4.19}$$

then the traditional binary method is faster,
otherwise the NAF-based binary method should be preferred.

Proof:

The results of the analyses of both variants of the binary method are subtracted. The total average costs of the traditional binary method are (see table 2.2)

$$c(Q(2)) \cdot (\lambda_2(e) - 1) + c(A) \cdot \frac{1}{2} \cdot (\lambda_2(e) - 1), \tag{4.20}$$

the costs of the NAF-based binary method are (see theorem 4.6)

$$c(I) + c(Q(2)) \cdot \left(\lambda_2(e) - \frac{1}{3}\right) + c(A) \cdot \left(\frac{1}{3} \cdot \lambda_2(e) - \frac{7}{9}\right). \tag{4.21}$$

Subtracting (4.20) from (4.21) leads to the stated claim:

$$\begin{aligned}
& c(I) + c(Q(2)) \cdot (\lambda_2(e) - \frac{1}{3} - \lambda_2(e) + 1) \\
& + c(A) \cdot (\frac{1}{3} \cdot \lambda_2(e) - \frac{7}{9} - \frac{\lambda_2(e)}{2} + \frac{1}{2}) \\
= \; & c(I) + c(Q(2)) \cdot \frac{2}{3} - c(A) \cdot \left(\frac{\lambda_2(e)}{6} + \frac{5}{18}\right)
\end{aligned}$$

□

Example:

(1) Recall the stated implementation of the RSA cryptosystem (example 2.4(1)), with costs of

$$c(A) = c(Q(2)) = 1$$
$$c(I) = log(\lambda_2(N)).$$

Assume, as in practical applications, that the modulo N has 512 bits, e.g. $\lambda_2(N) = 512$. Then the average binary cost comparison leads to

$$R := -\frac{1}{6}n + 9 + \frac{7}{18} = -\frac{n}{6} + \frac{169}{18},$$

which is positive for all $n \in \{1, 2, \ldots, 56\}$. Hence, in practical applications, if the private or public key has more than 56 digits, the NAF-based binary method is faster, otherwise the traditional binary method is a good choice.

(2) For the same application as in part (1), consider as input the standard example of chapter 1

$$\begin{array}{rclll} (219)_2 & = & 11011011 & \nu_2(219) = 6 & \lambda_2(219) = 8 \\ (219)_{\mathcal{NAF}} & = & 100\bar{1}00\bar{1}0\bar{1} & \nu_{\mathcal{NAF}}(219) = 4 & \lambda_{\mathcal{NAF}}(219) = 9 \end{array}$$

and suppose the exact cost comparison should be used. It is

$$\begin{aligned} R &= c(I) + c(Q(2)) \cdot (\lambda_{\mathcal{NAF}}(e) - \lambda_2(e)) + c(A) \cdot (\nu_{\mathcal{NAF}}(e) - \nu_2(e)) \\ &= \log \lambda_2(N) + 1 - 2 = \log \lambda_2(N) - 1 \end{aligned}$$

Therefore, for this input, the traditional method excels the NAF-based binary method for any modulo $N > 3$.

Note that in the cost comparison for the average case, only the costs of the addition scale with the length of the input. As this term is negative, the NAF-based binary method will win in almost every situation.

4.3.2 Cost comparison for the Brauer method

For the Brauer method, the results from section 4.2 also allow to state two inequalities, that can be used to determine which variation of the method should be preferred. The following two theorems are, as in the last section, cost comparisons for the exact case, when the input is known and information about its pattern can be used, and for the average case, where not every single input should be investigated, but the aim is to choose the best method on average.

Theorem 4.19 (exact cost comparison for the Brauer method)
Let $\Pi = (x \in G, e \in \mathbb{N})$ be a given exponentiation problem with random but fixed values. Assume that $\lambda_2(e), \lambda_{\mathcal{NAF}}(e), \nu_{2^d}(e)$ and $\nu_{d-\mathcal{NAF}}(e)$ are known, as well as $c(I), c(Q(2))$ and $c(A)$, the costs of the arithmetical operations for the computation of x^e.

Then the following inequality can be used to decide whether the traditional Brauer method or the NAF-based Brauer method is faster to compute x^e. Because the Brauer method offers different approaches for the precomputation, the cost comparison consists of three steps, determining the best precomputation for both methods and then subtraction to get the cost comparison inequality:

step 1:

For the precomputation of the traditional Brauer method, define

$$c_1 \quad := \quad \begin{cases} c(Q(2)) + c(A) \cdot \left(2^d - 3\right) & \text{if } c(A) \leq c(Q(2)) \\ (c(Q(2)) + c(A)) \cdot \left(2^{d-1} - 1\right) & \text{otherwise,} \end{cases}$$

hence, c_1 denotes the minimal costs of any addition chain γ performing the precomputation of the traditional Brauer method.

step 2:

For the NAF-based Brauer method, define

$$c_{2a} \quad := \quad \begin{cases} (c(I) + c(A)) \cdot (s_2(d-2) - 1) & \text{if } c(I) \leq c(A) \\ 2 \cdot c(A) \cdot (s_2(d-2) - 1) & \text{otherwise} \end{cases}$$

and

$$c_{2b} \quad := \quad \begin{cases} (c(I) + c(Q(2))) \cdot \left(\frac{1}{2} \cdot s_2(d-1) - \frac{1}{2}\right) & \text{if } c(I) \leq c(Q(2)) \\ c(Q(2)) \cdot (s_2(d-1) - 1) & \text{otherwise,} \end{cases}$$

hence, c_{2a} denotes the minimal costs of the construction of all elements, that could be constructed by squarings and c_{2b} denotes the minimal costs of the construction of all elements, that could be constructed by additions. Define c_2 as

$$c_2 \quad : \quad = c(I) + c_{2a} + c_{2b},$$

then c_2 denotes the minimal costs of any addition-subtraction chain $\bar{\gamma}$ performing the precomputation of the NAF-based Brauer method.

step 3:
With the above costs c_1 and c_2 known, the exact cost comparison inequality follows as:

$$\begin{aligned} R &:= c(Q(2)) \cdot d \cdot \left(\left\lceil \frac{\lambda_{\mathcal{NAF}}(e)}{d} \right\rceil - \left\lceil \frac{\lambda_2(e)}{d} \right\rceil \right) \\ &\quad + c(A) \cdot (\nu_{d-\mathcal{NAF}}(e) - \nu_{2^d}(e)) \\ &\quad + c_2 - c_1 \end{aligned}$$

If $R > 0$, then the ordinary Brauer method is faster,
otherwise the NAF-based Brauer method is faster.

Proof:

step 1

In step 1, the two addition chains for precomputation for the traditional Brauer method introduced in formulas (2.1) and (2.2) account for the operational costs. The costs have been shown in lemma 2.11. They represent the two possible approaches for maximizing squarings or multiplications.

step 2

In step 2, the results from lemma 4.10 are applied. The lemma shows that while there is always at least one mandatory inversion, half of the squarings and half of the multiplications can be replaced by inversions. As there are $s_2(d-1)-1$ squarings for $d > 1$ and $2 \cdot (s_2(d-2)-1)$ multiplications, half of both numbers can be counted as inversions, if inversion is cheaper. Step 2 performs that replacement if applicable. As the steps creating the elements can be mixed from both described versions, there are addition chains with the claimed costs.

step 3

If the costs of the precomputations have been chosen optimal according to the first two steps, they can be inserted for the total costs which have been found in the analyses of the two variations of the Brauer method. In corollary 2.15, the exact number of operations for the traditional method has been shown to be

$$\begin{aligned} Q(2) &= \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1 \right) \cdot d + Q_{\gamma(2^d)}(2) \\ A &= (\nu_{2^d}(e) - 1) + A_{\gamma(2^d)}, \end{aligned}$$

for any addition chain $\gamma(2^d) \in \Gamma(2^d)$. If now the optimal addition chain for the precomputation is chosen from $\Gamma(2^d)$, and operations are added with their individual costs, this result is

$$\left(\left\lceil\frac{\lambda_2(e)}{d}\right\rceil - 1\right)\cdot d\cdot c(Q(2)) + (\nu_{2^d}(e) - 1)\cdot c(A) + c_1.$$

For the NAF-based Brauer method, the exact results without the precomputation can be seen in theorem 4.16, where the following costs are stated (operations with their costs and those of the precomputation already summed up):

$$\left(d\cdot\left\lceil\frac{\lambda_{\mathcal{NAF}}(e)}{d}\right\rceil - d\right)\cdot c(Q(2)) + (\nu_{d-\mathcal{NAF}}(e) - 1)\cdot c(A) + c_2$$

Subtracting the traditional Brauer version from the NAF-based Brauer version leads to the claimed cost comparison inequality:

$$\begin{aligned}
& c(Q(2))\cdot\left(d\cdot\left\lceil\frac{\lambda_{\mathcal{NAF}}(e)}{d}\right\rceil - d - \left(\left\lceil\frac{\lambda_2(e)}{d}\right\rceil - 1\right)\cdot d\right)\\
& +c(A)\cdot(\nu_{d-\mathcal{NAF}}(e) - 1 - \nu_{2^d}(e) + 1) + c_2 - c_1\\
= \quad & c(Q(2))\cdot d\cdot\left(\left\lceil\frac{\lambda_{\mathcal{NAF}}(e)}{d}\right\rceil - \left\lceil\frac{\lambda_2(e)}{d}\right\rceil\right)\\
& +c(A)\cdot(\nu_{d-\mathcal{NAF}}(e) - \nu_{2^d}(e)) + c_2 - c_1.
\end{aligned}$$

□

For the Brauer method it is also possible to offer a cost comparison for the average case, just like for the binary method. If not every input should be examined, but the algorithm of choice should be better on average, the following inequality helps to find the best choice for a given application.

Theorem 4.20 (average cost comparison for the Brauer method) *Let $\lambda_2(e)$, $c(I), c(Q(2))$ and $c(A)$ be known and declared as in theorem 4.17. Then for $\lambda_2(e)$ large enough, the following inequality can be used to decide whether to use the traditional binary method or the NAF-based binary method in order to compute x^e.*

step 1:
Perform step 1 of theorem 4.19 to get c_1.

step 2:
Perform step 2 of theorem 4.19 to get c_2.

step 3:
With c_1 and c_2 known, set

$$\begin{aligned}
R \quad := \quad & c(A)\cdot\left(\frac{3-2^{d+1}}{3\cdot 2^d}\cdot\left\lceil\frac{\lambda_2(e)}{d}\right\rceil + \left(\frac{2}{3} - \frac{4}{3\cdot 2^d}\right)\cdot\left\lceil\frac{\lambda_2(e)+1}{d}\right\rceil + \frac{2}{3\cdot 2^d}\right)\\
& + c(Q(2))\cdot\frac{2d}{3}\cdot\left(\left\lceil\frac{\lambda_2(e)+1}{d}\right\rceil - \left\lceil\frac{\lambda_2(e)}{d}\right\rceil\right) + \ c_2 - c_1
\end{aligned}$$

If $R > 0$, then the ordinary Brauer method is faster, otherwise the NAF-based Brauer method is faster.

Proof:

step 1 and step 2

The precomputation step is independent from the configuration of the input, the same operations are carried out for every input. Therefore, the same results as in theorem 4.19 apply. This can also be seen in lemma 2.11 and lemma 4.10.

step 3

For step 3, recall the average case analysis of the traditional Brauer method depicted in corollary 2.15, where the following costs have been derived:

$$\begin{aligned} & c(Q(2)) \cdot \left(\left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1 \right) \cdot d \right) + c(Q_{\gamma(2^d)}(2)) \\ & + c(A) \cdot \left(\frac{2^d - 1}{2^d} \cdot \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1 \right) \right) + c(A_{\gamma(2^d)}) \end{aligned} \tag{4.22}$$

for any addition chain $\gamma(2^d) \in \Gamma(2^d)$, while the lemmas 4.13, 4.14 and 4.15 show the average case analysis of the NAF-based Brauer method to be:

$$\begin{aligned} & c(Q(2)) \cdot \left(\frac{d}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \frac{2d}{3} \cdot \left\lceil \frac{\lambda_2(e) + 1}{d} \right\rceil - d \right) \\ & + \quad c(A) \cdot \left(\frac{1}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \left\lceil \frac{\lambda_2(e) + 1}{d} \right\rceil - \frac{2^d - \frac{5}{3}}{2^d} \right) \\ & + c(\bar{\gamma}(\hat{n}(d) + 1), \end{aligned} \tag{4.23}$$

for any addition-subtraction chain $\bar{\gamma}(\hat{n}(d) + 1) \in \bar{\Gamma}(\hat{n}(d) + 1)$. If the optimal chains for the precomputation are inserted, with costs known from parts 1 and 2, subtracting formula (4.22) from (4.23) gives the claimed result:

$$\begin{aligned} & c(Q(2)) \cdot \left(\frac{d}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \frac{2d}{3} \cdot \left\lceil \frac{\lambda_2(e) + 1}{d} \right\rceil - d - \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1 \right) \cdot d \right) \\ & + c(A) \cdot \left(\frac{1}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \left\lceil \frac{\lambda_2(e) + 1}{d} \right\rceil - \frac{2^d - \frac{5}{3}}{2^d} \right. \\ & \left. - \frac{2^d - 1}{2^d} \cdot \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1 \right) \right) + c_2 - c_1 \end{aligned}$$

$$
\begin{aligned}
&= c(Q(2)) \cdot \left(\frac{d-3d}{3} \cdot \left\lceil\frac{\lambda_2(e)}{d}\right\rceil + \frac{2d}{3} \cdot \left\lceil\frac{\lambda_2(e)+1}{d}\right\rceil\right) \\
&\quad + c(A) \cdot \left(\frac{2^d - 3 \cdot 2^d + 3}{3 \cdot 2^d} \cdot \left\lceil\frac{\lambda_2(e)}{d}\right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d}\right) \cdot \left\lceil\frac{\lambda_2(e)+1}{d}\right\rceil\right. \\
&\quad \left. + \frac{-2^d + \frac{5}{3} + 2^d - 1}{2^d}\right) + c_2 - c_1 \\
&= c(Q(2)) \cdot \frac{2d}{3} \cdot \left(\left\lceil\frac{\lambda_2(e)+1}{d}\right\rceil - \left\lceil\frac{\lambda_2(e)}{d}\right\rceil\right) \\
&\quad + c(A) \cdot \left(\frac{3 - 2^{d+1}}{3 \cdot 2^d} \cdot \left\lceil\frac{\lambda_2(e)}{d}\right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d}\right) \cdot \left\lceil\frac{\lambda_2(e)+1}{d}\right\rceil\right. \\
&\quad \left. + \frac{2}{3 \cdot 2^d}\right) + c_2 - c_1
\end{aligned}
$$

□

The cost comparison inequalities for the Brauer method can be used to find the optimal variant of this method. The following examples give exemplary results, which show that the costs of the operation is a crucial point for the choice of the right method.

