On the complexity of some computational problems in the Turing model

Claus Diem

November 18, 2013

Abstract

Algorithms for concrete problems are usually described and analyzed in some random access machine model. This is in particular the case in the areas such as computational algebra, algorithmic number and cryptology. In this work the complexity of various computational problems is studied in the multitape Turing model of computation. It is shown that "up to logarithmic factors" key results also hold in the multitape Turing model. Specific problems which are considered are linear algebra computations, factorization of integers and the computation of discrete logarithms in various classes of groups.

1 Introduction

The area of computational complexity falls roughly into two parts: On the one hand we have "abstract" complexity theory and on the other hand, we have the study of algorithms for particular computational tasks; cf. [San12]. In the first part, exponents in running times are not considered and therefore it does not matter if an algorithm is analyzed on the basis of a particular RAM model with logarithmic cost function (and with a reasonable set of instructions) or in the Turing model of computation. On the other hand, if one studies algorithms for particular computational tasks, often the exponents are of crucial interest and consequently it is important that it is clearly stated what the underlying machine model is.

This work is concerned with the algorithmic side of computational complexity. Here, most of the time one chooses a particular random access machine (RAM) model as a formal basis for the analysis of an algorithm. In particular, it is generally postulated that almost immediate transfer of data from whatever location of the memory to a central processor is possible. For example, two standard textbooks on algorithms, [AHU74] and [CLRS01], are written from this point of view.

In [Die11d] we showed that any computation in a RAM model with "reasonable" commands and time and space measures can be simulated quasi-optimally (optimally up to logarithmic factors) by the successor RAM model with logarithmic cost function, a RAM model of extreme simplicity.

It is however the *multitape Turing model* which can be considered to be the most basic reasonable model for complexity theoretic investigations, and for various reasons it is of interest to ask for the complexity of computational problems in this model.

- From a mathematical point of view this is motivated by the simplicity of the model.

- Asymptotic complexity theory is motivated by physical intuition along the following thought experiment: How many resources (in terms of time and space) would one in principle need to perform computations for longer and longer inputs, assuming that large enough computers *could* be built? Of course, for a reasonable thought experiment physical laws should be kept in mind as much as possible. And this means that time for transport of data is bounded by the distance divided by the speed of light.

  We expand a bit on this thought experiment: The idea of computers operating on unboundedly long input sequences and for unboundedly long times and maybe also with an unboundedly large storage itself contradicts basic physical laws. Does this mean that because the idea is anyway unrealistic, it does not matter if we ignore a further physical law (in this case the fact that light does not travel instantaneously)? Our answer to this question is that it is still reasonable to obey physical laws as much as possible.

  We also mention another possible counter argument: One might argue that the observation on the speed of light is not relevant from a practical point of view. Here we answer that complexity theory as an area of mathematics is not concerned with practical computations.

- Furthermore, results in RAM models might indeed underestimate the resources needed *from a practical point of view* given today's technology. Indeed, for algorithms employing very large amounts of data, it is unrealistic to assume that all this data can be stored in the random access memory of the computer. So some *external storage* is needed. By the very intuition underlying the multitape Turing model (namely that one prints and ready symbols on tapes), the model can be considered to be a basic model of computers with external storage.

Clearly, any multitape Turing machine can be simulated quasi-optimally by a successor RAM with logarithmic cost function. However, no general quasi-optimal result on the simulation of successor RAMs by multitape Turing machines is known, and indeed it seems to be a difficult problem to obtain such a result.

In this work we show that nonetheless up to logarithmic factors, many complexity theoretic results arising in computational algebra, algorithmic number theory and cryptology can also be obtained in the multitape Turing model. Concretely, we are mostly interested in the time complexity or running time of multitape Turing machines for specific problems. As a second parameter we always also mention the space requirements. Our approach is here that the focus is always on the best result concerning running time (up to multiplicative constants if we employ the O-notation or up to logarithmic factors if we employ the $\tilde{O}$-notation), and the space requirements are treated as a second optimizing parameter. All results we state are proven; so-called heuristic results are not considered in this work.

The specific computational problems we consider are as follows: sorting, multiplication of square matrices and further problems from linear algebra, sparse linear algebra, factorization, and the computation of discrete logarithms in various classes of groups. Sorting can be performed efficiently with a multitape Turing machine via Merge Sort, and most other Turing algorithms we consider rely on an efficient solution of the sorting problem.

The choice of problems itself is motivated by cryptanalytic applications, in particular concerning public-key cryptography. But even apart from interest in the problems themselves, we think that it is worthwhile to consider the proposed Turing algorithms because of the general strategies employed: Both Merge Sort and the Turing algorithm for multiplication of matrices are based on recursion with a divide and conquer strategy. Furthermore, the currently fastest (randomized) RAM algorithms for the computation of discrete logarithms in the degree 0 class groups (Picard groups) of curves of fixed genus employ an extensive tree (or graph) which is constantly being modified. By modifying the tree in "stages", one can again in an efficient way use sorting. We are confident that for other computational problems these and similar techniques lead to quasi-optimal simulations of Random Access Machines by Turing machines too.

As already mentioned, we reduce many problems to sorting problems, and we use Merge Sort for efficient sorting. It is difficult to find the key observation that sorting can be performed efficiently on multitape Turing machines via Merge Sort in the literature. The algorithm does however appear as a subroutine of another algorithm in the foundational book [SGV94] by Schönhage et al. The algorithm (in subsection 6.5.2) is however quite well hidden – it appears neither in the table of contents nor in the index. It is the hope of the author that this work helps to publicize the use of this algorithm and the divide and conquer paradigm for Turing machines.

Even though it is hard to find any mentioning of the Merge Sort algorithm and more generally the divide and conquer paradigm in the context of Turing machines, these ideas are fundamental in works on external disk storage, as for example [Vet98], [Vet01].

Personally, I learned about the efficiency of Merge Sort on Turing machines via a conversation with Pierrick Gaudry. I heartily thank him for this conversation without which this work would not have been possible.

2 Terminology and basic definitions

In this work, the natural numbers are $\mathbb{N} := \{0, 1, 2 \ldots\}$. Moreover, the length of a bitstring s is denoted $|s|$. We use the *logarithmic function* $l : \mathbb{N} \longrightarrow \mathbb{R}_{\geq 0}$ from [AHU74], which is defined via $l(n) := 1$ if $n = 0$ and $l(n) := \lfloor \log_2(n) \rfloor + 1$ otherwise.

In the following, by a *Turing machine* we mean a deterministic multitape Turing machine with (read only) input tape and (write only) output tape on the alphabet $\{0, 1\}$. By a *randomized Turing machine* we mean a non-deterministic multitape Turing machine with input tape and output tape on the alphabet $\{0, 1\}$ such that in each branching there are two possibilities. In the execution of the machines, for each branching, each branch in chosen uniformly randomly and independently of the other branchings.