Example 4.21:

(1) Recall example 2.4(1) with costs of

$$c(A) = c(Q(2)) = 1 \qquad \text{and} \qquad c(I) = \log(\lambda_2(N)).$$

for exponentiation in $\mathbb{Z}_N$. Let N consist of 512 binary digits, as it is usual for the RSA cryptosystem. Assume that the Brauer method should be based on the number representation that yields the best results on average. Hence, the average cost comparison inequality should be used. For the window width d, the optimal value of

$$d \;=\; \text{round}\left(\frac{2}{\ln 2} \cdot W\left(\frac{1}{2}\sqrt{\lambda_2(e) \cdot \ln 2}\right)\right)$$

according to formula (2.5) (with $c = 1$ because $c(A) = c(Q(2))$) should be chosen. Then the three steps of the cost comparison result to the following:
As $c(A) = c(Q(2))$, in step 1 the first version is chosen, leading to costs of

$$c_1 \;=\; 2^d - 2$$

In step 2, the cost of one inversion is $c(I) = 9$ and hence, more expensive than a squaring or an addition. Step 2 results in:

$$\begin{aligned} c_2 &= 9 + 2 \cdot (s_2(d-2) - 1) + s_2(d-1) - 1 \\ &\overset{(3.8)}{=} s_2(d) + 6 = \frac{1}{3} \cdot \left(2^{d+2} + (-1)^{d+1}\right) + 6. \end{aligned}$$

Altogether, this leads in step 3 to the following solution for R:

$$\begin{aligned} R &= \frac{3 - 2^{d+1}}{3 \cdot 2^d} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d}\right) \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil \\ &\quad + \frac{2}{3 \cdot 2^d} + \frac{2d}{3} \cdot \left(\left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - \left\lceil \frac{\lambda_2(e)}{d} \right\rceil \right) + c_2 - c_1 \\ &= \frac{3 - 2^{d+1} - d \cdot 2^{d+1}}{3 \cdot 2^d} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil \\ &\quad + \frac{2^{d+1} - 4 + d \cdot 2^{d+1}}{3 \cdot 2^d} \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil \\ &\quad + \frac{2}{3 \cdot 2^d} + c_2 - c_1 \\ &= \frac{3 - 2^{d+1} \cdot (d+1)}{3 \cdot 2^d} \cdot \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil \right) \\ &\quad - \frac{1}{3 \cdot 2^d} \cdot \left(\left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - 2 \right) + c_2 - c_1 \end{aligned}$$

which is greater than 0 for all values of $\lambda_2(e)$, as it can be seen in figure 4.3, hence, for this ratio of the three basic operations, the NAF-based Brauer method is inferior to the traditional method and should not be used. Note that figure 4.3 depicts only one function, R, the parallel lines are in fact one function, the points alternate quickly between the two levels because of the use of the Gaussian brackets, which lead to a jump in the number of squarings each time $\lambda_2(e)$ is divided by the window width d.

The three figures 4.4, 4.5 and 4.6 show the performance of the two variants of the Brauer method for the ratio $c(A) = c(Q(2)) = 1$ chosen for this example. The blue line shows the lowest costs of the traditional Brauer method for d chosen from table 2.10. The red and the green line show the costs of the traditional Brauer method for the two surrounding values for d. The graphs are lines, because no inversions occur for the traditional method. The dashed lines depict the graphs of the corresponding NAF-based Brauer methods. It can be seen that only for small values of $c(I)$, e.g.

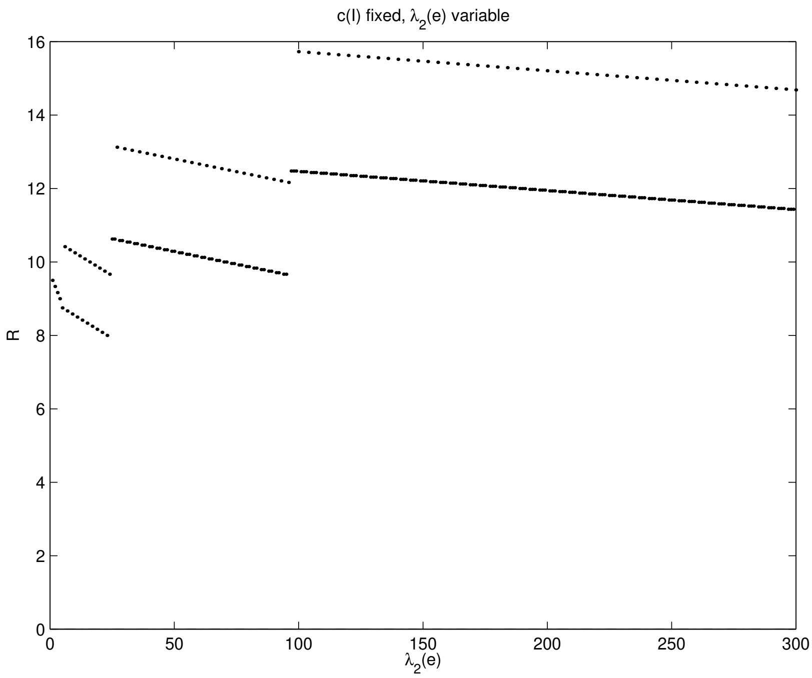

Figure 4.3: Graph of the function R for example 4.21(1)

$c(I) < c(A) = c(Q(2))$, the NAF-based Brauer method is faster than the traditional variant. The break at $c(I) = 1$ results from the change of the precomputation, which always prefers the cheaper operation, hence, right of $c(I) = 1$, only a single inversion occurs using the NAF-based Brauer method. The graphs are depicted for $\lambda_2(e) \in \{192, 512, 1024\}$. The graph for $\lambda_2(e) = 1024$ shows that for large values of $\lambda_2(e)$, the savings in multiplications in the main part of the algorithm (as explained in theorem 4.14) can be seen, because here, the NAF-based Brauer method is superior also for some values of $c(I) > c(A) = c(Q(2))$.

(2) Now recall example 2.4(3) with costs of

$$\begin{aligned} c(A) &= 100 \\ c(Q(2)) &= 60 \\ c(I) &= 0 \end{aligned}$$

for the implementation of an elliptic curve cryptosystem using mixed coordinates. Let the window width d be chosen as in example (1). Then, as squarings are faster than additions, the precomputation of the traditional Brauer method gives the following results in step 1:

$$c_1 = 160 \cdot \left(2^{d-1} - 1\right)$$

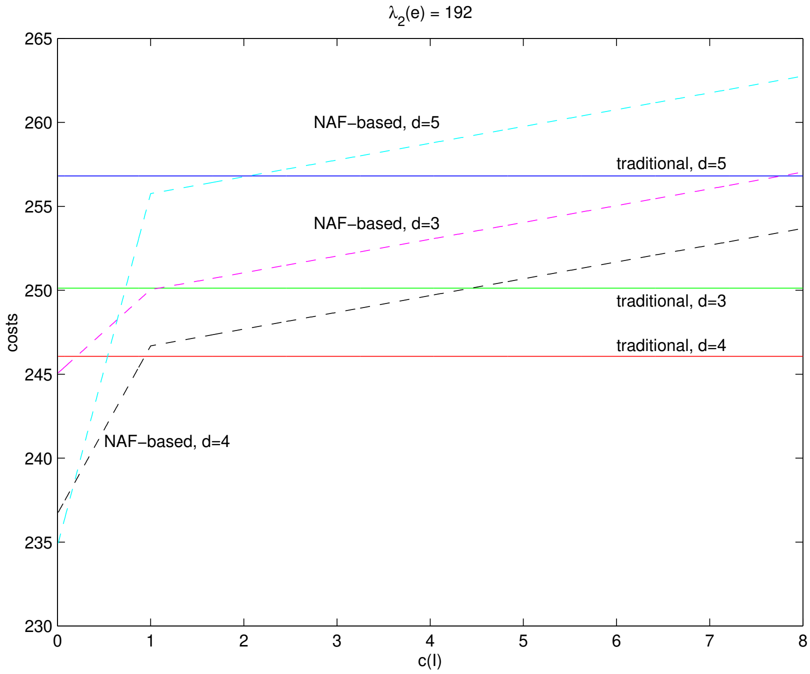

Figure 4.4: Costs of Brauer method variants for $\lambda_2(e) = 192$

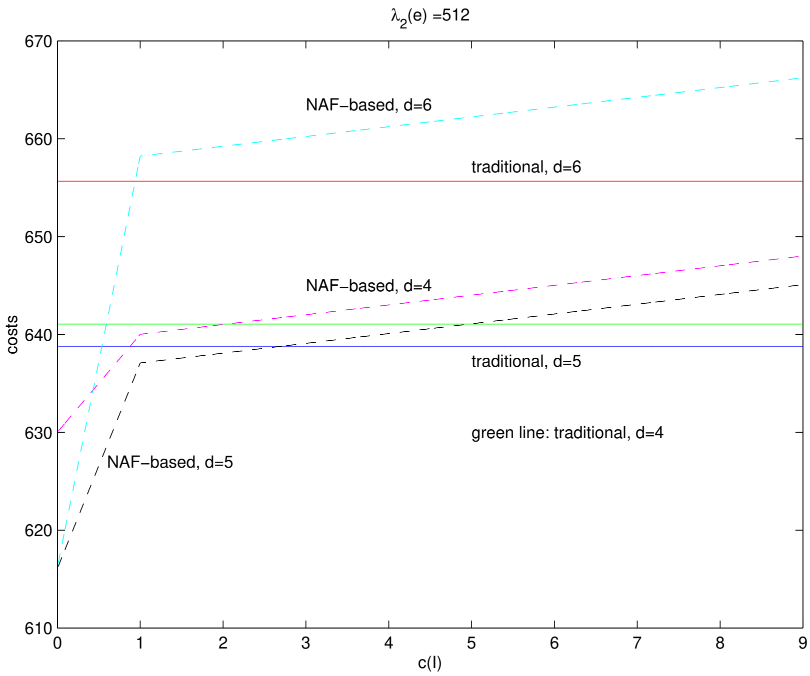

Figure 4.5: Costs of Brauer method variants for $\lambda_2(e) = 512$

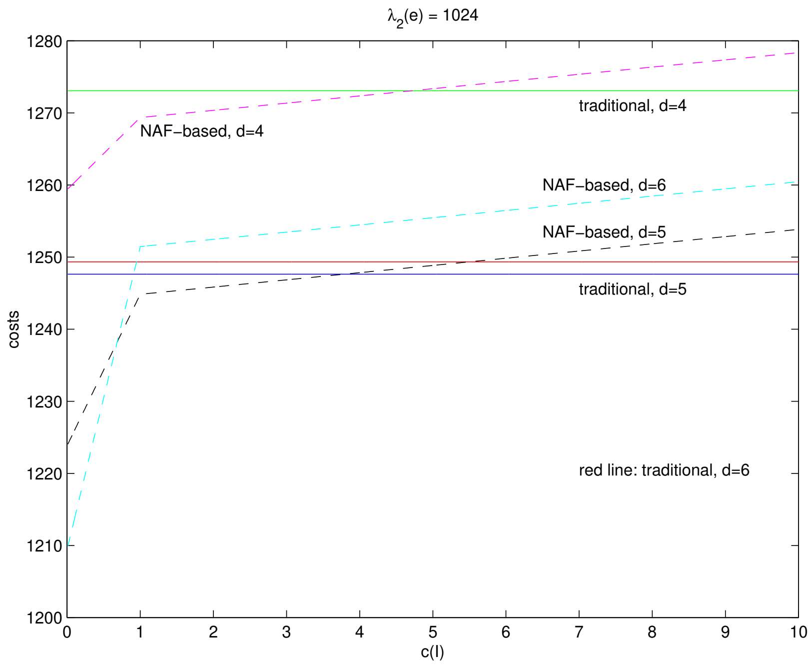

Figure 4.6: Costs of Brauer method variants for $\lambda_2(e) = 1024$

In step 2, as inversions are free of costs, the maximum number of inversions is chosen resulting in

$$c_2 \quad = \quad 100 \cdot (s_2(d-2) - 1) + 30 \cdot (s_2(d-1) - 1)\,.$$

And step 3 gives the average cost comparison inequality as

$$\begin{aligned} R \quad = \quad & 100 \cdot \left(\frac{3 - 2^{d+1}}{3 \cdot 2^d} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil \right) \\ & + \frac{200}{3 \cdot 2^d} + 40d \cdot \left(\left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - \left\lceil \frac{\lambda_2(e)}{d} \right\rceil \right) \\ & + 100 \cdot (s_2(d-2) - 1) + \; 30 \cdot (s_2(d-1) - 1) \\ & - \; 160 \cdot \left(2^{d-1} - 1 \right) \end{aligned}$$

The graph of this equation is depicted in figure 4.7. It can be seen, that for this example, the NAF-based Brauer method excels the traditional method even for very small inputs lengths. For today's choices of key length in elliptic curve cryptosystems, like $\lambda_2(e) \in \{128, 192, 256\}$, the new method is definitely faster than the traditional Brauer method.

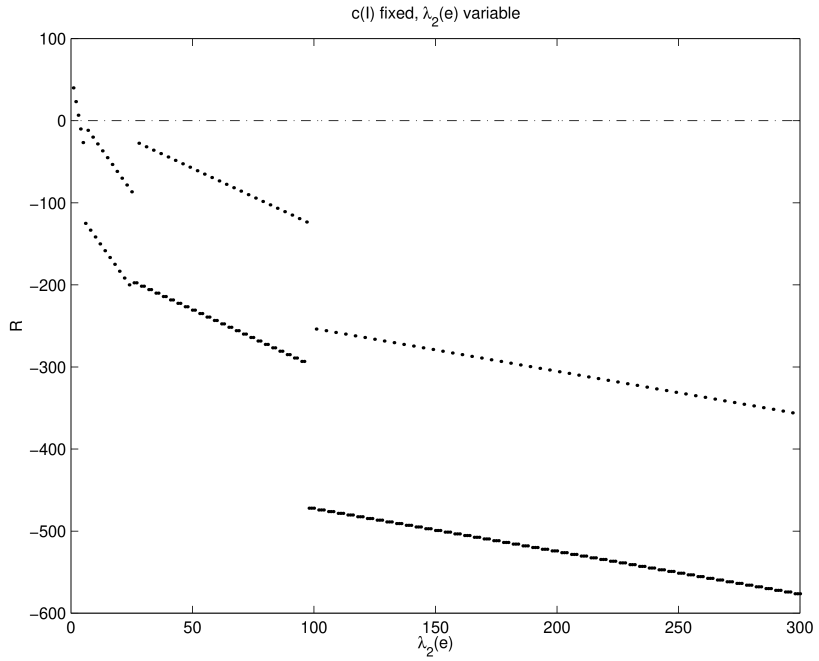

Figure 4.7: Graph of the function R for example 4.21(2)

Note that the labeling of the axes in both presented graphs of R is somewhat misleading. One might be tempted to judge by the values on the y-axis, that in the first figure, the traditional method outweighs the new method only slightly while in the second figure the savings are substantial. Of course savings increase with input length, but the labeling only reflects the chosen magnitudes of the cost measures, hence cost measures of 60 and 100 as from example 2 will suggest greater superiority than the cost measures 1, 1, 9 in the first example. The pictures can only show which method is superior and how the ratio develops.

The examples once again show that not only the choice of the method, but also the choice of the implementation and the environment of the exponentiation problem, e.g. the costs of the basic operations, determine which representation of the exponent e and, hence, which variant of the method is superior.