Given some Turing machine M and a bitstring x, we have the running time and the space requirements of the operation of M on x. Note here that input and output tape are discarded for the space requirements. For varying x, the running time and the space requirements are then functions $\{0, 1\}^* \longrightarrow \mathbb{R}_{\geq 0} \,\dot{\cup}\, \{\infty\}$. We stress this because in complexity theory the running time and the space requirements (the space complexity) are usually defined in terms of the input length and not in terms of the input (cf. [Pap94]). The standard definitions are however inappropriate for the present work: We would not be able to state the theorems with the standard definition.

If now a randomized Turing machine M and a bitstring x is given, the whole computation is a random process and the running time is a random variable. By taking expected values, we obtain the expected running time of the application of M to x. For varying x, we then again have a function $\{0, 1\}^* \longrightarrow \mathbb{R}_{\geq 0} \,\dot{\cup}\, \{\infty\}$.

We give all results concerning deterministic Turing machines in the following form: We state that there exists some Turing machine "with the following specification". Then we state *input*, *attributes*, *output*, *running time* and *space requirements*. We explain now what it means that a Turing machine follows such a specification.

As usual, we describe the input to the machine on an abstract "high level" without indicating how exactly the concerning mathematical objects are to be represented via bitstrings.

Let us, in order to express the following thoughts accurately, use the following terminology: We have the *abstract input instances of the computational problem*, which are mathematical objects we use for the high-level description. These objects form a class. Given such a class, we fix a representation of the objects in the class by bitstrings (each object is then represented by at least one bitstring). Except for the algorithms for the discrete logarithm problem in the degree 0 class groups of curves, there are always obvious ways to represent the abstract input instances (even though – as usual – we do not fix all the details). For computations in the degree 0 class groups of curves, we follow [Heß01] and [Die11b]. We assume for the following that a representation of the objects by bit strings has been fixed.

In the specification, we call the abstract input instances simply *input*. Now, in this work, by an *attribute* we mean an assignment from the class of abstract input instances to the real numbers. We use the attributes in order to reason about the output, the running time and the space requirements.

If applied to a bitstring representing some abstract input instance (with respect to the fixed representation), the machine outputs a bitstring representing the object given under *output*. We make no statement concerning the behavior of the machines if they are applied to other bitstrings.

Many results depend on arithmetic in finite fields. There exists a Turing machine which performs basic arithmetic (i.e. addition, subtraction, multiplication, division) in finite fields in a time of $O(l(q) \cdot (l(l(q)))^2 \cdot l(l(l(q))))$, where q is the field size. One can obtain such a machine by combining the Schönhage-Strassen algorithm ([SS71]) with a fast Euclidean algorithm for integers and then with a fast algorithm for polynomial multiplication and finally a fast Euclidean algorithm for polynomials (see [GG03, Corollary 11.8, Corollary 11.10]). We note that the Schönhage-Strassen algorithm for integer multiplication is now superseded by Fürer's algorithm ([Für07]), but still it is currently not known whether there exists a Turing machine which can perform arithmetic in all finite fields is a time of $o(l(q) \cdot (l(l(q)))^2 \cdot l(l(l(q))))$.

Inspired by this, we define the function $A : \mathbb{N} \longrightarrow \mathbb{R},\ n \mapsto l(n) \cdot (l(l(n)))^2 \cdot l(l(l(n)))$. Note that $A(n^2) \in O(A(n))$ for $n \longrightarrow \infty$. The function A can be substituted by any function on $\mathbb{N}$ for which there exists a Turing machine which performs arithmetic in finite fields in time $O(A(q))$ and for which $A(q^2) \in O(A(q))$ holds.

Whereas the term "Turing machine" has a precise meaning, we use the term “algorithm” freely and in an informal way. In the same manner, the phrase “Turing algorithm” is also used informally as a description of a Turing machine in the form of an algorithm. Consequently, in the following we never argue about the running time or space requirements of an “algorithm”.

3 Sorting

We begin with the key problem of sorting a list of natural numbers. Let us recall the idea of the Merge Sort algorithm. We give a recursive form of the algorithm.

For this, we fix the following notation: Given a list L, $\mathsf{L}(L)$ is the list of the first $\lceil \frac{\#L}{2} \rceil$ entries of L and $\mathsf{R}(L)$ is the list of the remaining entries.

Furthermore, let Sort be an algorithm which merges two sorted lists into one sorted list. We then have:

MergeSort

Input: A list L of natural numbers
Output: An ascendingly sorted list obtained by permutation from L.

```
Select
    • If $L$ has no entries, then output the empty list.
    • If $L$ has one entry, then output $L$.
    • If $L$ has at least two entries, then output
      Sort(MergeSort(L($L$)),MergeSort(R($L$))).
```

We wish to adapt the algorithm to the Turing model. Here the following aspects should be kept in mind: One can perform the algorithm without copying the lists upon application of MergeSort. Moreover, if two tape heads point to two sorted lists L_1, L_2 on two different tapes, one can go along the two lists to obtain a sorted list on a third tape. With these considerations, one easily obtains:

Theorem 1 *There exists a Turing machine with the following specification:*

- *Input: A list L of natural numbers*
- *Attributes:*
  - *n, the size of L*
  - *m, the largest number occurring in L*

- *Output: An ascendingly sorted list which is obtained by permuting L*
- *Running time: $O(n \cdot l(n) \cdot l(m))$*
- *Space requirements: $O(|\mathbf{input}|) \subseteq O(n \cdot l(m))$*

We now consider lists of tuples in $\mathbb{N} \times \{0,1\}^*$. We interpret the first component of a tuple in the list as a label, and we sort according to the label. By exactly the same technique as in the algorithm for the previous theorem, we obtain the following result:

Theorem 2 *There exists a Turing machine with the following specification:*

- *Input: A list L of tuples in $\mathbb{N} \times \{0,1\}^*$*
- *Attributes:*
  - *n, the size of L*
  - *m, the largest first component ("label") of all tuples in L*
  - *B, the length of the largest second component of all tuples in L*
- *Output: A list which is obtained by permuting L and is sorted ascendingly with respect to the first components*
- *Running time: $O(n \cdot l(n) \cdot (l(m) + B))$*
- *Space requirements: $O(|\mathbf{input}|) \subseteq O(n \cdot (l(m) + B))$*

4 Dense linear algebra

In this section, we consider algorithms for operations with matrices in dense representation over finite fields. We recall that dense representation of vectors means that a vector of length n is stored in a array of length n, with one field for each entry. Furthermore, a matrix of size $m \times n$ is represented by n consecutive vectors of length m.

It is common to study the complexities of computational problems associated with matrix computations in non-uniform algebraic models. In particular, this approach is taken in [BCS91]. In this monograph, first an algebraic complexity theory is developed, and then this theory is applied to various computational problems. Here, we wish to transfer basic results from [BCS91] from non-uniform algebraic models to the Turing model. As already mentioned, in contrast to [BCS91], we restrict ourselves hereby to computations over finite fields.