4.4 A generalized average case analysis

For all results about the average or expected number of operations, the definition of the average case from remark 2.5 has been used, stating results about trials from the set Ω_n of all inputs of the same binary length. However, in practical applications like cryptosystems, the average input is not always of a certain length, but of a certain maximal length, requiring sometimes to examine results according to a more general definition of the average case. This generalized average case will be defined in the following definition and the task of this section will be to expand the known results about the arithmetical costs of both variants of the Brauer method as well as the cost comparison of the average results according to that generalized average case.

Definition 4.22 (generalized average case for variable length)
For the generalized average case, the average is taken as the average of all inputs $e \in \Omega(n) := \{0, 1, \ldots, 2^n - 1\}$, where the inputs are assumed to be results of an independent identically distributed trial from the set $\Omega(n)$. Note that the set $\Omega(n)$ contains 2^n elements e with $\lambda_2(e) \leq n$ for $0 \leq e < 2^n$.
For each $2 \leq k \leq n$, the set $\Omega_k = \{e \in \Omega(n) \mid \lambda_2(e) = k\}$, which has been defined in remark 2.5, contains 2^{k-1} elements from $(2^{k-1})_2 = (10^{k-1})$ to $(2^k - 1)_2 = (1^k)$. For $k = 1$, the set $\Omega_1 = \{0, 1\}$ contains 2 elements. Note that the sets $\Omega_1, \Omega_2, \ldots, \Omega_n$ form a disjoint decomposition of the set $\Omega(n)$.
For the Brauer method, the window width d is assumed to be determined once for the whole algorithm, hence, it is assumed not to depend on the input (which would also be a possible implementation).

Lemma 4.23 (average length of the input's binary expansion)
The average length of the binary expansions of inputs $e \in \Omega(n)$, e i.i.d., is

$$\frac{1}{2^{n-1}} + (n-1) \;\approx\; n-1.$$

Proof:

As the Ω_k, $1 \leq k \leq n$, form a disjoint decomposition of $\Omega(n)$, the average length of inputs i.i.d. from $\Omega(n)$ is

$$\begin{aligned}
\frac{1}{\#\Omega(n)} \cdot \sum_{k=1}^{n} k \cdot \#\Omega_k \;&=\; \frac{1}{2^n} \cdot \left(\sum_{k=1}^{n} k \cdot 2^{k-1} + 1 \right) \\
&=\; \frac{1}{2^n} \cdot \left(\frac{1 - (n+1) \cdot 2^n + n \cdot 2^{n+1}}{(1-2)^2} + 1 \right) && \text{(see [BSMM95], table 1.2(10), p. 16)} \\
&=\; \frac{1}{2^n} \cdot (2 + 2^n \cdot (n-1)) && (4.24) \\
&=\; \frac{1}{2^{n-1}} + (n-1) \;\approx\; n-1.
\end{aligned}$$

□

The window width d can be chosen according to that number, for example, the optimal value of d presented in section 2.3.6, could be chosen dependent on $n - 1$ rather than dependent on n. However, it is also always possible

to determine the window width dependent on the input length. For the remainder of this chapter, the window value d will not be specified explicitly, because in practice, the formulas presented in section 2.3.6 are not taken as they are to determine the optimal window width, but the surrounding 2 or 4 values for d are also tried.
Using definition 4.22, the results from corollary 2.15 (for the traditional Brauer method) and theorem 4.16 (for the NAF-based Brauer method) can be extended to the new average case by taking the known average results for each set Ω_k, which is the average of all values within Ω_k, if they are assumed to be i.i.d., and adding them up with their specific weight, e.g. multiplied with the fraction of the set Ω_k from the set $\Omega(n)$. The following theorems state the derived results. But first, a lemma needed to prove the theorems will be established.

Lemma 4.24 (weighted Gaussian bracket sums)
For $n, d \in \mathbb{N}$, $d < n$, let k be uniquely defined by the division with remainder $n = k \cdot d + r$, $0 \leq r < d$. Then the following equations hold:

$$\sum_{i=1}^{n} \left(2^i \cdot \left\lceil \frac{i}{d} \right\rceil \right) = (k+1) \cdot 2^{n+1} - \frac{2}{2^d - 1} \cdot \left(2^{k \cdot d + d} - 1 \right) \tag{4.25}$$

$$\sum_{i=1}^{n} \left(2^i \cdot \left\lceil \frac{i+1}{d} \right\rceil \right) = (k+1) \cdot 2^n + 2^n \cdot \left\lceil \frac{n+1}{d} \right\rceil - \frac{2^{k \cdot d + d} - 1}{2^d - 1} - 1 \tag{4.26}$$

Proof:

first equation:

$$\begin{aligned}
\sum_{i=1}^{n} \left(2^i \cdot \left\lceil \frac{i}{d} \right\rceil \right) &= \sum_{j=1}^{k} \left(j \cdot \sum_{i=(j-1) \cdot d+1}^{j \cdot d} 2^i \right) + \sum_{i=k \cdot d+1}^{n} \left((k+1) \cdot 2^i \right) \\
&= \sum_{j=1}^{k} \left(j \cdot \left(2^{j \cdot d+1} - 2^{(j-1) \cdot d+1} \right) \right) + (k+1) \cdot \left(2^{n+1} - 2^{k \cdot d+1} \right) \\
&= \left(2^{d+1} - 2 \right) \cdot \sum_{j=1}^{k} \left(j \cdot \left(2^d \right)^{j-1} \right) + (k+1) \cdot \left(2^{n+1} - 2^{k \cdot d+1} \right) \\
&\overset{(*)}{=} 2 \cdot \left(2^d - 1 \right) \cdot \frac{1 - (k+1) \cdot 2^{k \cdot d} + k \cdot 2^{d \cdot (k+1)}}{(1 - 2^d)^2} \\
&\quad + (k+1) \cdot \left(2^{n+1} - 2^{k \cdot d+1} \right)
\end{aligned}$$

$$
\begin{aligned}
&= \frac{2}{2^d-1} \cdot \left(1 - 2^{k\cdot d} \cdot \left(1 + k - k \cdot 2^d\right)\right) \\
&\quad + \frac{2}{2^d-1} \cdot \left(2^d - 1\right) \cdot (k+1) \cdot \left(2^n - 2^{k\cdot d}\right) \\
&= (k+1) \cdot 2^{n+1} + \frac{2}{2^d-1} \cdot \Big(1 - 2^{k\cdot d} - k \cdot 2^{k\cdot d} + k \cdot 2^{k\cdot d+d} \\
&\quad - k \cdot 2^{k\cdot d+d} - 2^{k\cdot d+d} + k \cdot 2^{k\cdot d} + 2^{k\cdot d}\Big) \\
&= (k+1) \cdot 2^{n+1} - \frac{2}{2^d-1} \cdot \left(2^{k\cdot d+d} - 1\right)
\end{aligned}
$$

The equality $(*)$ can be found in some collections of values of finite series, for example in [BSMM95], table 1.2(10), page 16. It is derived from the differentiated finite geometric series.

second equation:

$$
\begin{aligned}
\sum_{i=1}^{n} \left(2^i \cdot \left\lceil \frac{i+1}{d} \right\rceil\right) &= \frac{1}{2} \cdot \sum_{i=1}^{n} \left(2^{i+1} \cdot \left\lceil \frac{i+1}{d} \right\rceil\right) = \frac{1}{2} \cdot \sum_{i=2}^{n+1} \left(2^i \cdot \left\lceil \frac{i}{d} \right\rceil\right) \\
&= \frac{1}{2} \cdot \sum_{i=1}^{n} \left(2^i \cdot \left\lceil \frac{i}{d} \right\rceil\right) - \frac{1}{2} \cdot 2 \cdot \left\lceil \frac{1}{d} \right\rceil + \frac{1}{2} \cdot 2^{n+1} \cdot \left\lceil \frac{n+1}{d} \right\rceil \\
&\overset{(4.25)}{=} \frac{1}{2} \cdot \left((k+1) \cdot 2^{n+1} - \frac{2}{2^d-1} \cdot \left(2^{k\cdot d+d} - 1\right)\right) - 1 \\
&\quad + 2^n \cdot \left\lceil \frac{n+1}{d} \right\rceil \\
&= (k+1) \cdot 2^n + 2^n \cdot \left\lceil \frac{n+1}{d} \right\rceil - \frac{2^{k\cdot d+d} - 1}{2^d - 1} - 1
\end{aligned}
$$

□

First, the generalized average case for the traditional Brauer method is formulated.

Theorem 4.25 (generalized traditional Brauer average analysis) *The traditional Brauer method performed on a random input number $e \in \Omega(n)$, e i.i.d., requires the following costs on average. Let $\gamma(2^d) \in \Gamma(2^d)$ be any addition chain suitable to compute all values for the precomputation step (see definition 2.10). Assume that d is fixed for all inputs and k be* *defined as* $k := \left\lfloor \frac{n}{d} \right\rfloor$.

$$Q(2) \quad = \quad k \cdot d - \frac{d}{2^{n-d}} \cdot \frac{2^{k \cdot d} - 1}{2^d - 1} + Q_{\gamma(2^d)}(2) \tag{4.27}$$

$$A \quad = \quad k - \frac{2^{k \cdot d} - 1}{2^n} - \frac{k}{2^d} + A_{\gamma(2^d)} \tag{4.28}$$

Proof:

As the costs of the precomputation do not depend on the definition of the average case, as they do not depend on the inputs, they are not changed. Hence, only the new costs of the main part of the traditional Brauer method need to be shown. Note that for $d \in \mathbb{N}_{>0}$, it is $\left\lceil \frac{1}{d} \right\rceil = 1$. According to table 2.8 on page 67, the average costs for inputs $e \in \Omega_i$, e i.i.d., are

$$Q(2) = d \cdot \left\lceil \frac{i}{d} \right\rceil - d,$$

which gives the average costs for $e \in \Omega(n)$, e i.i.d., d fixed, as

$$\begin{aligned}
Q(2) \quad &= \quad \frac{1}{\#\Omega(n)} \cdot \sum_{i=1}^{n} \left(\#\Omega_i \cdot \left(d \cdot \left\lceil \frac{i}{d} \right\rceil - d \right) \right) \\
&= \quad \frac{d}{2^n} \cdot \left(\sum_{i=1}^{n} \left(2^{i-1} \cdot \left\lceil \frac{i}{d} \right\rceil \right) + \left\lceil \frac{1}{d} \right\rceil \right) - d \\
&= \quad \frac{d}{2^{n+1}} \cdot \sum_{i=1}^{n} \left(2^i \cdot \left\lceil \frac{i}{d} \right\rceil \right) + \frac{d}{2^n} - d \\
&\overset{(4.25)}{=} \quad \frac{d}{2^{n+1}} \cdot \left((k+1) \cdot 2^{n+1} - \frac{2}{2^d - 1} \cdot \left(2^{k \cdot d + d} - 1 \right) \right) + \frac{d}{2^n} - d \\
&= \quad k \cdot d - \frac{d \cdot \left(2^{k \cdot d + d} - 1 \right)}{2^n \cdot (2^d - 1)} + \frac{d \cdot \left(2^d - 1 \right)}{2^n \cdot (2^d - 1)} \\
&= \quad k \cdot d - \frac{d \cdot \left(2^{k \cdot d + d} - 2^d \right)}{2^n \cdot (2^d - 1)} \quad = \quad k \cdot d - \frac{d}{2^{n-d}} \cdot \frac{2^{k \cdot d} - 1}{2^d - 1}
\end{aligned}$$

The number of additions can also be derived using table 2.8. For inputs $e \in \Omega_k$, e i.i.d., they have been found to be

$$A = \frac{2^d - 1}{2^d} \cdot \left(\left\lceil \frac{k}{d} \right\rceil - 1 \right).$$

This leads to the generalized average case as

$$
\begin{aligned}
A &= \frac{1}{2^n}\cdot\sum_{i=1}^{n}\left(\#\Omega_i\cdot\frac{2^d-1}{2^d}\cdot\left(\left\lceil\frac{i}{d}\right\rceil-1\right)\right)\\
&= \frac{2^d-1}{2^{n+d}}\cdot\left(\sum_{i=1}^{n}\left(2^{i-1}\cdot\left\lceil\frac{i}{d}\right\rceil\right)+\left\lceil\frac{1}{d}\right\rceil\right)-\frac{2^d-1}{2^d}\\
&= \frac{2^d-1}{2^{n+d}}\cdot\sum_{i=1}^{n}\left(2^{i-1}\cdot\left\lceil\frac{i}{d}\right\rceil\right)+\frac{2^d-1}{2^{n+d}}-\frac{2^d-1}{2^d}\\
&= \frac{2^d-1}{2^{n+d+1}}\cdot\sum_{i=1}^{n}\left(2^{i}\cdot\left\lceil\frac{i}{d}\right\rceil\right)-\frac{(2^n-1)\cdot\left(2^d-1\right)}{2^{n+d}}\\
&\overset{(4.25)}{=} \frac{2^d-1}{2^{n+d+1}}\cdot\left((k+1)\cdot 2^{n+1}-\frac{2}{2^d-1}\cdot\left(2^{k\cdot d+d}-1\right)\right)\\
&\quad-\frac{(2^n-1)\cdot\left(2^d-1\right)}{2^{n+d}}\\
&= \frac{2^d-1}{2^d}\cdot(k+1)-\frac{2^{k\cdot d+d}-1+(2^n-1)\cdot\left(2^d-1\right)}{2^{n+d}}\\
&= (k+1)-\frac{k+1}{2^d}-\frac{2^{k\cdot d+d}-2^n-2^d}{2^{n+d}}-1\\
&= k-\frac{k}{2^d}-\frac{2^{k\cdot d}-1}{2^n}
\end{aligned}
$$

□

Now the generalized results for the NAF-based Brauer method can also be formulated:

Theorem 4.26 (refined NAF-based Brauer average analysis)
The NAF-based Brauer method performed on a random input number $e \in \Omega(n)$, e i.i.d., requires on average the following numbers of operations (note that the number of operations in the precomputation step doesn't change with the new definition of the average case and can be found in lemma 4.10). Assume that the window width d is fixed for all inputs and let k be defined

as $k = \lfloor \frac{n}{d} \rfloor$, *then we have:*

$$\begin{aligned} I &= I_{\bar{\gamma}(\hat{n}(d)+1)} \\ Q(2) &= \frac{d}{3} \cdot \left(2k + \left\lceil \frac{n+1}{d} \right\rceil + \frac{1}{2^{n-1}} \cdot \left\lceil \frac{2}{d} \right\rceil - \frac{2^{k \cdot d + d} - 1}{2^{n-1} \cdot (2^d - 1)} - 1 \right) \\ &\quad + Q_{\bar{\gamma}(\hat{n}(d)+1)} \qquad (4.29) \\ A &= \frac{1}{3} \cdot (2k-1) - \frac{2}{3} \cdot \frac{2^{k \cdot d + d} - 1}{2^{n+d}} - \frac{2k-3}{3 \cdot 2^d} \\ &\quad + \left(\frac{1}{3} - \frac{2}{3 \cdot 2^d} \right) \cdot \left(\left\lceil \frac{n+1}{d} \right\rceil + \frac{2}{2^n} \cdot \left\lceil \frac{2}{d} \right\rceil - \frac{1}{2^n} \right) + \frac{1}{3 \cdot 2^n} \\ &\quad + A_{\bar{\gamma}(\hat{n}(d)+1)} \qquad (4.30) \end{aligned}$$

Proof:

The costs of the precomputation step are always the same for any input, if the window width d is not computed for every input but once for the whole application. As this is assumed, the numbers of operations in the precomputation step are always the same, hence, the results depicted in theorem 4.16 apply to the general average case precomputation, too. Only the costs in the main exponentiation step have to be refined. As no inversions occur whatsoever after the precomputation has ended, only the values of $Q(2)$ and A need to be computed.