The central question addressed in [BCS91] is arguably the complexity of matrix multiplication, and a central object of interest is the *exponent of* *matrix multiplication* ω. Let us briefly recall some concepts from [BCS91], following the same notation:

Let us first fix any field k. (In contrast to [BCS91, Chapter 16] we make no assumption on k.)

Let $M_k(m)$ be the minimum of the total complexities of a straight-line programs (without divisions) for multiplication of two matrices in $k^{m\times m}$ (cf. [BCS91, subsection 15.1 and (4.7)]).

Then the *exponent of matrix multiplication* is defined as

$$\omega_k := \inf\{\tau \in \mathbb{R} \mid M_k(h) = O(h^\tau) \text{ for } h \longrightarrow \infty\} \ .$$

This exponent of matrix multiplication is of greatest importance because many other computational problems associated to matrices reduce in an efficient way to multiplication of matrices.

For these other computational problems, one changes the model of computations and allows divisions. Because of necessary case distinctions concerning division by zero, one now regards computation trees instead of straight line programs.

In [BCS91, Chapter 16] the following problems associated to square matrices over the fixed field k are considered: Computation of the LUP-decomposition (if it exists), computation of the inverse (if it exists), computation of the determinant, computation of the characteristic polynomial, computation of the reduced row echelon form, computation of the kernel. For all these problems, it is shown that if one considers the problems for square matrices and defines an exponent in a similar way as for matrix multiplication, then this exponent is $\leq \omega_k$. For all the indicated problems except for the computation of the reduced row echelon form it is shown that if the field k is infinite, then the exponent of the corresponding problem is equal to ω_k.

Our goal is now to use these results to obtain upper bounds on the complexity of the corresponding problems for matrices over finite fields in the Turing model of computation.

We are first concerned with the central problem of matrix multiplication. Let us recall the notion of *bilinear complexity* of matrix multiplication ([BCS91, Section 14.1]).

Let $e, h, \ell \in \mathbb{N}_{\geq 1}$. Let $r \in \mathbb{N}$, let $f_1, \ldots, f_r \in k[X_{1,1}, \ldots, X_{e,h}], g_1, \ldots, g_r \in k[Y_{1,1}, \ldots, Y_{h,\ell}]$ be linear forms, and let $C_1, \ldots, C_r \in k^{e\times\ell}$. If now

$$M \cdot N = \sum_{\rho=1}^{r} f_\rho(((m_{i,j}))_{i,j})\, g_\rho(((n_{i,j}))_{i,j})\, C_\rho$$

for all $M \in k^{e\times h}, N \in k^{h\times\ell}$, $(f_1, g_1, C_1; \ldots; f_r, g_r, C_r)$ is called a *bilinear computation* of length r of the matrix multiplication $k^{e\times h} \times k^{h\times\ell} \longrightarrow k^{e\times\ell}$.

The minimal length of such a computation is called the *bilinear complexity* of matrix multiplication and is denoted $R(\langle e,h,\ell\rangle)$ in [BCS91]. ($\langle e,h,\ell\rangle$ denotes the tensor defining the matrix multiplication and $R(\langle \ell,h,\ell\rangle)$ is also the rank of this tensor.) Let us – in order to emphasize the dependence on k – denote the tensor under consideration by $\langle e,h,\ell\rangle_k$ and consequently the bilinear complexity for the matrix multiplication under consideration by $R(\langle e,h,\ell\rangle_k)$.

By [BCS91, Proposition 15.1] we have

$$\omega_k = \inf\{\tau \in \mathbb{R} \mid R(\langle h,h,h\rangle_k) = O(h^\tau) \text{ for } h \longrightarrow \infty\} \ .$$

The proof in fact also shows that

$$\omega_k = \inf_{h\in\mathbb{N}_{\geq 1}} \log_h(R(\langle h,h,h\rangle_k)) \ .$$

The proof relies on a divide and conquer strategy which can also be successfully applied in the Turing model. Because of this, we recall the key ingredients of the proof here.

Let θ_1 be the first right-hand side and θ_2 the second one.

One can show that $\frac{1}{2}R(\langle h,h,h\rangle_k) \leq M_k(h)$. This gives immediately that $\theta_1 \leq \omega_k$. Moreover, $\theta_2 \leq \theta_1$. Indeed, let $\epsilon > 0$. Then by definition of θ_1 there exists a $C > 0$ such that $R(\langle h,h,h\rangle_k) \leq C \cdot h^{\theta_1+\epsilon}$ for h large enough. Let us fix such a C. Then we have $\log_h(R(\langle h,h,h\rangle_k) \leq \log_h(C) + \theta_1 + \epsilon$ for h large enough. Moreover, for h large enough we have $\log_h(C) \leq \epsilon$. Thus there exists some $h \in \mathbb{N}$ with $\log_h(R(\langle h,h,h\rangle_k) \leq \theta_1 + 2\epsilon$. Thus $\theta_2 \leq \theta_1 + 2\epsilon$. This gives the result.

It remains to show that $\omega_k \leq \theta_2$. Let for this again $\epsilon > 0$, and let $m \in \mathbb{N}$ with $r := R(\langle m,m,m\rangle_k) \leq m^{\theta_2+\epsilon}$. Let $(f_1, g_1, C_1; \cdots ; f_r, g_r, C_r)$ be a bilinear computation of the multiplication of $m \times m$-matrices over k. We therefore have

$$M \cdot N = \sum_{\rho=1}^{r} f_\rho(((m_{i,j}))_{i,j})\, g_\rho(((n_{i,j}))_{i,j})\, C_\rho$$

for all $M, N \in k^{m\times m}$. If now A is any (not necessarily commutative) k-algebra, the above equality still holds in $A^{m\times m}$. One applies this with $A = k^{m^{i-1}\times m^{i-1}}$ for $i \in \mathbb{N}_{\geq 1}$. Block decomposition of matrices gives an isomorphism $(k^{m^{i-1}\times m^{i-1}})^{m\times m} \simeq k^{m^i\times m^i}$ as k-algebras. One obtains that there exists some $C > 0$ such that

$$M_k(m^i) \leq r \cdot M_k(m^{i-1}) + C \cdot m^{2i} \tag{1}$$

for all $i \geq 1$. Together with the fact that $r \geq m^2$ (see [BCS91, Remark 14.28]), it follows that

$$M_k(m^i) \leq C \cdot \sum_{j=0}^{i} r^j \cdot (m^2)^{i-j} \leq C \cdot i \cdot r^i \leq C \cdot i \cdot m^{(\theta+\epsilon)\cdot i} \ .$$

The result then follows easily.

By these considerations, we obtain for every fixed bilinear computation $(f_1, g_1, C_1; \cdots ; f_r, g_r, C_r)$ of the multiplication of $m \times m$-matrices over k the following divide and conquer algorithm.

FastMatrixMultiplication

```
Input: Square matrices $M, N$ of the same size $m^i \times m^i$ for some $i \in \mathbb{N}$ over
the fixed field $k$
Output: $MN$.

If $i = 0$ then output $MN$.
Otherwise
    Divide $M$ and $N$ respectively into blocks of size $m^{i-1} \times m^{i-1}$;
      let $M_{i,j}, N_{i,j} \in k^{m^{i-1} \times m^{i-1}}$ be these blocks.
    Let $R$ be the $m^{i-1} \times m^{i-1}$ zero matrix over $k$.
    For $\rho := 1$ to $r$ do
      $R \longleftarrow R +$ FastMatrixMultiplication$(f_\rho((M_{i,j})_{i,j}), g_\rho((N_{i,j})_{i,j})) \cdot C_\rho$.
    Output $R$.
```

If one wants to multiply two matrices M, N over k of size $n \times n$ for some n, one can apply the above algorithm to the matrices $\begin{pmatrix} M & O \\ O & O \end{pmatrix}, \begin{pmatrix} N & O \\ O & O \end{pmatrix}$ size $m^{\lceil \log_m(n) \rceil \times \lceil \log_m(n) \rceil}$.

In a suitable uniform algebraic model of computation (an algebraic RAM), one obtains a machine which operates in $O(n^{\log_m(r)})$ field operations.

Such a machine can then also be applied to matrices over extension fields of k. Moreover, using the tensor formulation of bilinear complexity, one can show that $R(\langle h, h, h \rangle_k)$ and therefore ω_k is invariant under field extensions. Therefore, it only depends on the characteristic of k ([BCS91, Corollary 15.8]). We therefore define:

Definition Let p be a prime or 0. Then we set ω_p as the exponent of matrix multiplication of fields of characteristic p.

We now describe an efficient adaption of the algorithm FastMatrixMultiplication for Turing machines for computations of finite fields of a fixed characteristic. For this we fix a bilinear computation as above over $\mathbb{F}_p$ and consider the following modification of the algorithm for the multiplication of matrices over extensions $\mathbb{F}_q$ of $\mathbb{F}_p$.

The essential idea is to proceed similarly to MergeSort, described in the previous section. However, in contrast to the previous, we cannot simply "unroll" the recursion and proceed with the smallest submatrices considered (that is, $m \times m$-matrices) first. Rather, given two $m^i \times m^i$-matrices, we have to iterate over ρ, and for each ρ, we first have to compute the values of the linear forms f_ρ and g_ρ on the matrices.

Recall that usually in function call, the current values of the variables are stored in a stack. We proceed in exactly the same way, that is, we use one tape as a stack, and whenever FastMatrixMultiplication is called, we write the current values of M, N and R to the stack. Let us say for concreteness that we write to the right whenever we put an element on the stack. Then in fact, we can even put the current values of M, N and R on the same tape, to the right of the stored elements in the stack. Note that the result is not a stack anymore because some data to the right of the tape is is now also of importance.

A fixed number of further tapes is used for intermediate computations. In particular, to compute the value of f_ρ or of g_ρ, we go over the current value of M, resp. N and store the result on a third tape. Then we write this to the right of the stack, followed by the zero matrix for R. Similarly, say that $\mathsf{FastMatrixMultiplication}(f_\rho((M_{i,j})_{i,j}), g_\rho((N_{i,j})_{i,j}))$ has been computed. Then we store the multiplication of this and C_ρ on a third tape and perform the addition with R on a forth one.

The running time of a machine as described can be estimated similarly to (1): Let $T(i, q)$ be the running time for matrices of size $m^i \times m^i$ over $\mathbb{F}_q$. Then there exists a constant $C > 0$ such that for all $i \geq 1$ and q,

$$T(i, q) \leq r \cdot T(i-1, q) + C \cdot m^{2i} \cdot \log(q) \ ,$$

$$T(0, q) = A(q) \ .$$

By analogous calculations as below (1) one obtains that the running time is in $O(r^i \cdot A(q)) = O(m^{i \log_m(r)} \cdot A(q))$.

Let now $\epsilon > 0$. Then we have some $m, r \in \mathbb{N}_{\geq 1}$ and a bilinear computation of length r of the multiplication of $m \times m$-matrices over $\mathbb{F}_p$ with $\log_m(r) \leq \omega_k + \epsilon$. This leads to:

Theorem 3 *Let a prime number p and $\epsilon > 0$ be fixed. Then there exists a Turing machine with the following specification:*

- *Input: A power q of p and two square matrices M, N over $\mathbb{F}_q$ of the same size*
- *Attributes: a natural number n with $M, N \in \mathbb{F}_q^{n \times n}$*
- *Output: $M \cdot N$.*
- *Running time: $O(n^{\omega_p + \epsilon} \cdot A(q))$*

- *Space requirements:* $O(n^2 \cdot \log(n) \cdot l(q) + A(q))$

We now come to the other computational problems associated with square matrices mentioned at the beginning of this section. As mentioned, matrix multiplication is central for studying these problems. The reduction to matrix multiplication is however not straightforward and is itself based on recursion. On the other hand, the adaption to the Turing model is quite similar to the one for matrix multiplication. For this reason, we only give a very brief description, following [BCS91].

First, one regards the problem to compute an inverse of a triangular matrix of full rank. This problem can again be tackled with a divide and counter algorithm, based on division of the matrix into four parts and matrix multiplication. One obtains that an analogous result as Theorem 3 holds also for this problem. In the same spirit, one can reduce the problem to compute an LUP-decomposition of a square matrix with a divide and counter algorithm to compute products and inverses of matrices. One again obtains a result as Theorem 3.

The problem to compute a reduced row echelon form of a square matrix can be reduced to the problem of matrix computation via three divide and counter algorithms which are nested in each other and the application of matrix multiplication. Given this, one can easily compute the kernel of the matrix. Again these problems can be computed in the time indicated above. In the same spirit, one can also solve an inhomogeneous system of equations in the indicated time.

Given that one can compute the kernel in the desired time bound, one can also decide if a square matrix is non-singular or not. If this is the case, one can easily compute the determinant using the LUP-decomposition.

With a different algorithm, based on the computation of row echelon forms, one can also compute a block diagonal matrix which is similar to the given square matrix and whose blocks are companion matrices of polynomials. From this, one can then read off the characteristic polynomial of the matrix.

All in all, as already stated, given any $\epsilon > 0$, one can perform all the computations discussed for an $n \times n$-matrix over $\mathbb{F}_q$ in a time of $O(n^{\omega_p+\epsilon} \cdot A(q))$.