Note that for $d \in \mathbb{N}_{>0}$, it is $\lceil \frac{1}{d} \rceil = 1$.

For the number of squarings $Q(2)$, the average results for inputs $e \in \Omega_i$, e i.i.d., for some $i \in \mathbb{N}$, have been shown in theorem 4.16 to be

$$Q(2) = \frac{d}{3} \cdot \left\lceil \frac{i}{d} \right\rceil + \frac{2d}{3} \cdot \left\lceil \frac{i+1}{d} \right\rceil - d.$$

Therefore, the average costs of inputs $e \in \Omega(n)$, e i.i.d., are the following. Note that $d \geq 1$.

$$
\begin{aligned}
Q(2) = \quad & \frac{1}{\#\Omega(n)} \cdot \sum_{i=1}^{n} \left(\#\Omega_i \cdot \left(\frac{d}{3} \cdot \left\lceil \frac{i}{d} \right\rceil + \frac{2d}{3} \cdot \left\lceil \frac{i+1}{d} \right\rceil - d \right) \right) \\
= \quad & \frac{d}{3 \cdot 2^n} \cdot \sum_{i=1}^{n} \left(\#\Omega_i \cdot \left\lceil \frac{i}{d} \right\rceil \right) + \frac{2d}{3 \cdot 2^n} \cdot \sum_{i=1}^{n} \left(\#\Omega_i \cdot \left\lceil \frac{i+1}{d} \right\rceil \right) - d \\
= \quad & \frac{d}{3 \cdot 2^n} \cdot \left(\sum_{i=1}^{n} \left(2^{i-1} \cdot \left\lceil \frac{i}{d} \right\rceil \right) + \left\lceil \frac{1}{d} \right\rceil \right) \\
& + \frac{2d}{3 \cdot 2^n} \cdot \left(\sum_{i=1}^{n} \left(2^{i-1} \cdot \left\lceil \frac{i+1}{d} \right\rceil \right) + \left\lceil \frac{2}{d} \right\rceil \right) - d \\
= \quad & \frac{d}{3 \cdot 2^n} \cdot \left(\frac{1}{2} \cdot \sum_{i=1}^{n} \left(2^i \cdot \left\lceil \frac{i}{d} \right\rceil \right) + 1 + \sum_{i=1}^{n} \left(2^i \cdot \left\lceil \frac{i+1}{d} \right\rceil \right) \right. \\
& \left. + 2 \cdot \left\lceil \frac{2}{d} \right\rceil - 3 \cdot 2^n \right) \\
\overset{(*)}{=} \quad & \frac{d}{3 \cdot 2^n} \cdot \left(\frac{1}{2} \cdot \left((k+1) \cdot 2^{n+1} - \frac{2}{2^d - 1} \cdot \left(2^{k \cdot d + d} - 1 \right) \right) \right. \\
& + \ (k+1) \cdot 2^n + 2^n \cdot \left\lceil \frac{n+1}{d} \right\rceil - \frac{2^{k \cdot d + d} - 1}{2^d - 1} - 1 + 1 \\
& \left. + 2 \cdot \left\lceil \frac{2}{d} \right\rceil - 3 \cdot 2^n \right) \\
= \quad & \frac{d}{3 \cdot 2^n} \cdot \left((k+1) \cdot 2^{n+1} - 2 \cdot \frac{2^{k \cdot d + d} - 1}{2^d - 1} + 2^n \cdot \left\lceil \frac{n+1}{d} \right\rceil \right. \\
& \left. + 2 \cdot \left\lceil \frac{2}{d} \right\rceil - 3 \cdot 2^n \right) \\
= \quad & \frac{d}{3} \cdot \left(2k + 2 - \frac{2^{k \cdot d + d} - 1}{2^{n-1} \cdot (2^d - 1)} + \left\lceil \frac{n+1}{d} \right\rceil + \frac{1}{2^{n-1}} \cdot \left\lceil \frac{2}{d} \right\rceil - 3 \right) \\
= \quad & \frac{d}{3} \cdot \left(2k + \left\lceil \frac{n+1}{d} \right\rceil + \frac{1}{2^{n-1}} \cdot \left\lceil \frac{2}{d} \right\rceil - \frac{2^{k \cdot d + d} - 1}{2^{n-1} \cdot (2^d - 1)} - 1 \right)
\end{aligned}
$$

Note that $(*)$ follows with formulas (4.25) and (4.26).

For the average number of multiplications for inputs $e \in \Omega_i$, e i.i.d., for some $i \in \mathbb{N}$, theorem 4.16 states that

$$
A = \frac{1}{3} \cdot \left\lceil \frac{i}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \left\lceil \frac{i+1}{d} \right\rceil - \frac{2^d - \frac{5}{3}}{2^d}.
$$

Therefore, the average number of multiplications for inputs $e \in \Omega(n)$, e i.i.d., are

$$
\begin{aligned}
A \;=\;& \frac{1}{\#\Omega(n)} \cdot \sum_{i=1}^{n} \left(\#\Omega_i \cdot \left(\frac{1}{3} \cdot \left\lceil \frac{i}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \left\lceil \frac{i+1}{d} \right\rceil \right.\right. \\
& \left.\left. - \frac{2^d - \frac{5}{3}}{2^d} \right)\right) \\
=\;& \frac{1}{3 \cdot 2^n} \cdot \sum_{i=1}^{n} \left(\#\Omega_i \cdot \left\lceil \frac{i}{d} \right\rceil \right) \\
& + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \frac{1}{2^n} \cdot \sum_{i=1}^{n} \left(\#\Omega_i \cdot \left\lceil \frac{i+1}{d} \right\rceil \right) - \frac{2^d - \frac{5}{3}}{2^d} \\
=\;& \frac{1}{3 \cdot 2^n} \cdot \left(\sum_{i=1}^{n} \left(2^{i-1} \cdot \left\lceil \frac{i}{d} \right\rceil \right) + \left\lceil \frac{i}{d} \right\rceil \right) - \frac{2^d - \frac{5}{3}}{2^d} \\
& + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \frac{1}{2^n} \cdot \left(\sum_{i=1}^{n} \left(2^{i-1} \cdot \left\lceil \frac{i-1}{d} \right\rceil \right) + \left\lceil \frac{2}{d} \right\rceil \right) \\
=\;& \frac{1}{3 \cdot 2^{n+1}} \cdot \sum_{i=1}^{n} \left(2^i \cdot \left\lceil \frac{i}{d} \right\rceil \right) + \frac{1}{3 \cdot 2^n} + \left(\frac{1}{3} - \frac{2}{3 \cdot 2^d} \right) \cdot \frac{1}{2^n} \cdot \\
& \sum_{i=1}^{n} \left(2^i \cdot \left\lceil \frac{i+1}{d} \right\rceil \right) + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \frac{1}{2^n} \cdot \left\lceil \frac{2}{d} \right\rceil - \frac{2^d - \frac{5}{3}}{2^d} \\
\overset{(*)}{=}\;& \frac{1}{3 \cdot 2^{n+1}} \cdot \left((k+1) \cdot 2^{n+1} - \frac{2}{2^d - 1} \cdot \left(2^{k \cdot d + d} - 1 \right) \right) \\
& + \frac{1}{3 \cdot 2^n} + \left(\frac{1}{3} - \frac{2}{3 \cdot 2^d} \right) \cdot \frac{1}{2^n} \cdot \left((k+1) \cdot 2^n + 2^n \cdot \left\lceil \frac{n+1}{d} \right\rceil \right. \\
& \left. - \frac{2^{k \cdot d + d} - 1}{2^d - 1} - 1 \right) + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \frac{1}{2^n} \cdot \left\lceil \frac{2}{d} \right\rceil - \frac{2^d - \frac{5}{3}}{2^d} \\
=\;& \frac{k+1}{3} - \frac{2^{k \cdot d + d} - 1}{3 \cdot 2^n \cdot (2^d - 1)} + \frac{1}{3 \cdot 2^n} + \frac{k+1}{3} - \frac{2k - 2}{3 \cdot 2^d} \\
& + \left(\frac{1}{3} - \frac{2}{3 \cdot 2^d} \right) \cdot \left\lceil \frac{n+1}{d} \right\rceil - \left(\frac{1}{3} - \frac{2}{3 \cdot 2^d} \right) \cdot \frac{2^{k \cdot d + d} - 1}{2^n \cdot (2^d - 1)} \\
& - \left(\frac{1}{3} - \frac{2}{3 \cdot 2^d} \right) \cdot \frac{1}{2^n} + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \frac{1}{2^n} \cdot \left\lceil \frac{2}{d} \right\rceil - \frac{2^d - \frac{5}{3}}{2^d}
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{2}{3}\cdot(k+1) - \frac{2}{3\cdot 2^n}\cdot\frac{2^{k\cdot d+d}-1}{2^d-1} + \frac{2}{3\cdot 2^{n+d}}\cdot\frac{2^{k\cdot d+d}-1}{2^d-1} \\
&\quad - \frac{2k+2}{3\cdot 2^d} + \left(\frac{1}{3}-\frac{2}{3\cdot 2^d}\right)\cdot\left(\left\lceil\frac{n+1}{d}\right\rceil - \frac{1}{2^n} + \frac{1}{2^{n-1}}\cdot\left\lceil\frac{2}{d}\right\rceil\right) \\
&\quad - \frac{2^d-\frac{5}{3}}{2^d} + \frac{1}{3\cdot 2^n} \\
&= \frac{2}{3}\cdot(k+1) - \frac{2}{3}\cdot\frac{2^d\cdot\left(2^{k\cdot d+d}-1\right)+2^{k\cdot d+d}-1}{2^{n+d}\cdot(2^d-1)} - \frac{3\cdot 2^d+2k-3}{3\cdot 2^d} \\
&\quad + \left(\frac{1}{3}-\frac{2}{3\cdot 2^d}\right)\cdot\left(\left\lceil\frac{n+1}{d}\right\rceil + \frac{2}{2^n}\cdot\left\lceil\frac{2}{d}\right\rceil - \frac{1}{2^n}\right) + \frac{1}{3\cdot 2^n} \\
&= \frac{1}{3}\cdot(2k-1) - \frac{2}{3}\cdot\frac{2^{k\cdot d+d}-1}{2^{n+d}} - \frac{2k-3}{3\cdot 2^d} \\
&\quad + \left(\frac{1}{3}-\frac{2}{3\cdot 2^d}\right)\cdot\left(\left\lceil\frac{n+1}{d}\right\rceil + \frac{2}{2^n}\cdot\left\lceil\frac{2}{d}\right\rceil - \frac{1}{2^n}\right) + \frac{1}{3\cdot 2^n}
\end{aligned}
$$

The equality $(*)$ follows again from formulas (4.25) and (4.26). □

With these results for the general average case of both variants of the Brauer method, a new cost comparison for the average case can be formulated.

Theorem 4.27 (generalized average Brauer cost comparison)
Let $c(I)$, $c(Q(2))$ and $c(A)$ be known and declared as in theorem 4.17, let n define the set $\Omega(n)$ of all possible inputs.
Then the following inequality can be used to decide whether to use the traditional Brauer method or the NAF-based Brauer method in order to compute x^e, if the inputs $e \in \Omega(n)$ are i.i.d.
step 1:
Perform step 1 of theorem 4.19 to get c_1.
step 2:
Perform step 2 of theorem 4.19 to get c_2.
step 3:
With c_1 and c_2 known, set

$$
\begin{aligned}
R &:= c(Q(2))\cdot\frac{d}{3}\cdot\left(\left\lceil\frac{n+1}{d}\right\rceil + \frac{2}{2^n}\cdot\left\lceil\frac{2}{d}\right\rceil + \frac{2^{k\cdot d+d}-3\cdot 2^d+2}{2^n\cdot(2^d-1)} - k - 1\right) \\
&\quad + \frac{c(A)}{3}\cdot\left(\frac{k+3}{2^d} - k - 1 + \frac{2^d-2}{2^d}\cdot\left(\left\lceil\frac{n+1}{d}\right\rceil + \frac{2}{2^n}\cdot\left\lceil\frac{2}{d}\right\rceil\right)\right. \\
&\quad \left. + \frac{2^{k\cdot d}-3}{2^n} + \frac{4}{2^{n+d}}\right) + c_2 - c_1
\end{aligned}
$$

If $R > 0$, then the ordinary Brauer method is faster, otherwise the NAF-based Brauer method is faster.

Proof:

step 1 and step 2

The precomputation step is independent from the configuration of the input, the same operations are carried out for every input. Therefore, the same results as in theorem 4.19 apply. This can also be seen in lemma 2.11 and lemma 4.10.

step 3

For step 3, recall the generalized average case analysis of the traditional Brauer method depicted in theorem 4.25, where the following costs have been derived to form formulas (4.27) and (4.28):

$$\begin{aligned} & c(Q(2)) \cdot \left(k \cdot d - \frac{d}{2^{n-d}} \cdot \frac{2^{k \cdot d} - 1}{2^d - 1}\right) \\ + \quad & c(A) \cdot \left(k - \frac{2^{k \cdot d} - 1}{2^n} - \frac{k}{2^d}\right) + c_1 \end{aligned} \tag{4.31}$$

And recall that the costs derived for the generalized average case of the NAF-based Brauer method from theorem 4.26, formulas (4.29) and (4.30) are

$$\begin{aligned} & c(Q(2)) \cdot \left(\frac{d}{3} \cdot \left(2k + \left\lceil \frac{n+1}{d} \right\rceil + \frac{1}{2^{n-1}} \cdot \left\lceil \frac{2}{d} \right\rceil - \frac{2^{k \cdot d + d} - 1}{2^{n-1} \cdot (2^d - 1)} - 1\right)\right) \\ + \quad & c(A) \cdot \left(\frac{1}{3} \cdot (2k-1) - \frac{2}{3} \cdot \frac{2^{k \cdot d + d} - 1}{2^{n+d}} - \frac{2k-3}{3 \cdot 2^d} + \left(\frac{1}{3} - \frac{2}{3 \cdot 2^d}\right) \cdot \right. \\ & \left. \left(\left\lceil \frac{n+1}{d} \right\rceil + \frac{2}{2^n} \cdot \left\lceil \frac{2}{d} \right\rceil - \frac{1}{2^n}\right) + \frac{1}{3 \cdot 2^n}\right) + c_2 \end{aligned} \tag{4.32}$$

Subtracting formula (4.31) from (4.32) leads to the claimed result of

$$
\begin{aligned}
&(4.32) - (4.31)\\
= \;& c(Q(2)) \cdot \left[\frac{2}{3} \cdot d \cdot k + \frac{d}{3} \cdot \left\lceil \frac{n+1}{d} \right\rceil + \frac{2d}{3 \cdot 2^n} \cdot \left\lceil \frac{2}{d} \right\rceil - \frac{d}{3} \right.\\
& \left. - \frac{d}{3} \cdot \frac{2 \cdot 2^{k \cdot d + d} - 2}{2^n \cdot (2^d - 1)} - k \cdot d + \frac{d}{3 \cdot 2^n} \cdot \frac{3 \cdot 2^{k \cdot d + d} - 3 \cdot 2^d}{2^d - 1} \right]\\
& + c(A) \cdot \left[-\frac{1}{3} \cdot (k+1) + \frac{k+3}{3 \cdot 2^d} - \frac{2 \cdot 2^{k \cdot d + d} - 2}{3 \cdot 2^{n+d}} \right.\\
& + \frac{2^d - 2}{3 \cdot 2^d} \cdot \left(\left\lceil \frac{n+1}{d} \right\rceil + \frac{2}{2^n} \cdot \left\lceil \frac{2}{d} \right\rceil \right) - \frac{2^d - 2}{3 \cdot 2^{n+d}} + \frac{2^d}{3 \cdot 2^{n+d}}\\
& \left. + \frac{3 \cdot 2^{k \cdot d + d} - 3 \cdot 2^d}{3 \cdot 2^{n+d}} \right] + c_2 - c_1\\
= \;& c(Q(2)) \cdot \left[\frac{d}{3} \cdot (-k - 1) + \frac{d}{3} \cdot \left(\left\lceil \frac{n+1}{d} \right\rceil + \frac{2}{2^n} \cdot \left\lceil \frac{2}{d} \right\rceil \right) \right.\\
& \left. + \frac{d}{3} \cdot \frac{2^{k \cdot d + d} - 3 \cdot 2^d + 2}{2^n \cdot (2^d - 1)} \right] + c(A) \cdot \left[\frac{k+3}{3 \cdot 2^d} - \frac{1}{3} \cdot (k+1) \right.\\
& + \frac{2^d - 2}{3 \cdot 2^d} \cdot \left(\left\lceil \frac{n+1}{d} \right\rceil + \frac{2}{2^n} \cdot \left\lceil \frac{2}{d} \right\rceil \right)\\
& \left. + \frac{3 \cdot 2^{k \cdot d + d} - 3 \cdot 2^d - 2 \cdot 2^{k \cdot d + d} + 2 - 2^d + 2 + 2^d}{3 \cdot 2^{n+d}} \right] + c_2 - c_1\\
= \;& c(Q(2)) \cdot \frac{d}{3} \cdot \left(\left\lceil \frac{n+1}{d} \right\rceil + \frac{2}{2^n} \cdot \left\lceil \frac{2}{d} \right\rceil + \frac{2^{k \cdot d + d} - 3 \cdot 2^d + 2}{2^n \cdot (2^d - 1)} - k - 1 \right)\\
& + \frac{c(A)}{3} \cdot \left(\frac{k+3}{2^d} - k - 1 + \frac{2^d - 2}{2^d} \cdot \left(\left\lceil \frac{n+1}{d} \right\rceil + \frac{2}{2^n} \cdot \left\lceil \frac{2}{d} \right\rceil \right) \right.\\
& \left. + \frac{2^{k \cdot d} - 3}{2^n} + \frac{4}{2^{n+d}} \right) + c_2 - c_1
\end{aligned}
$$

□

4.5 Conclusions

Starting from the problem to solve a general exponentiation problem $\Pi = (x, e)$, the results of this thesis are the analyses of the binary method and Brauer's method to generate addition chains using both the binary expansion and the non-adjacent form (NAF) as basis for the representation of the exponent e. The NAF was introduced and several properties of that special signed binary digit representation were shown in order to give results of the performance of the NAF-based variants of the binary method and Brauer's method. Some of the results are new to this thesis. The results most interesting for practical applications are the results concerning the worst case and the average case. They are presented next to each other for an easy overview on pages 164 and 165 (tables 4.7 and 4.8). Note that for the precomputation, it has been shown that there exist addition chains γ_1, γ_2 (lemma 2.11) and addition-subtraction chains $\bar{\gamma}_1, \bar{\gamma}_2, \bar{\gamma}_3, \bar{\gamma}_4$ (theorem 4.16) which require the stated numbers of operations. These choices are not the only possible choices, but other choices are not needed. Results concerning a more general definition of "average case" have been shown in section 4.4.

The main result of this thesis is the construction of application oriented cost comparisons, stated in algorithmic form. They can be used to decide which representation of the exponent e yields the minimal costs for a chosen method and binary length of the exponent e. These cost comparison inequalities are once again depicted in table 4.9 on page 166.

The analyses within this thesis have shown that the approach to judge algorithms for fast exponentiations only by comparing the total number of arithmetical operations doesn't relate to reality very well. In the opposite, the same algorithm may be optimal (like the traditional binary method for inputs e of the form $e = 2^k$ for some k), or not to recommend (like the binary method in the general case, because it is only a special case of the m-ary method, where an optimal window size can be easily determined).

But this is not only the fact for inputs of a special form, it also depends on the actual implementation of the system, where fast exponentiation is needed. Examples have been shown to emphasize that the same practical problem (for example elliptic curve exponentiation) may have different solutions with very different costs (see example 2.4 on page 54). These costs have a great impact on the performance of the chosen algorithms, as all algorithms use the basic operations of inversion, multiplication and squaring with different ratios. The examples in section 4.3 have shown this effect. Therefore, for application purposes, one must pay respect to this problem and analyze algorithms with a view on the operations involved.

The search for a fast algorithm that computes an optimal addition chain will not be successful, as this cannot be done in polynomial time. But although the search for an excellent suboptimal algorithm is an interesting and valuable task for mathematicians, the practical background requires to

The binary expansion-based algorithms

costs of the binary expansion-based binary method

worst case costs		average case costs	
$I =$	0	$I =$	0
$Q(2) =$	$\lambda_2(e) - 1$	$Q(2) =$	$\lambda_2(e) - 1$
$A =$	$\lambda_2(e) - 1$	$A =$	$\frac{1}{2} \cdot (\lambda_2(e) - 1)$

costs of the binary expansion-based Brauer method

	worst case costs
$I =$	0
$Q(2) =$	$\left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) \cdot d + Q_{\gamma(2^d)}(2)$
$A =$	$\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1 + A_{\gamma(2^d)}$

	average costs
$I =$	0
$Q(2) =$	$\left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) \cdot d + Q_{\gamma(2^d)}(2)$
$A =$	$\frac{2^d-1}{2^d} \cdot \left(\left\lceil \frac{\lambda_2(e)}{d} \right\rceil - 1\right) + A_{\gamma(2^d)}$

important possible values for the precomputation

	$\boldsymbol{I_{\gamma(2^2d)}}$	$\boldsymbol{Q_{\gamma(2^d)}(2)}$	$\boldsymbol{A_{\gamma(2^d)}}$
$\gamma(2^d) = \gamma_1$	0	1	$2^d - 3$
$\gamma(2^d) = \gamma_2$	0	$\left\lceil \frac{2^d-2}{2} \right\rceil$	$\left\lfloor \frac{2^d-2}{2} \right\rfloor$

Table 4.7: Final overview: the costs of the binary expansion-based methods

The NAF-based algorithms

costs of the NAF-based binary method

worst case costs		**average costs** (for d large enough)	
$I =$	1	$I =$	1
$Q(2) =$	$\lambda_2(e)$	$Q(2) =$	$\lambda_2(e) - \frac{1}{3}$
$A =$	$\left\lfloor \frac{\lambda_2(e)}{2} \right\rfloor$	$A =$	$\frac{1}{3} \cdot \lambda_2(e) - \frac{7}{9}$

costs of the NAF-based Brauer method

worst case costs	
$I =$	$I_{\bar{\gamma}(\hat{n}(d)+1)}$
$Q(2) =$	$d \cdot \left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor + Q_{\bar{\gamma}(\hat{n}(d)+1)}(2)$
$A =$	$\left\lfloor \frac{\lambda_2(e)}{d} \right\rfloor + A_{\bar{\gamma}(\hat{n}(d)+1)}$

average costs (for d large enough)	
$I =$	$I_{\bar{\gamma}(\hat{n}(d)+1)}$
$Q(2) =$	$\frac{d}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \frac{2d}{3} \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - d + Q_{\bar{\gamma}(\hat{n}(d)+1)}(2)$
$A =$	$\frac{1}{3} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d}\right) \cdot \left\lceil \frac{\lambda_2(e)+1}{d} \right\rceil - \frac{2^d - \frac{5}{3}}{2^d} + A_{\bar{\gamma}(\hat{n}(d)+1)}$

important possible values for the precomputation

	$\boldsymbol{I_{\bar{\gamma}(\hat{n}(d)+1)}}$	$\boldsymbol{Q_{\bar{\gamma}(\hat{n}(d)+1)}}$	$\boldsymbol{A_{\bar{\gamma}(\hat{n}(d)+1)}}$
$\bar{\gamma}(\hat{n}(d)+1) = \bar{\gamma}_1$	1	$s_2(d-1) - 1$	$2 \cdot s_2(d-2) - 2$
$\bar{\gamma}(\hat{n}(d)+1) = \bar{\gamma}_2$	$\frac{1}{2} \cdot s_2(d) - \frac{1}{2}$	$\frac{1}{2} \cdot s_2(d-1) - \frac{1}{2}$	$s_2(d-2) - 1$
$\bar{\gamma}(\hat{n}(d)+1) = \bar{\gamma}_3$	$\frac{1}{2} + \frac{1}{2} \cdot s_2(d-1)$	$\frac{1}{2} \cdot s_2(d-1) - \frac{1}{2}$	$2 \cdot s_2(d-2) - 2$
$\bar{\gamma}(\hat{n}(d)+1) = \bar{\gamma}_4$	$s_2(d-2)$	$s_2(d-1) - 1$	$s_2(d-2) - 1$

Table 4.8: Final overview: the costs of the NAF-based Brauer methods

Average cost comparisons

average cost comparison for the binary methods

If

$$R := c(I) + c(Q(2)) \cdot \frac{2}{3} - c(A) \cdot \frac{3\lambda_2(e) + 5}{18} > 0$$

then the traditional binary method is faster,
otherwise the NAF-based binary method should be preferred.

average cost comparison for the Brauer methods

step 1: Compute

$$c_1 \quad := \quad \begin{cases} c(Q(2)) + c(A) \cdot \left(2^d - 3\right) & \text{if } c(A) \leq c(Q(2)) \\ (c(Q(2)) + c(A)) \cdot \left(2^{d-1} - 1\right) & \text{otherwise,} \end{cases}$$

step 2: Compute

$$c_{2a} \quad := \quad \begin{cases} (c(I) + c(A)) \cdot (s_2(d-2) - 1) & \text{if } c(I) \leq c(A) \\ 2 \cdot c(A) \cdot (s_2(d-2) - 1) & \text{otherwise} \end{cases}$$

and

$$c_{2b} \quad := \quad \begin{cases} (c(I) + c(Q(2))) \cdot \left(\frac{1}{2} \cdot s_2(d-1) - \frac{1}{2}\right) & \text{if } c(I) \leq c(Q(2)) \\ c(Q(2)) \cdot (s_2(d-1) - 1) & \text{otherwise,} \end{cases}$$

and set

$$c_2 \quad : \quad = c(I) + c_{2a} + c_{2b},$$

step 3: If for

$$\begin{aligned} R \ := \ c(A) \cdot & \left(\frac{3 - 2^{d+1}}{3 \cdot 2^d} \cdot \left\lceil \frac{\lambda_2(e)}{d} \right\rceil + \left(\frac{2}{3} - \frac{4}{3 \cdot 2^d} \right) \cdot \left\lceil \frac{\lambda_2(e) + 1}{d} \right\rceil + \frac{2}{3 \cdot 2^d} \right) \\ & + c(Q(2)) \cdot \frac{2d}{3} \cdot \left(\left\lceil \frac{\lambda_2(e) + 1}{d} \right\rceil - \left\lceil \frac{\lambda_2(e)}{d} \right\rceil \right) + \ c_2 - c_1, \end{aligned}$$

$R > 0$, then the ordinary Brauer method is faster,
otherwise the NAF-based Brauer method is faster.

Table 4.9: Average cost comparisons derived in section 4.3

not only count operations, but the related computational costs.

This thesis has shown that there is a large number of different approaches to construct fast exponentiation algorithms using addition chains and the most important approaches have been presented in detail in chapter 1. The most popular ones, the binary method and its first generalization, the m-ary method, have been analyzed in detail in chapter 2 and modified into their NAF-based variants in chapters 3 and 4. It has been shown that the NAF can be a valuable improvement, if inversions are cheap.

Future work could include the analyses of other basic approaches, for example the sliding window methods and data compression approaches, based on the NAF. Some work has been done on that subject (for example in [KY98]), but costs have not yet been shown individually for the basic arithmetical operations. There are also promising approaches with signed q-adic digit representations as introduced in section 3.5 and signed digit representations, which are not equal to the NAF, but show other properties. To give a taste of the latter, the following section will give in brief the idea of an approach by K. Koyama and Y. Tsuruoka.

4.6 Beyond NAF based algorithms

The algorithms analyzed in chapter 4 have been based on the binary NAF defined in section 3.3. This has led to cost comparison inequalities in order to decide when this representation can account for a valuable gain of speed in the well known exponentiation algorithms.

Besides the binary NAF, other approaches are possible. In section 3.5, possible generalizations of the NAF for general base representations have been mentioned. It would be possible to use those representations as a basis for Brauer's method, too. This seems not to have been presented in literature so far.

Another approach, which has been attracting the interest of different authors is an idea first published by K. Koyama and Y. Tsuruoka in [KT92]. They suggest another representation of the exponent, which doesn't give the NAF, but has minimal weight, too, and aims to create longer runs of zeros, hence increasing the chance of zero windows for all window methods applied on top of that representation. This is a very promising approach, because it aims to further reduce the number of expected multiplications in the main exponentiation step of Brauer's method. As it creates the same Hamming weight as the NAF, it is not intended to replace the NAF-based binary method.

The proposed algorithm is basically a modification of the transformation rule (3.4) by Morain and Olivos (presented in section 3.3.1). It also searches for subsets of the binary expansion and replaces them according to the following rule:

$$B = (1 \ldots b_i \ldots 1) \quad \longmapsto \quad T = (10 \ldots t_i \ldots \bar{1}) \qquad (4.33)$$
$$\text{where } t_i = b_i - 1$$

The transformation rule is applied, whenever the difference between the number of ones in B (the Hamming weight of B) and the number of zeros in B is greater or equal to 3. In this, it modifies the algorithm of Morain and Olivos, who apply their rule if the difference was greater or equal to 2. While Morain and Olivos had to apply their transformation for this case, too, in order to get the NAF, Koyama and Tsuruoka do not compute the NAF but keep runs of two adjacent nonzero digits.
As the weight of the two representations is the same, the chance of getting zero runs with more than one digit is increased using transformation (4.33) and, hence, the chance of getting zero windows in the d-NAF is higher, thus providing a lower average number of multiplications.
It is not known if this possible improvement will be significant. But this approach shows, that there are possibilities beyond the non-adjacent form to gain further speed using signed digit representations of the exponent.