We now want to obtain a machines for the tasks discussed above but for matrices over *arbitrary* finite fields. We first make the following observation:

Let a bilinear computation $\Gamma = (f_1, g_1, C_1; \ldots; f_r, g_r, C_r)$ for multiplication of $m \times m$-matrices over $\mathbb{Q}$ be given. Then for all prime numbers p for which the p-adic valuation of the coefficients of f_i, g_i and C_i is ≥ 0, we can reduce the computation modulo p to obtain a bilinear computation $\overline{\Gamma}$ of matrix multiplication of $m \times m$-matrices over $\mathbb{F}_p$.

This implies that every such computation sequence gives rise to an algorithm which for multiplication of square matrices over finite fields of nearly all characteristics. Such an algorithm can then be combined with algorithms for the remaining characteristics.

We define:

Definition Let $\omega := \sup_p \omega_p$, where p ranges over the prime numbers and 0.

Remark 1 It is known that for all p, $\omega_p < 2.38$ ([CW90]). Therefore $\omega < 2.38$ as well. Note also that the upper bound in [CW90] is equal for all p, and for every p it is the best known upper bound on ω_p.

We obtain immediately:

Theorem 4 *Let $\epsilon > 0$. Then there exists a Turing machine with the following specification:*

- *Input: A prime power q and M, N, square matrices of the same size over $\mathbb{F}_q$*
- *Attributes: a natural number n with $M, N \in \mathbb{F}_q^{n \times n}$*
- *Output: $M \cdot N$*
- *Running time: $O(n^{\omega+\epsilon} \cdot A(q))$*
- *Space requirements: $O(n^2 \cdot \log(n) \cdot l(q) + A(q))$*

Again, one can perform the other computations discussed above in a time of $O(n^{\omega+\epsilon} \cdot A(q))$ for any $\epsilon > 0$.

5 Sparse linear algebra

We now consider algorithms for sparse linear algebra over finite fields. The goal is to obtain efficient Turing machines for solving systems of sparse systems of linear equations. As algorithms in the literature for this problem rely crucially on the multiplication of matrices in sparse representation with vectors in dense representation, we study this problem first.

Let us first indicate what we mean by sparse representation of matrices over finite fields: Let first $\underline{v} \in \mathbb{F}_q^n$. Then for a sparse representation, we first represent $\underline{v}$ by the set of tuples (i, v_i) with $v_i \neq 0$. Then we represent this set by some list whose entries are the elements of the set (without repetition). We do not impose a condition on the ordering of the list.

Let now M be a matrix in $\mathbb{F}_q^{m \times n}$. We represent M by the set of tuples $(i, j, m_{i,j})$ with $m_{i,j} \neq 0$ and this set by a list, just as for vectors. Again, we do not impose a condition on the ordering of the list.

In order to multiply a matrix in sparse representation with a vector in dense representation, we first consider a Turing machine based on the following idea: First for each $j = 1, \ldots, n$, the entry v_j of $\underline{v}$ is brought near to all the entries of column j of the matrix. Then all multiplications are performed. After this, for each $i = 1, \ldots, m$, all the resulting products for row i are brought near to each other. Finally, for each row, the products are added and the final sum is output.

We now describe in greater detail a Turing algorithm based on this idea. (The italicized texts are information we need for the description of the algorithm; no computations are performed.)

MultiplicationByVector

Input: A prime power q, a matrix M over $\mathbb{F}_q$ in sparse representation and a vector $\underline{v}$ over $\mathbb{F}_q$ in dense representation such that the width of M is equal to the length of $\underline{v}$.

Output: $M\underline{v}$

Let $M \in \mathbb{F}_q^{m \times n}$, let L_M be the list representing M.

```
1. Compute a list $L_{\overline{v}}$ representing $\underline{v}$ in sparse representation.

2. Sort $L_M$ and $L_{\underline{v}}$ jointly in the following way: Let $L$ be the resulting list.
   Then $L$ is the join of sublists $L_j, j = 1, \ldots, m$, where $L_j$ starts with $(j, v_j)$,
   followed by all tuples $(i, j, m_{i,j})$ of $L_M$. Let $L$ be stored on tape 1.

3. Rewind all tapes and compute a list $L'$ on tape 2 in the following way:
   Repeat
       If $L$ ends here, then exit the Repeat loop.
       Read the current entry $(j, v_j)$ of $L$.
       Rewind tape 3 and copy $v_j$ to tape 3.
       Go to the next entry of $L$.
       Repeat
           If $L$ ends here, then exist both repeat loops.
           If the current entry is $(j', v'_j)$ for some $j'$,
               then exit the present repeat loop.
           Read the current entry $(i, j, m_{i,j})$ and store $(i, m_{i,j} \cdot v_i)$ in $L'$.
           Go to the next entry of $L$.

4. Sort $L'$ according to the first components of the tuples.
```

5. Rewind all tapes.
We use two variables i_0 and s. The variable i_0 is stored on tape 3. It indicates the current row under consideration. The variable s is stored on tape 4. It is used to compute the sums.

```
Let $i_0 \longleftarrow 1$, $s \longleftarrow 0$.
Repeat
    If $L'$ ends here then terminate.
    Repeat
        Read the first component $i$ of the current entry of $L'$,
            if $i \neq i_0$ then exit the present Repeat loop.
        Read the second component $c$ of the current entry of $L'$,
            let $s \longleftarrow s + c$.
            Go to the next entry of $L'$.
            If $L'$ ends here then exit the present Repeat loop.
    Output $s$.
    Let $i_0 \longleftarrow i$.
    Let $s \longleftarrow 0$.
```

Remark 2 The algorithm is reminiscent of Bernstein's proposal ([Ber01]) to use mesh sorting in order to multiply a matrix in sparse representation by a vector.

Let N be the number of non-zero entries of the matrix under consideration. Then with this algorithm, one can obtain a Turing machine with a running time of $O((N+n) \cdot l(N+n) \cdot (l(m)+l(n)+l(q)) + N \cdot \mathrm{A}(q) + (N+m) \cdot l(q))$. Here, the first term corresponds to the sorting, the second term to the multiplications and the last term to the computation of the final sums. Because of the second term, the last term can be substituted by $m \cdot l(q)$. If the zero in $\mathbb{F}_q$ is represented by a constant number of symbols, it can be substituted by m.