Appendix A

Optimal addition chains

	n	l(n)	example for $S(\chi(n))$	n	l(n)	example for $S(\chi(n))$
1	1	0	(1)	26	6	(1,2,3,5,8,13,26)
	2	1	(1,2)	27	6	(1,2,3,6,9,18,27)
	3	2	(1,2,3)	28	6	(1,2,3,4,7,14,28)
	4	2	(1,2,4)	29	7	(1,2,3,4,7,11,18,29)
	5	3	(1,2,3,5)	30	6	(1,2,3,5,10,15,30)
	6	3	(1,2,3,6)	31	7	(1,2,3,4,7,14,17,31)
	7	4	(1,2,3,4,7)	32	5	(1,2,4,8,16,32)
	8	3	(1,2,4,8)	33	6	(1,2,4,8,16,17,33)
	9	4	(1,2,3,6,9)	34	6	(1,2,4,8,9,17,34)
	10	4	(1,2,3,5,10)	35	7	(1,2,3,4,7,14,21,35)
	11	5	(1,2,3,4,7,11)	36	6	(1,2,3,6,9,18,36)
	12	4	(1,2,3,6,12)	37	7	(1,2,3,5,8,16,21,37)
	13	5	(1,2,3,5,8,13)	38	7	(1,2,3,4,8,11,19,38)
	14	5	(1,2,3,4,7,14)	39	7	(1,2,3,5,8,13,26,39)
	15	5	(1,2,3,5,10,15)	40	6	(1,2,3,5,10,20,40)
	16	4	(1,2,4,8,16)	41	7	(1,2,3,5,10,20,21,41)
	17	5	(1,2,4,8,9,17)	42	7	(1,2,3,4,7,14,21,42)
	18	5	(1,2,3,6,9,18)	43	7	(1,2,3,5,10,20,23,43)
	19	6	(1,2,3,4,8,11,19)	44	7	(1,2,3,4,7,11,22,44)
	20	5	(1,2,3,5,10,20)	45	7	(1,2,3,5,10,15,30,45)
	21	6	(1,2,3,4,7,14,21)	46	7	(1,2,3,5,10,13,23,46)
	22	6	(1,2,3,4,7,11,22)	47	8	(1,2,3,4,7,10,20,27,47)
	23	6	(1,2,3,5,10,13,23)	48	6	(1,2,3,6,12,24,48)
	24	5	(1,2,3,6,12,24)	49	7	(1,2,3,6,12,24,25,49)
	25	6	(1,2,3,5,10,15,25)	50	7	(1,2,3,5,10,15,25,50)
51	51	7	(1,2,3,6,12,24,27,51)	76	8	(1,2,3,4,8,11,19,38,76)
	52	7	(1,2,3,5,8,13,26,52)	77	8	(1,2,4,5,9,18,36,72,77)
	53	8	(1,2,3,5,6,12,24,29,53)	78	8	(1,2,3,5,8,13,26,52,78)
	54	7	(1,2,3,6,9,18,27,54)	79	9	(1,2,3,4,7,9,18,36,72,79)
	55	8	(1,2,3,4,7,11,22,33,55)	80	7	(1,2,3,5,10,20,40,80)
	56	7	(1,2,3,4,7,14,28,56)	81	8	(1,2,3,5,10,20,40,80,81)
	57	8	(1,2,3,4,7,14,28,29,57)	82	8	(1,2,3,5,10,20,21,41,82)
	58	8	(1,2,3,4,7,11,18,29,58)	83	8	(1,2,3,5,10,20,40,80,83)
	59	8	(1,2,3,4,7,14,28,31,59)	84	8	(1,2,3,4,7,14,21,42,84)
	60	7	(1,2,3,5,10,15,30,60)	85	8	(1,2,3,5,10,20,40,80,85)
	61	8	(1,2,3,5,7,14,28,33,61)	86	8	(1,2,3,5,10,20,23,43,86)
	62	8	(1,2,3,4,7,14,17,31,62)	87	9	(1,2,3,4,7,10,20,40,80,87)
	63	8	(1,2,3,4,7,14,21,42,63)	88	8	(1,2,3,4,7,11,22,44,88)
	64	6	(1,2,4,8,16,32,64)	89	9	(1,2,3,4,7,11,22,44,88,89)

	65	7	(1,2,4,8,16,32,33,65)	90	8	(1,2,3,5,10,15,30,60,90)
	66	7	(1,2,4,8,16,17,33,66)	91	9	(1,2,3,4,7,11,22,44,88,91)
	67	8	(1,2,3,4,8,16,32,35,67)	92	8	(1,2,3,5,10,13,23,46,92)
	68	7	(1,2,4,8,9,17,34,68)	93	9	(1,2,3,4,7,14,17,31,62,93)
	69	8	(1,2,3,5,8,16,32,37,69)	94	9	(1,2,3,4,7,10,20,27,47,94)
	70	8	(1,2,3,4,7,14,21,35,70)	95	9	(1,2,3,4,7,11,22,44,88,95)
	71	9	(1,2,3,4,7,8,16,32,39,71)	96	7	(1,2,3,6,12,24,48,96)
	72	7	(1,2,3,6,9,18,36,72)	97	8	(1,2,3,6,12,24,48,96,97)
	73	8	(1,2,3,6,9,18,36,37,73)	98	8	(1,2,3,6,12,24,25,49,98)
	74	8	(1,2,3,5,8,16,21,37,74)	99	8	(1,2,3,6,12,24,48,96,99)
	75	8	(1,2,3,5,10,15,25,50,75)	100	8	(1,2,3,5,10,15,25,50,100)
101	101	9	(1,2,3,5,6,12,24,48,96,101)	126	9	(1,2,3,4,7,14,21,42,84,126)
	102	8	(1,2,3,6,12,24,27,51,102)	127	10	(1,2,3,4,7,8,15,30,60,120,127)
	103	9	(1,2,3,5,7,12,24,48,96,103)	128	7	(1,2,4,8,16,32,64,128)
	104	8	(1,2,3,5,8,13,26,52,104)	129	8	(1,2,4,8,16,32,64,128,129)
	105	9	(1,2,3,4,7,14,21,35,70,105)	130	8	(1,2,4,8,16,32,33,65,130)
	106	9	(1,2,3,5,6,12,24,29,53,106)	131	9	(1,2,3,4,8,16,32,64,128,131)
	107	9	(1,2,3,5,8,13,26,52,104,107)	132	8	(1,2,4,8,16,17,33,66,132)
	108	8	(1,2,3,6,9,18,27,54,108)	133	9	(1,2,3,5,8,16,32,64,128,133)
	109	9	(1,2,3,5,8,13,26,52,104,109)	134	9	(1,2,3,4,8,16,32,35,67,134)
	110	9	(1,2,3,4,7,11,22,33,55,110)	135	9	(1,2,3,5,10,15,30,45,90,135)
	111	9	(1,2,3,5,8,16,21,37,74,111)	136	8	(1,2,4,8,9,17,34,68,136)
	112	8	(1,2,3,4,7,14,28,56,112)	137	9	(1,2,4,8,9,16,32,64,128,137)
	113	9	(1,2,3,4,7,14,28,56,112,113)	138	9	(1,2,3,5,8,16,32,37,69,138)
	114	9	(1,2,3,4,7,14,28,29,57,114)	139	10	(1,2,3,4,7,10,17,34,68,136,139)
	115	9	(1,2,3,4,7,14,28,56,112,115)	140	9	(1,2,3,4,7,14,21,35,70,140)
	116	9	(1,2,3,4,7,11,18,29,58,116)	141	10	(1,2,3,4,7,10,20,27,47,94,141)
	117	9	(1,2,3,5,7,14,28,56,112,117)	142	10	(1,2,3,4,7,8,16,32,39,71,142)
	118	9	(1,2,3,4,7,14,28,31,59,118)	143	10	(1,2,3,4,7,10,17,34,68,136,143)
	119	9	(1,2,3,4,7,14,28,56,112,119)	144	8	(1,2,3,6,9,18,36,72,144)
	120	8	(1,2,3,5,10,15,30,60,120)	145	9	(1,2,3,6,9,18,36,72,144,145)
	121	9	(1,2,3,5,10,15,30,60,120,121)	146	9	(1,2,3,6,9,18,36,37,73,146)
	122	9	(1,2,3,5,7,14,28,33,61,122)	147	9	(1,2,3,6,9,18,36,72,144,147)
	123	9	(1,2,3,5,10,15,30,60,120,123)	148	9	(1,2,3,5,8,16,21,37,74,148)
	124	9	(1,2,3,4,7,14,17,31,62,124)	149	9	(1,2,4,5,9,18,36,72,144,149)
	125	9	(1,2,3,5,10,15,25,50,100,125)	150	9	(1,2,3,5,10,15,25,50,100,150)
151	151	10	(1,2,3,4,7,9,18,36,72,144,151)	176	9	(1,2,3,4,7,11,22,44,88,176)
	152	9	(1,2,3,4,8,11,19,38,76,152)	177	10	(1,2,3,4,7,11,22,44,88,176,177)
	153	9	(1,2,3,6,9,18,36,72,144,153)	178	10	(1,2,3,4,7,11,22,44,45,89,178)
	154	9	(1,2,4,5,9,18,36,41,77,154)	179	10	(1,2,3,4,7,11,22,44,88,176,179)
	155	10	(1,2,3,4,7,11,18,36,72,144,155)	180	9	(1,2,3,5,10,15,30,45,90,180)
	156	9	(1,2,3,5,8,13,26,39,78,156)	181	10	(1,2,3,5,6,11,22,44,88,176,181)
	157	10	(1,2,3,5,7,12,19,38,76,152,157)	182	10	(1,2,3,4,7,11,22,44,47,91,182)
	158	10	(1,2,3,4,7,9,18,36,43,79,158)	183	10	(1,2,3,4,7,11,22,44,88,176,183)
	159	10	(1,2,3,4,8,16,19,35,70,140,159)	184	9	(1,2,3,5,10,13,23,46,92,184)
	160	8	(1,2,3,5,10,20,40,80,160)	185	10	(1,2,3,5,8,16,21,37,74,148,185)
	161	9	(1,2,3,5,10,20,40,80,160,161)	186	10	(1,2,3,4,7,14,17,31,62,124,186)
	162	9	(1,2,3,5,10,20,40,41,81,162)	187	10	(1,2,3,4,7,11,22,44,88,176,187)
	163	9	(1,2,3,5,10,20,40,80,160,163)	188	10	(1,2,3,4,7,10,20,27,47,94,188)
	164	9	(1,2,3,5,10,20,21,41,82,164)	189	10	(1,2,3,4,7,14,21,42,63,126,189)
	165	9	(1,2,3,5,10,20,40,80,160,165)	190	10	(1,2,3,4,7,11,22,44,51,95,190)
	166	9	(1,2,3,5,10,20,40,43,83,166)	191	11	(1,2,3,4,7,8,15,22,44,88,103,191)
	167	10	(1,2,3,4,7,10,20,40,80,160,167)	192	8	(1,2,3,6,12,24,48,96,192)
	168	9	(1,2,3,4,7,14,21,42,84,168)	193	9	(1,2,3,6,12,24,48,96,97,193)
	169	10	(1,2,3,4,7,14,21,42,84,168,169)	194	9	(1,2,3,6,12,24,48,49,97,194)
	170	9	(1,2,3,5,10,20,40,45,85,170)	195	9	(1,2,3,6,12,24,48,96,99,195)
	171	10	(1,2,3,4,7,14,21,42,84,168,171)	196	9	(1,2,3,6,12,24,25,49,98,196)
	172	9	(1,2,3,5,10,20,23,43,86,172)	197	10	(1,2,3,5,6,12,24,48,96,101,197)

	173	10	(1,2,3,5,7,14,21,42,84,168,173)	198	9	(1,2,3,6,12,24,48,51,99,198)
	174	10	(1,2,3,4,7,10,20,40,47,87,174)	199	10	(1,2,3,5,7,12,24,48,96,192,199)
	175	10	(1,2,3,4,7,14,21,35,70,140,175)	200	9	(1,2,3,5,10,15,25,50,100,200)
201	201	10	(1,2,3,4,8,16,32,35,67,134,201)	226	10	(1,2,3,4,7,14,28,56,57,113,226)
	202	10	(1,2,3,5,6,12,24,48,53,101,202)	227	10	(1,2,3,4,7,14,28,56,112,115,227)
	203	10	(1,2,3,5,10,15,25,50,100,200,203)	228	10	(1,2,3,4,7,14,28,29,57,114,228)
	204	9	(1,2,3,6,12,24,27,51,102,204)	229	10	(1,2,3,5,7,14,28,56,112,117,229)
	205	10	(1,2,3,5,10,15,25,50,100,200,205)	230	10	(1,2,3,4,7,14,28,56,59,115,230)
	206	10	(1,2,3,5,7,12,24,48,55,103,206)	231	10	(1,2,3,4,7,14,28,56,112,119,231)
	207	11	(1,2,3,5,8,16,32,37,69,138,207)	232	10	(1,2,3,4,7,11,18,29,58,116,232)
	208	9	(1,2,3,5,8,13,26,52,104,208)	233	10	(1,2,4,5,9,14,28,56,112,121,233)
	209	10	(1,2,3,5,8,13,26,52,104,208,209)	234	10	(1,2,3,5,7,14,28,56,61,117,234)
	210	10	(1,2,3,4,7,14,21,35,70,140,210)	235	11	(1,2,3,4,7,10,20,27,47,94,141,235)
	211	10	(1,2,3,5,8,13,26,52,104,107,211)	236	10	(1,2,3,4,7,14,28,31,59,118,236)
	212	10	(1,2,3,5,6,12,24,29,53,106,212)	237	11	(1,2,3,4,7,9,18,36,43,79,158,237)
	213	10	(1,2,3,5,8,13,26,52,104,109,213)	238	10	(1,2,3,4,7,14,28,56,63,119,238)
	214	10	(1,2,3,5,8,13,26,52,55,107,214)	239	11	(1,2,3,4,7,11,18,29,58,116,123,239)
	215	10	(1,2,3,5,10,15,25,50,100,115,215)	240	9	(1,2,3,5,10,15,30,60,120,240)
	216	9	(1,2,3,6,9,18,27,54,108,216)	241	10	(1,2,3,5,10,15,30,60,120,121,241)
	217	10	(1,2,3,6,9,18,27,54,108,109,217)	242	10	(1,2,3,5,10,15,30,60,61,121,242)
	218	10	(1,2,3,5,8,13,26,52,57,109,218)	243	10	(1,2,3,5,10,15,30,60,120,123,243)
	219	10	(1,2,3,6,9,18,27,54,108,111,219)	244	10	(1,2,3,5,7,14,28,33,61,122,244)
	220	10	(1,2,3,4,7,11,22,33,55,110,220)	245	10	(1,2,3,5,10,15,30,60,120,125,245)
	221	10	(1,2,3,5,8,13,26,52,104,117,221)	246	10	(1,2,3,5,10,15,30,60,63,123,246)
	222	10	(1,2,3,5,8,16,21,37,74,111,222)	247	11	(1,2,3,4,7,8,15,30,60,120,127,247)
	223	11	(1,2,3,4,7,9,18,27,54,108,115,223)	248	10	(1,2,3,4,7,14,17,31,62,124,248)
	224	9	(1,2,3,4,7,14,28,56,112,224)	249	10	(1,2,3,5,10,20,40,43,83,166,249)
	225	10	(1,2,3,4,7,14,28,56,112,113,225)	250	10	(1,2,3,5,10,15,25,50,75,125,250)
251	251	11	(1,2,3,4,7,11,15,30,60,120,131,251)	276	10	(1,2,3,5,8,16,32,37,69,138,276)
	252	10	(1,2,3,4,7,14,21,42,63,126,252)	277	11	(1,2,3,5,6,11,17,34,68,136,141,277)
	253	11	(1,2,3,4,7,11,22,33,55,110,143,253)	278	11	(1,2,3,4,7,10,17,34,68,71,139,278)
	254	11	(1,2,3,4,7,8,15,30,60,67,127,254)	279	11	(1,2,3,4,7,10,17,34,68,136,143,279)
	255	10	(1,2,3,5,10,15,30,60,120,135,255)	280	10	(1,2,3,4,7,14,21,35,70,140,280)
	256	8	(1,2,4,8,16,32,64,128,256)	281	10	(1,2,4,8,9,17,34,68,136,145,281)
	257	9	(1,2,4,8,16,32,64,128,129,257)	282	11	(1,2,3,4,7,10,17,34,68,136,146,282)
	258	9	(1,2,4,8,16,32,64,65,129,258)	283	11	(1,2,3,4,7,14,21,35,70,140,143,283)
	259	10	(1,2,3,4,8,16,32,64,128,131,259)	284	11	(1,2,3,4,7,8,16,32,39,71,142,284)
	260	9	(1,2,4,8,16,32,33,65,130,260)	285	11	(1,2,3,4,7,11,22,44,51,95,190,285)
	261	10	(1,2,3,5,8,16,32,64,128,133,261)	286	11	(1,2,3,4,7,10,17,34,68,75,143,286)
	262	10	(1,2,3,4,8,16,32,64,67,131,262)	287	11	(1,2,3,4,7,14,21,35,70,140,147,287)
	263	11	(1,2,3,4,7,8,16,32,64,128,135,263)	288	9	(1,2,3,6,9,18,36,72,144,288)
	264	9	(1,2,4,8,16,17,33,66,132,264)	289	10	(1,2,3,6,9,18,36,72,144,145,289)
	265	10	(1,2,4,8,9,16,32,64,128,137,265)	290	10	(1,2,3,6,9,18,36,72,73,145,290)
	266	10	(1,2,3,5,8,16,32,64,69,133,266)	291	10	(1,2,3,6,9,18,36,72,144,147,291)
	267	11	(1,2,3,4,7,11,22,33,66,132,135,267)	292	10	(1,2,3,6,9,18,36,37,73,146,292)
	268	10	(1,2,3,4,8,16,32,35,67,134,268)	293	10	(1,2,4,5,9,18,36,72,144,149,293)
	269	11	(1,2,3,4,8,16,32,35,67,134,135,269)	294	10	(1,2,3,6,9,18,36,72,75,147,294)
	270	10	(1,2,3,5,10,15,30,45,90,135,270)	295	11	(1,2,3,4,7,9,18,36,72,144,151,295)
	271	11	(1,2,3,4,7,11,22,33,66,132,139,271)	296	10	(1,2,3,5,8,16,21,37,74,148,296)
	272	9	(1,2,4,8,9,17,34,68,136,272)	297	10	(1,2,3,6,9,18,36,72,144,153,297)
	273	10	(1,2,4,8,9,17,34,68,136,137,273)	298	10	(1,2,4,5,9,18,36,72,77,149,298)
	274	10	(1,2,4,8,9,17,34,68,69,137,274)	299	11	(1,2,3,4,7,11,18,36,72,144,155,299)
	275	11	(1,2,3,4,7,10,17,34,68,136,139,275)	300	10	(1,2,3,5,10,15,25,50,75,150,300)
301	301	11	(1,2,3,4,7,14,21,35,70,140,161,301)	326	10	(1,2,3,5,10,20,40,80,83,183,326)
	302	11	(1,2,3,4,7,9,18,36,72,79,151,302)	327	11	(1,2,3,4,7,10,20,40,80,160,167,327)
	303	11	(1,2,3,4,8,16,32,35,67,134,169,303)	328	10	(1,2,3,5,10,20,21,41,82,164,328)
	304	10	(1,2,3,4,8,11,19,38,76,152,304)	329	11	(1,2,3,4,8,16,32,64,67,131,198,329)
	305	11	(1,2,3,4,8,11,19,38,76,152,153,305)	330	10	(1,2,3,5,10,20,40,80,85,165,330)

306	10	(1,2,3,6,9,18,36,72,81,153,306)	331	11	(1,2,3,5,10,11,20,40,80,160,171,331)
307	11	(1,2,3,4,8,11,19,38,76,152,155,307)	332	10	(1,2,3,5,10,20,40,43,83,166,332)
308	10	(1,2,4,5,9,18,36,41,77,154,308)	333	11	(1,2,3,5,8,16,21,37,74,111,222,333)
309	11	(1,2,3,5,7,12,19,38,76,152,157,309)	334	11	(1,2,3,4,7,10,20,40,80,87,167,334)
310	11	(1,2,3,4,7,11,18,36,72,83,155,310)	335	11	(1,2,3,4,8,16,32,35,67,134,201,335)
311	11	(1,2,3,5,7,12,19,38,76,152,159,311)	336	10	(1,2,3,4,7,14,21,42,84,168,336)
312	10	(1,2,3,5,8,13,26,39,78,156,312)	337	11	(1,2,3,4,7,14,21,42,84,168,169,337)
313	11	(1,2,3,5,8,13,26,39,78,156,157,313)	338	11	(1,2,3,4,7,14,21,42,84,85,169,338)
314	11	(1,2,3,5,7,12,19,38,76,81,157,314)	339	11	(1,2,3,4,7,14,21,42,84,168,171,339)
315	11	(1,2,3,4,7,14,21,35,70,105,210,315)	340	10	(1,2,3,5,10,20,40,45,85,170,340)
316	11	(1,2,3,4,7,9,18,36,43,79,158,316)	341	11	(1,2,3,5,7,14,21,42,84,168,173,241)
317	11	(1,2,3,5,8,13,26,39,78,156,161,317)	342	11	(1,2,3,4,7,14,21,42,84,87,171,342)
318	11	(1,2,3,4,8,16,19,35,70,89,159,318)	343	11	(1,2,3,4,7,14,21,42,84,168,175,343)
319	11	(1,2,3,6,7,13,26,39,78,156,163,319)	344	10	(1,2,3,5,10,20,23,43,86,172,344)
320	9	(1,2,3,5,10,20,40,80,160,320)	345	11	(1,2,3,4,7,14,28,56,59,115,230,345)
321	10	(1,2,3,5,10,20,40,80,160,161,321)	346	11	(1,2,3,5,7,14,21,42,84,89,173,346)
322	10	(1,2,3,5,10,20,40,80,81,161,322)	347	11	(1,2,3,5,10,11,21,42,84,168,179,347)
323	10	(1,2,3,5,10,20,40,80,160,163,323)	348	11	(1,2,3,4,7,10,20,40,47,87,174,348)
324	10	(1,2,3,5,10,20,40,41,81,162,324)	349	11	(1,2,3,5,8,13,21,42,84,168,181,349)
325	10	(1,2,3,5,10,20,40,80,160,165,325)	350	11	(1,2,3,4,7,14,21,35,70,105,175,350)

List of Figures

1.1 Comparison between $l(n)$ and Schönhage's lower bound . . . 17
1.2 Evolution of addition chain methods 23
1.3 Continuing the powertree 34
1.4 The powertree up to level 7 35
1.5 The partially constructed Yacobi-compression tree 42
1.6 The Yacobi-compression tree for $e = 219$ 42

3.1 Configurations of $w \in N_q(n)$ starting with $\mathtt{0}$ 95
3.2 Configurations of $w \in N_q(n)$ not starting with $\mathtt{0}$ 95

4.1 The first 5 levels of the NAF-tree computation 119
4.2 The first 5 levels of the pruned NAF-tree computation 124
4.3 Graph of the function R for example 4.21(1) 148
4.4 Costs of Brauer method variants for $\lambda_2(e) = 192$ 149
4.5 Costs of Brauer method variants for $\lambda_2(e) = 512$ 149
4.6 Costs of Brauer method variants for $\lambda_2(e) = 1024$ 150
4.7 Graph of the function R for example 4.21(2) 151

List of Tables

1.1 Some numbers of different minimal chains (taken from [Thu93]) 18

2.2 Costs of the binary method 58
2.6 Exact costs of the m-ary method 64
2.7 Costs of the m-ary method 65
2.8 Costs of the Brauer method 67
2.9 Comparison of some suggested optimal values for d 69
2.10 Comparison of the numbers of operations for some values of d 70

3.1 Some examples of $l(e) > l(e + 1) + 1$ 74
3.6 Some numbers of sparse signed binary n-digit strings 93
3.7 The first 16 values of the sequence $s_2^{(-1)}(n)$ 101
3.8 The first NAF transformations for exact $n - 1$ digit numbers 103

4.5 Experimental and theoretical results for $Q(2)$ 137
4.6 Experimental and theoretical results for A 138
4.7 Final overview: costs of the binary expansion-based methods 164
4.8 Final overview: costs of the NAF-based Brauer methods . . . 165
4.9 Average cost comparisons derived in section 4.3 166

Bibliography

[AW93] Steven Arno and Ferrel S. Wheeler. Signed digit representation of minimal hamming weight. *IEEE Transactions on Computers*, 42(8):1007 – 1010, August 1993.

[BBB94] F. Bergeron, J. Berstel, and S. Brlek. Efficient computation of addition chains. *Journal de Théorie des Nombres de Bordeaux*, 6:21 – 38, 1994.

[BBBD89] F. Bergeron, J. Berstel, S. Brlek, and C. Duboc. Addition chains using continued fraction. *Journal of Algorithms*, 10:403 – 412, 1989.

[BC90] J. Bos and M. Coster. Addition chain heuristics. *Advances in Cryptology – Proceedings of Crypto'89, Lecture Notes in Computer Science*, 435:400 – 407, 1990. Springer-Verlag, New York.

[BCHM95] S. Brlek, P. Castéran, L. Habsieger, and R. Mallette. On-line evaluation of powers using Euclid's Algorithm. *Theoretical Informatics and Applications*, 29(5):431 – 450, 1995.

[BGMW92] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. *Advances in Cryptology – Proceedings of Eurocrypt '92*, 658:200–207, 1992. Springer-Verlag, Berlin/New York.

[BK95] Irina D. Bocharova and Boris D. Kudryashov. Fast exponentiation in cryptography. *Lecture notes in computer science: Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, 948:146–157, 1995.

[Boo51] Andrew D. Booth. A signed binary multiplication technique. *Quart. Journ. Mech. and Applied Math.*, IV(2):236 – 240, 1951.

[Bra39] Alfred Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45:736–739, 1939.

[BSMM95] N. Bronstein, I. A. Semendjajew, K. G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik.* Verlag Harri Deutsch, Thun und Frankfurt am Main, second edition, 1995.

[CL73] W. Edwin Clark and J. J. Liang. On arithmetic weight for a general radix representation of integers. *IEEE Transactions on Information Theory*, 19:823–826, November 1973.

[CT91] T. M. Cover and J. A. Thomas. *Elements of information theory.* Wiley, New York, 1991.

[Dat35] B. Datta. *History of Hindu Mathematics*, volume 2. Motilal Banarsi Das, Lahore 1935.

[DL80] D. Dobkin and R. J. Lipton. Addition chain methods for the evaluation of specific polynomial. *SIAM Journal on Computing*, 9(1):121 – 125, 2 1980.

[DLS81] P. Downey, B. Leong, and R. Sethi. Computing sequences with addition chains. *SIAM J. Comput.*, 10(3):638 – 646, 1981.

[DMS96] W. De Melo and B. F. Svaiter. The cost of computing integers. *Proceedings of the American Mathematical Society*, 124(5):1377 –1378, 1996.

[EgK90] Ö. Eğecioğlu and Çetin K. Koç. Fast modular exponentiation. *Communication, Control, and Signal Processing*, pages 188 – 194, 1990.

[Gat91a] Joachim von zur Gathen. Efficient and optimal exponentiation in finite fields. *Computational complexity*, 1:360 – 394, 1991. Birkhäuser Verlag, Basel.

[Gat91b] Joachim von zur Gathen. Efficient exponentiation in finite fields. *Proceedings of the 32nd IEEE Symposium on the Foundation of Computer Science*, pages 384–391, 1991.

[Gat99] Joachim von zur Gathen. Skript zur Vorlesung Kryptographische Protokolle. published through the internet, `http://www-math.upb.de/~aggathen/upb-only/skript-99ss-crypto2.ps`, Sommersemester 1999.

[GG99] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[GN97] Joachim von zur Gathen and Michael Nöcker. Exponentiation in finite fields: theory and practice. *Proc. 12th Symposium Applied Algebra, Algebraic Algorithms and Error-Correcting*

Codes, AAECC-12 in: Springer lecture notes in computer science, 1255:88–133, Toulouse, France, 1997.

[GN00] Joachim von zur Gathen and Michael Nöcker. Exponentiation using addition chains for finite fields. preliminary version, October 2000.

[Gor98] Daniel M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27:129 – 146, 1998. Article No. AL970913.

[Hen88] Kurt Hensel. Über die Darstellung der Zahlen eines Gattungsbereiches für einen beliebigen Primdivisor. *Journal für die Reine und Angewandte Mathematik*, 103:230–237, 1888.

[Heu94] H. Heuser. *Lehrbuch der Analysis*, volume 1 of *Mathematische Leitfäden*. B. G. Teubner Stuttgart, 1994.

[HL94] L. C. K. Hui and K.-Y. Lam. Fast square-and-multiply exponentiation for RSA. *Electronics Letters*, 30(17):1396 – 1397, August 1994.

[JM89] J. Jedwab and C. J. Mitchell. Minimum weight modified signed-digit representations and fast exponentiation. *Electronic Letters*, 25(17):1171–1172, 1989.

[JS72] F. Jelinek and K. S. Schneider. On variable-length-to-block coding. *The structural and distance properties of punctured convolutional codes, IEEE Trans. Inform. Theory*, IT-18(6), November 1972.

[Jun93] D. Jungnickel. *Finite Fields: Structure and Arithmetics.* BI Wissenschaftsverlag, Mannheim, 1993.

[Knu62] Donald E. Knuth. Evaluation of polynomials by computer. *Communications of the ACM*, 5(1):595 – 599, January 1962.

[Knu97] Donald E. Knuth. *The art of computer programming*, volume 2 / Seminumerical Algorithms. Addison-Wesley, third edition, Reading MA 1997.