A slight improvement can be made: One can first sort L_M according to the second components and then only save tuples (j, v_j) for which column j of M is non-empty. One then obtains:

Theorem 5 *There exists a Turing machine with the following specification:*

- *Input: A prime power q, a matrix M over $\mathbb{F}_q$ in sparse representation and a vector $\underline{v}$ over $\mathbb{F}_q$ in dense representation such that the width of M is equal to the length of $\underline{v}$.*
- *Attributes:*
    - *m, n with $M \in \mathbb{F}_q^{m \times n}$*
    - *N, the number of non-zero entries of M*

- *Output: The vector $M\underline{v} \in \mathbb{F}_q^m$ in dense representation*
- *Running time: $O(N \cdot l(N) \cdot (l(m)+l(n)+l(q)) + (m+n) \cdot l(q) + N \cdot \mathrm{A}(q))$*
- *Space requirements: $O(N \cdot (l(m) + l(n) + l(q)) + A(q))$*

One can now for example use a suitable adaption of Lanczos's algorithm to finite fields to obtain a randomized Turing machine for solving sparse inhomogeneous linear systems. Such an adaption together with a rigorous analysis is given in [EK97, Section 6]. Here, in order to obtain a positive success probability, it is required that the field size be not too small in relation to the rank of the matrix. In [EG02, Section 4] it is discussed how one applies the algorithm to small fields too. The key idea is here to pass to an extension field and use the trace map to project the solution.

Let a matrix $M \in \mathbb{F}_q^{m \times n}$ of rank r in sparse representation and a vector $\underline{b} \in \mathbb{F}_q^m$ in dense representation be given. We consider the inhomogeneous linear system given by M and $\underline{b}$. According to [EK97, Theorem 6.2], if one applies the algorithm in [EK97, Section 6] to such a matrix, one needs $\leq r+3$ matrix-vector multiplications with M and also $\leq r+3$ matrix-vector multiplications with M^t (both from the right) and $O(nr)$ arithmetic operations in $\mathbb{F}_q$. However, as already remarked, it is only proven that the success probability is positive if q is not too small in relation to r.

With the modification in [EG02], one uses a field extension of $\mathbb{F}_q$ of size $\leq r^2q^2$. Note that the arithmetic in such a field can be performed in a time of $A(rq)$.

Altogether, with the algorithm in [EK97, Section 6], the modifications in [EG02] and the previous theorem one obtains an expected running time of $O((r+1) \cdot \big(N \cdot l(N) \cdot (l(m)+l(n)+l(q)) + (m+n) \cdot l(q) + (N+n) \cdot \mathrm{A}(rq)\big))$.

This can be slightly improved in the following way:

First, one sorts L_M according to the first components. Given this new list, one computes from M a smaller matrix in sparse representation in which all zero rows are deleted. At the same time, one computes a corresponding shorter right-hand side. More precisely: If for some i row i of M is non-zero, one stores b_i. If on the other hand row i of M is zero, one checks if b_i is 0, and if this is not the case, one outputs "failure".

Second, one computes from M a smaller matrix in sparse representation in which all zero columns are deleted. Here, a list of indices of non-zero columns of M has to be stored and used at the very end of the algorithm again in order to finally compute a solution to the original system.

In this way, one obtains:

Theorem 6 *There exists a randomized Turing machine with the following specification:*

- *Input: A prime power q, a matrix M over $\mathbb{F}_q$ in sparse representation and a vector $\underline{b}$ over $\mathbb{F}_q$ in dense representation such that the height of M is equal to the length of $\underline{b}$.*
- *Attributes:*
  - *m, n with $M \in \mathbb{F}_q^{m \times n}$*
  - *r, the rank of M*
  - *N, the number of non-zero entries of M*
- *Output: A vector $\underline{x} \in \mathbb{F}_q^n$ in dense representation with $M\underline{x} = \underline{b}$ or "failure" such that under the condition that the system is solvable, the probability of "failure" is $\leq \frac{1}{2}$.*
- *Running time: $O(r \cdot N \cdot l(N) \cdot (l(m) + l(n) + l(q)) + (m + n) \cdot l(q) + r \cdot N \cdot \mathrm{A}(rq))$*
- *Space requirements: $O(N \cdot (l(m) + l(n) + l(q)) + (m + n) \cdot l(q) + A(rq))$*

6 Factorization

We now come to the problem of integer factorization. The best known asymptotic result is due to H.W. Lenstra and C. Pomerance ([LP92]) who obtain an expected time of $\exp((1 + o(1)) \cdot (l(n) \cdot l(l(n)))^{\frac{1}{2}})$ for the complete factorization of a natural number n in a randomized RAM model. As will be seen shortly, the result holds in the randomized Turing model as well.

The essential part of the algorithm in [LP92] is a subalgorithm with the following characteristic (see Algorithm 10.1, Theorem 10.3 in [LP92]): If applied to a natural number n which is not a prime power, with a probability of at least $\frac{1}{64}$ it finds a non-trivial factorization of n. Moreover, the expected running time of the algorithm is the one indicated.

Together with an efficient primality test, one then easily obtains an algorithm which computes a complete factorization and which also runs in the indicated time. Concerning the primality test, the situation is in fact even better than at the time the algorithm was proposed because with the AKS-algorithm ([AKS04]) or a similar algorithm, one can now test for primality in polynomial time.

We now give some details on the subalgorithm in [LP92] mentioned above and its adaption to the Turing model.

As all fast randomized algorithms for integer factorization, the algorithm in [LP92] follows a "relation generation and linear algebra" strategy. A further key aspect of the algorithm is that the computations take place in class groups for discriminants of the form $-cn$ for small positive integers c.

The algorithm employs a connection, due to Gauß, between ambiguous classes (classes of order 2 in the class group) and factorizations of the corresponding discriminant. The relation generation and linear algebra strategy is employed in a very similar way to other, more basic factoring algorithms. Broadly speaking (and discarding important ideas as well) one proceeds as follows: First one fixes a discriminant which is of the form multiple of $-cn$ for a small positive integer c. One then uses a Monte Carlo algorithm to compute a generating system for the class group to the discriminant $c-n$. Let us call the output a *potential generating system*; it is a system of elements which is a generating system with probability $\geq \frac{1}{2}$. One computes a *factor base*. After these preliminary steps one tries to find relations between elements in the potential generating system and the factor base. If this is not successful after a predefined time, one terminates. Finally, with a linear algebra computation over $\mathbb{F}_2$ based on an algorithm from sparse linear algebra, one finds a non-trivial ambiguous form, which is used to obtain a non-trivial factorization.

The factor base has a size of $\exp\left((\frac{1}{2} + o(1)) \cdot (l(n) \cdot l(l(n)))^{\frac{1}{2}}\right)$, and thus the linear algebra computation takes place on a sparse matrix of this size as well. The size of the generating system is asymptotically negligible. For an adaption to the Turing model we make the following remarks: In a randomized RAM model, all parts of the algorithm except relation generation and linear algebra can be performed in an expected running time of $\exp\left(o(1) \cdot (l(n) \cdot l(l(n)))^{\frac{1}{2}}\right)$. Obviously, this can be achieved in the Turing model too. Moreover, one try to find a relation between elements of the potential generating system and factor base elements also has such an expected running time. So all we have to consider is the expected number of tries and the expected time for the linear algebra computation. The expected number of tries is in $\exp\left((1+o(1)) \cdot (l(n) \cdot l(l(n)))^{\frac{1}{2}}\right)$, leading to an expected running time of $\exp\left((1+o(1)) \cdot (l(n) \cdot l(l(n)))^{\frac{1}{2}}\right)$ for the relation generation part. By Theorem 6 and the sparsity of the relation matrix, one can achieve such an expected running time for the linear algebra part too.

Therefore one obtains:

Theorem 7 *There exists a randomized Turing machine with the following specification:*

- *Input: Some natural number n*
- *Output: The factorization of n*
- *Expected running time:* $\exp\left((1 + o(1)) \cdot (l(n) \cdot l(l(n)))^{\frac{1}{2}}\right)$
- *Space requirements:* $\exp\left((\frac{1}{2} + o(1)) \cdot (l(n) \cdot l(l(n)))^{\frac{1}{2}}\right)$

7 Discrete logarithms

We study discrete logarithm problems in various classes of groups.

7.1 The classical discrete logarithm

A classical algorithm to compute discrete logarithm problems is the baby step giant step algorithm.

We give here the essential idea of the algorithm.

We start with an observation: Let G be a group and $a, b \in G$ with $b \in \langle a \rangle$. Let now e be the discrete logarithm of b with respect to a. Let B be a bound on the order of a. Then $e < B$, and we can write $e = i + j \cdot \lceil \sqrt{B} \rceil$ with $i, j \in \{0, 1, \ldots, \lceil \sqrt{B} \rceil - 1$. We therefore have $a^{j \cdot \lceil \sqrt{B} \rceil} = b \cdot a^{-i}$ with i, j as before.

The input to the algorithm is now the group $G, a, b \in G$ and the bound B. We do not assume that $b \in \langle a \rangle$. Then the two lists consisting of $(j, a^{j \cdot \lceil \sqrt{B} \rceil})$ respectively $(i, b \cdot a^{-i})$ with $i, j \in \{0, 1, \ldots, \lceil \sqrt{B} \rceil - 1\}$ are computed and sorted for the second entries. If matches are found, one easily obtains the discrete logarithm, and if not then $b \notin \langle a \rangle$.

By combining the baby step giant step algorithm with a sorting algorithm following the specifications of Theorem 2, we obtain:

Theorem 8 *There exists a Turing machine with the following specification:*

- *Input: A prime power q, $a, b \in \mathbb{F}_q^*$ and $B \in \mathbb{N}$ with $B \geq \mathrm{ord}(a)$.*
- *Output:*
  - *If $b \in \langle a \rangle$: the discrete logarithm of b with respect to a*
  - *if $b \notin \langle a \rangle$: "no"*
- *Running time: $O(\sqrt{B} \cdot (l(B) \cdot l(q) + A(q)))$*
- *Space requirements: $O(\sqrt{B} \cdot l(q) + A(q))$*

One can now also vary the bound B. If we apply the algorithm with powers of 2 for B, we obtain:

Theorem 9 *There exists a Turing machine with the following specification:*

- *Input: A prime power q and $a, b \in \mathbb{F}_q^*$.*
- *Output:*
  - *If $b \in \langle a \rangle$: the discrete logarithm of b with respect to a*
  - *if $b \notin \langle a \rangle$: "no"*

- *Running time:* $O(\sqrt{\mathrm{ord}(a)} \cdot l(\mathrm{ord}(a)) \cdot (l(\mathrm{ord}(a)) \cdot l(q) + A(q)))$
- *Space requirements:* $O(\sqrt{\mathrm{ord}(a)} \cdot l(q) + A(q))$

Just as the problem of factoring integers, the problem of computing discrete logarithms in a particular class of groups can be approached via a relation generation and linear algebra method. It is common to refer to this method as *index calculus.* (*Index* is a classical term for discrete logarithm in multiplicative groups of prime fields with respect to generating elements.)

The most basic index calculus algorithms roughly operate as follows: First a *factor base* is fixed. Then relations between the input elements and elements of the factor base are found. If enough relations have been obtained, one tries to find a relation involving only the input elements, using a linear algebra algorithm. From this relation, one tries to derive the discrete logarithm.

A good overview over index calculus algorithms for various classes of groups is given in [EG02]. By combining the algorithm for prime fields in [EG02] with a machine for sparse linear algebra following the specification of Theorem 6, we obtain the following two theorems.

Theorem 10 *There exists a randomized Turing machine with the following specification:*

- *Input: A prime number p and $a, b \in \mathbb{F}_p^*$ with $a \in \langle a \rangle$*
- *Output: The discrete logarithm of b with respect to a*
- *Expected running time:* $\exp\left((\sqrt{2} + o(1)) \cdot (l(p) \cdot l(l(p)))^{\frac{1}{2}}\right)$
- *Space requirements:* $\exp\left((\frac{1}{\sqrt{2}} + o(1)) \cdot (l(p) \cdot l(l(p)))^{\frac{1}{2}}\right)$

There is also an algorithm for extension fields in [EG02]. For the application of this algorithm, it is necessary that the ground field of the extension field is a prime field. With a suitable variant of the representation of elements and the definition of the factor base, this condition can however be dropped. The definition of the factor base is in fact easier than the one in [EG02].

Let us assume that we want to compute a discrete logarithm in $\mathbb{F}_{q^n}^*$. We take $\mathbb{F}_q$ as the ground field and fix a monic irreducible polynomial f of degree n (such that $\mathbb{F}_{q^n} \simeq \mathbb{F}_q[X]/(f)$). Now each element in $\mathbb{F}_{q^n}$ is uniquely represented by a monic polynomial of degree n in $\mathbb{F}_q[X]$. As in [EG02], we fix a smoothness bound S. We define as factor base the set of all irreducible monic polynomials of degree $\leq S$. (This approach is similar to the approach for class groups of hyperelliptic curves in imaginary quadratic representation in [EG02].)

Let us now fix some constant $\theta > 0$ and consider instances with $n \geq \theta \cdot l(q)$. Then in particular one has $q \in \exp(o(1) \cdot (l(q^n) \cdot l(l(q^n)))^{\frac{1}{2}})$. One obtains:

Theorem 11 *Let $\theta > 0$ be fixed. Then there exists a randomized Turing machine with the following specification:*

- *Input: A prime power q, a natural number n, and $a, b \in \mathbb{F}_{q^n}^*$ with $b \in \langle a \rangle$ and $n \geq \theta \cdot l(q)$*
- *Output: The discrete logarithm of b with respect to a*
- *Expected running time:* $\exp\left((\sqrt{2} + o(1)) \cdot (l(q^n) \cdot l(l(q^n)))^{\frac{1}{2}}\right)$
- *Space requirements:* $\exp\left(((\frac{1}{\sqrt{2}} + o(1)) \cdot (l(q^n) \cdot l(l(q^n)))^{\frac{1}{2}}\right)$

Remark 3 In [EG02] it is assumed that $l(q) \in o(n)$ in order that the above bound on the expected running time holds (additionally to the restriction that q be prime and in a randomized RAM model). However, with the above remark that one automatically has $q \in \exp(o(1) \cdot (l(q^n) \cdot l(l(q^n)))^{\frac{1}{2}})$, it is immediate that the analysis in [EG02] can be improved. A similar remark holds concerning the analysis for class groups of hyperelliptic curves in imaginary quadratic representation.