[Koç91] Çetin K. Koç. High-radix and bit recoding techniques for modular exponentiation. *Intern. J. Computer Math.*, 40:139 – 156, 1991.

[Koç95] Çetin K. Koç. Analysis of sliding window techniques for exponentiation. *Computers and Mathematics with Applications*, 30(10):17 – 24, 1995.

[KT92] K. Koyama and Y. Tsuruoka. Speeding up elliptic cryptosystems by using a signed binary window method. *Advances in Cryptography – CRYPTO '92, 12th Annual International Cryptology Conference Santa Barbara, USA*, pages 345 – 358, August 1992.

[KY98] Noboru Kunihiro and Hirosuke Yamamoto. Window and extended window methods for addition chain and addition-subtraction chain. *IEICE Trans. Fundamentals*, E81-A(1):72–81, January 1998.

[LD76] R. J. Lipton and D. Dobkin. Complexity measures and hierarchies for the evaluation of integers and polynomials. *Theoretical Computer Science*, 3:349 – 357, 1976.

[Len93] H. W. Lenstra. Generating units modulo an odd integer by addition and subtraction. *ACTA ARITHMETICA*, LXIV(4):383 – 388, 1993.

[Lin98] J. H. van Lint. *Introduction to Coding Theory.* Graduate Texts in Mathematics. Springer-Verlag, Berlin, Heidelberg, New York, third edition, 1998.

[MO90] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *Theoretical Informatics and Applications*, 24(6):531–544, 1990.

[Nöc01] Michael Nöcker. *Data structures for parallel exponentiation.* Dissertation, Universität-Gesamthochschule Paderborn, Paderborn, Germany, May 2001.

[Oli81] J. Olivos. On vectorial addition chains. *Journal of Algorithms*, 2:13 – 21, 1981.

[Par90] Behrooz Parhami. Generalized signed-digit number systems: A unifying framework for redundant number representations. *IEEE Transactions on Computers*, 39(1):89–98, January 1990.

[PPC99] H. Park, K. Park, and Y. Cho. Analysis of the variable length nonzero window method for exponentiation. *Computers and Mathematics with Applications*, 37: 21–29, 1999.

[Sac79] E. Sachau. al-bīrūnī's chronology of ancient nations, London 1879.

[Sai75] A. S. Saidan. *The Arithmetic of al-Uqlīdisī.* D. Reidel, Dordrecht 1975.

[Sau92] J. Sauerbrey. Resource requirements for the application of addition chains in modulo exponentiation. *EUROCRYPT '92*, pages 174 – 182, 1992.

[Sch75] A. Schönhage. A lower bound for the length of addition chains. *Theoretical Computer Science*, 1:1 – 12, 1975.

[Sem83] I. Semba. Systematic method for determining the number of multiplications required to computer x^m, where m is a positive integer. *Journal of Information Proceeding*, 6(1), 1983.

[Sma00] N. P. Smart. A comparison of different finite fields for use in elliptic curve cryptosystems. University of Bristol, Computer Science Department, June 2000.

[Sti95] Douglas R. Stinson. *Cryptography – theory and practice*. Discrete Mathematics and Its Applications. CRC Press, Boca Raton, London, New York, Washington, D.C., 1995.

[Str30] W. W. Struve. *Quellen und Studien zur Geschichte der Mathematik*. **A1**, 1930.

[Sub89] M. V. Subbarao. Addition chains – some results and problems. *Number Theory and Applications*, pages 555 – 574, 1989.

[Thu76] Edward G. Thurber. Addition chains and solutions of $l(2n) = l(n)$ and $l(2^n - 1) = n + l(n) - 1$. *Discrete Mathematics*, 16:279 – 289, 1976.

[Thu93] Edward G. Thurber. Addition chains – an erratic sequence. *Discrete Math*, 122(1-3):287–305, 1993.

[Thu99] Edward G. Thurber. Efficient generation of minimal length addition chains. *SIAM J. Comput.*, 28(4):1247–1263, March 1999.

[Tun68] Brian P. Tunstall. *Synthesis of noiseless compression codes*. Ph.D. dissertation, Georgia Inst. Technol., Atlanta, 1968.

[Yac90] Y. Yacobi. Exponentiating faster with addition chains. *Proceedings of EUROCRYPT'90*, 1990.

[Yac98] Y. Yacobi. Fast exponentiation using data compression. *SIAM Journal on Computing*, 28(2):700 – 703, 1998.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable rate coding. *IEEE Trans. Inform. Theory*, 24(5), Sep. 1978.

Index

$\lceil\ \rceil$ upper Gaussian brackets, 16
$\lfloor\ \rfloor$ lower Gaussian brackets, 16
$\bar{1}$ denoting -1, 76
$\oplus$ concatenation of addition chains, 31
$\otimes$ concatenation of addition chains, 31
( )
  $(e)_B$ partitioning of e according to Bocharova's and Kudryashov's method, 46
  $(e)_Y$ partitioning of e according to Yacobi's method, 42
  $(e)_{\bar{q}}$ signed q-ary digit representation of e, 76
  $(n)_b$ the b-adic expansion of n, 16

A
  A intermediate value in the proof of theorem 3.18, 98
  $A_{\chi(e)}$ number of addition steps, 52
  NAF transformation by Arno and Wheeler, 86

a
  a_i element of the semantics of an addition chain, 12, 19, 72
  a one of the Weierstrass coefficients of an elliptic curve, 7
  addition chain, 12
  addition-subtraction chain, 71
  admissible, 109
  arithmetic weight, 76
  average case, 56
    generalized, **152**, 151–162
  average case analysis, 56
  average Hamming weight of the binary NAF, 107

B
  B intermediate value in the proof of theorem 3.18, 99
  BGMW method, 35
  Bocharova's data compression method, 43
  Brauer method, 66–70
    cost analysis, 59, 67
    NAF-based Brauer method, *see* NAF-based Brauer method

b
  b one of the Weierstrass coefficients of an elliptic curve, 7
  b-adic expansion, 16
  binary method, 56–58
    cost analysis, 57
    cost comparison, *see* cost comparison
    examples, 23
    general idea, 22
    history, 22, 57
    the NAF-based binary method, *see* NAF-based binary method

C
  C loop condition in a formal verification, 86

c
  $\chi(\vec{e})$ vectorial addition chain, 19

$\chi(e)$ an addition-chain of e, 12
$\chi(m)$ addition chain for the window length m-step, 63
$\overline{\chi}(e)$ addition-subtraction chain for integer e, 71
$c(\bar{\chi})$ costs of an addition chain χ, 73
$c(\chi)$ costs of an addition chain χ, 54
c_1 costs of the traditional Brauer precomputation step, 142, 166
c_2 costs of the NAF-based Brauer precomputation, 142, 166
c_{2a} costs of the possible squarings, 142, 166
c_{2b} costs of the possible additions, 142, 166
$c\bar{\gamma}(n)$ costs of the precomputation addition chain $\bar{\gamma}(n)$, 121
$c\gamma(m)$ costs of the precomputation addition chain $\gamma(m)$, 61
c(A) cost of an addition step, 54
c(I) cost of an inversion, 54
c(Q(q)) cost of a q-step, 54
c(Q) used for c(Q(2)), 54
complete, 45
concatenation of addition chains, 31
continued fractions method, 29
cost comparisons, 139–151
  binary method average, 140
  binary method exact, 139
  Brauer method average, 144
  Brauer method exact, 142
cost measures, 54
costs of an m-step, 63
costs of precomputation, 60

D

D a typical set, 44
D some subset of the set of possible results of a probabilistic trial, 44

d

d-NAF the d digit partitioned NAF, 117
d order of a point on an elliptic curve, 8
d window width for the Brauer method, 66
data compression methods, 40, 43

E

E set of numbers that are computed in an addition chain, 12, 72
E digit of an m-adic expansion, 59
E elliptic curve, **7**, 8
  addition on E, 7
  group structure, 8
  order of a point, 8
ElGamal cryptosystem, 8
Euclid's method, 29

e

ϵ carry bit, 86
ε the empty word in a formal language, 106
e general exponent, 11
elliptic curve, *see* E elliptic curve
expected length of the NAF, 102
exponentiation problem, 11

F

F probability distribution, 44
F_n the n-th Fibonacci number, 105
$\mathbb{F}_q = \mathbb{Z}/q\mathbb{Z}$, 8
F field underlying an elliptic curve, 7
the Frobenius automorphism, 48

f

$f_{\mathcal{L}}(t,x)$ generating function, 106
factor method, 27

G

$\Gamma(m)$ set of precomputation addition chains, 61
$\bar{\Gamma}(n)$ costs of the precomputation addition-subtraction chain $\bar{\gamma}(n)$, 120
Gaussian brackets, 16

g

$\bar{\gamma}(n)$ addition-subtraction chain from $\bar{\Gamma}(n)$, 121
$\gamma(m)$ addition chain from $\Gamma(m)$, 61
general algorithm, 50, 75

H

H an arbitrary semigroup, 11
Hamming weight
- average Hamming weight of the binary NAF, 107
- for a signed q-ary digit representation, 76
- for b-adic expansions, 16

h

hidden doubling, 15

I

$I_{\bar{\chi}(e)}$ number of inversions in addition-subtraction chain $\bar{\chi}(e)$ for e, 73
IV, IV_1, IV_2 loop invariants in a formal verification, 86

i

i.i.d. - independent identically distributed (uniformly), 56

J

NAF transformation by Jedwab and Mitchell, 83

j

$j'(i)$ first predecessor in an addition-subtraction chain tuple, 71
$j(i)$ first predecessor in an addition chain tuple, 12, 19

K

κ_i order of the i-th prime factor in a prime factorization of an integer, 27
K size of optimal Tunstall code set, 45

k

$k'(i)$ second predecessor in an addition-subtraction chain tuple, 71
$k(i)$ second predecessor in an addition chain tuple, 12, 19

L

$L(\bar{\chi})$ length of an addition-subtraction chain, 72
$L(\chi)$ length of an addition chain, 12, 19
L_i length of the i-th window plus leading zeros in a Yacobi compression partitioning, 41
λ
- λ_+ abbreviating parts of the explicit formula (3.9), 94
- λ_- abbreviating parts of the explicit formula (3.9), 94
- $\lambda_Y(e)$ the length of the Yacobi compression partitioning of an integer e, 41
- $\lambda_b(e)$ length of the b-adic expansion of e, 16
- $\lambda_{\bar{q}}(e)$ length of a signed q-ary digit representation, 76

$\lambda_B(e)$ the length of the Bocharova-partition of the binary expansion of e, 46
$\mathcal{L}$ denoting a specific formal language, 106
Lambert's W function, 69

l

$l(e)$ minimal length of an addition chain for e, 17
length of the NAF, 91

M

M_i multiplication with value i, 24
M a message, 9
NAF transformation by Morain and Olivos, 80

m

m-ary method, 58–70
  Brauer method, *see* Brauer method
  costs of an m-step, 63
  costs of the exponentiation step, 62
  examples, 24
  general idea, 24
  NAF-based Brauer method, *see* NAF-based Brauer method
m defined as the digit q-1, 98

N

$\mathcal{NAF}_q$ a general NAF, 108
$N_q(n)$ the **set** of all sparse signed q-ary digit strings with at most n digits, **92**, 94
$N_q^{(-1)}(n)$ the **set** of all sparse signed q-ary digit strings with exactly n digits whose binary expansion needs one digit less, 92
NAF the non-adjacent form, 78, 80
NAF-based binary method, 112–117
  cost overview, 116
  number of inversions, 115
  number of multiplications, 113
  number of squarings, 112
NAF-based Brauer method, 117–138
  cost overview, 135
  costs of precomputation, 121
  inversions in the exponentiation step, 134
  multiplications in the exponentiation step, 129
  squarings in the exponentiation step, 127
NMC number of minimal chains, 18

n

$\hat{n}(d)$ variable, **121**, 130
$\nu_b(e)$ Hamming weight of the b-adic expansion of e, **16**, 17
$\nu_{\bar{2}}(e)$ Hamming weight for a signed binary digit representation of e, **76**, 107
normal basis, 49
number of q-steps, 52
number of additions, 52
number of operations, 73
number of sparse signed q-ary digit strings, 94
number of sparse unsigned binary digit strings, 105

O

Ω set of possible (elementary) results of a probabilistic trial, 44
$\Omega(n)$ set of possible results of a probabilistic trial, 152
Ω_n the set of all integers with exact n bit long binary expansion, 56, 152
$\mathcal{O}$ point at infinity of an elliptic curve, 7

o

occurence of the letter 0 in the NAF, 106
operation dependent cost measure, 73
optimal addition chains, 169

P

P_x, P_y coordinates of the point P, 7
Π an exponentiation problem, 11
P point on an elliptic curve, 7, 11

p

p_A Alice's public key, 8
powertree method, 34
probability distribution of $\nu_{\bar{q}}(e)$, 108
proper, 45

Q

Q_x, Q_y coordinates of the point Q, 7
$Q_{\chi(e)}$ has been defined to be equal to $Q_{\chi(e)}(2)$, 52
$Q_{\chi(e)}(q)$ number of q-steps, 52
Q point on an elliptic curve, 7

q

q-addition chain, 20
q-steps, 52
q a large prime, 8

R

R result of a cost comparison, 139, 140, 143, 144, 160, 166

r

ρ_i prime factor in the prime factorization of an integer, 27
replacing doublings by additions, 13

S

$S(\bar{\chi})$ the semantics of an addition-subtraction chains, 72
$S(\chi)$ the semantics of an addition chains, 12, 19
$\mathcal{S}$
$s_q(n)$ the **number** of all sparse signed q-ary digit strings with at most n digits, 94, 102
$s_q^{+}(n)$ the **number** of non-adjacent unsigned binary digit strings, 105
S^{m} exponentiation with m, 24

s

σ total number of operations (Q+A), 68
$s_q(n)$ the **number** of all sparse signed q-ary digit strings with at most n digits, **92**
$s_q^{(-1)}(n)$ the number of sparse signed q-ary digit strings with exactly n digits whose binary expansion needs one digit less, 93
s_A Alice's private key, 8
s_B Bob's private key, 9
$\mathsf{sgn}(\mathrm{x})$ denoting the signum function, 86
signed digit representation, 76
sliding window methods, 25

T

T_i the tail of a window, 130
Tunstall code, 45

t

t_i the length of a tail T_i of a window, 130

u

u_i elements in a continued fractions vector, 30

V

$Val(N_q(n))$ the set of the integer values of all elements of $N_q(n)$, 92, 94
$Val(w)$ the integer e whose NAF is w, 92

v

vectorial addition chain, **19**, 30

W

W_i window in a Yacobi compression partitioning, 41
W_i window of a Brauer partition, 130
W_i' window of a d-NAF partition, 130
Lambert's W function, 69

w

w word in a formal language, 106
$w_q(e)$ the arithmetic weight of a number e, 76

x

x general base of an exponentiation, 11

Y

Yacobi's data compression method, 40

Z

$\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z}$, 7
$\mathcal{Z}$ intermediate value in proof of theorem 3.18(2), 96

z

ζ_n total number of occurrences of the letter 0 over all NAF strings of length n, 106
z_i number of zeros between the i-th and the (i-1)st window in a Yacobi or Bocharova partitioning of the input's binary expansion, 41