Remark 4 The reader might ask for a reason why one can obtain a better expected running time for factorization than for the computation of discrete logarithms in prime fields, even though the two problems seem to be so closely related and both algorithms are based on a relation generation and linear algebra approach. Briefly, one can give this answer: The factorization algorithm by Lenstra and Pomerance is based on computations in class groups with discriminants Δ with $-\Delta \in \Theta(n)$. Now, the order of the class group of discriminant Δ is $|\Delta|^{\frac{1}{2}+o(1)}$ for $\Delta \longrightarrow -\infty$ by Siegel's theorem. If the bound on the expected running time of the factorization algorithm is expressed in terms of the group order of the class group, then one sees the similarity of the results for factorization and discrete logarithm computations.

7.2 The discrete logarithm problem in class groups of curves

We now come to the discrete logarithm problem in degree 0 class groups (Picard groups) of curves over finite fields. (A *curve* is always assumed to be proper and non-singular.)

Let us briefly give some information on the representation of curves and divisors and divisor classes on curves, as this is not straightforward.

Further information on these topics can be found in [Die11b], and even more information and details can be found in [Die08].

First, curves are given by what we call plane models, that is plane curves which are birational to the curves under consideration. Following [Heß01], we take an ideal theoretic approach to the representation of divisors. Divisors are represented by tuples of two ideals in two orders of the function field of the curves, a "finite" and an "infinite" order.

One can show that curves over finite fields have plane models of degree $O(g)$, where g is the genus of the curve. We represent curves by such models. Furthermore, every curve over a finite field has a divisor D_0 of degree 1 and height $O(g)$; see [Heß05]. We fix such a divisor D_0. We then represent an arbitrary divisor D by an along D_0 reduced divisor. This is an effective divisor $\tilde{D}$ such that D is linearly equivalent to $\tilde{D} + (D - \tilde{D}) \cdot D_0$ and the linear system $|\tilde{D} - D_0|$ is empty.

With the baby step giant step algorithm and arguments in [Die11b, Section 2] one can obtain the following theorem. Note here that the class of inputs in the following specification includes all instances of the elliptic curve discrete logarithm problem.

Theorem 12 *There exists a Turing machine with the following specification:*

- *Input: A prime power q, a curve $\mathcal{C}/\mathbb{F}_q$ and $a, b \in \mathrm{Cl}^0(\mathcal{C})$*
- *Output:*
  - *If $b \in \langle a \rangle$: the discrete logarithm of b with respect to a*
  - *if $b \notin \langle a \rangle$: "no"*
- *Running time: $\tilde{O}(\sqrt{\# \mathrm{Cl}^0(\mathcal{C})})$*
- *Space requirements: $\tilde{O}(\sqrt{\# \mathrm{Cl}^0(\mathcal{C})})$*

Index calculus algorithms are also available for curves. A very general result with an attractive expected running time is due to Heß ([Heß05]). By again using a machine for sparse linear algebra following the specification of Theorem 6, we immediately obtain the same result in the Turing model. So we have the following theorem.

Theorem 13 *Let $\epsilon > 0$ and $\theta > 0$ be fixed. Then there exists a randomized Turing machine with the following specification:*

- *Input: A prime power q, a curve $\mathcal{C}/\mathbb{F}_q$ with $g \geq \theta \cdot l(q)$, where g is the genus, and $a, b \in \mathrm{Cl}^0(\mathcal{C})$ with $b \in \langle a \rangle$.*

- *Output: The discrete logarithm of* b *with respect to* a.
- *Expected running time:* $\exp\left((\sqrt{32}+\epsilon)\cdot(l(q^g)\cdot l(l(q^g)))^{\frac{1}{2}}\right)$
- *Space requirements:* $\exp\left((\sqrt{8}+\epsilon)\cdot(l(q^g)\cdot l(l(q^g)))^{\frac{1}{2}}\right)$

Remark 5 Just as in [Heß05] we stated the bound on the expected running time and the space requirements in terms of q^g. It is however more satisfying to state the complexities in terms of the group order, $\#\,\mathrm{Cl}^0(\mathcal{C})$. Fortunately, by a result due to Lachaud and Martin-Deschamps ([LMD90]), we have $q^g \in \tilde{O}(\#\,\mathrm{Cl}^0(\mathcal{C}))$ for all curves over finite fields. This makes it possible to replace q^g by $\#\,\mathrm{Cl}^0(\mathcal{C})$. This observation should also be kept in mind for the following theorems.

Remark 6 As mentioned in [Heß05], if one considers curves given as coverings of the projective line with a bounded covering degree, one can obtain better exponents. For example, for hyperelliptic curves (and again for $g \geq \theta \cdot l(q)$) one obtains an expected running time of $\exp\left((2+\epsilon)\cdot(l(q^g)\cdot l(l(q^g)))^{\frac{1}{2}}\right)$ for any $\epsilon > 0$. If one considers merely hyperelliptic curves in imaginary quadratic representation, one obtains an expected running time of $\exp\left(\sqrt{2}+o(1))\cdot(l(q^g)\cdot l(l(q^g)))^{\frac{1}{2}}\right)$. (If the degree 0 class group is cyclic, this result follows from [EG02], for the general case, the algorithm in [EG02] can be modified according to the description in [Heß05].) Note here the similarity with the previous results on the classical discrete logarithm problem.

Another line of research is to consider only curves of a fixed genus $g \geq 2$. Here it is state of the art to proceed as follows: Let $\mathcal{C}/\mathbb{F}_q$ be a curve under consideration. One fixes a factor base $\mathcal{F} \subseteq \mathcal{C}(\mathbb{F}_q)$, and one considers the set $\mathcal{L} := \mathcal{C}(\mathbb{F}_q) - \mathcal{F}$ as "large primes". (The name comes from the classical discrete logarithm problem or the problem of integer factorization and is a bit misplaced here.)

One uses relations between the input elements, factor base elements and large primes to construct in one way or the other a *graph of large prime relations* on $\mathcal{L} \,\dot{\cup}\, \{*\}$. For technical reasons, it is common to not use a full graph of large prime relations for algorithms leading to theoretical results but rather a tree of large prime relations. But even then, there are different approaches: In [GTTD07], where the discrete logarithm problem in hyperelliptic curves in imaginary quadratic representation is considered, each relation which enlarges the tree is immediately inserted into the tree. In [Die11b], where arbitrary curves are considered, enlargement is performed in stages in the following way: Let us call the beginning stage 0. Then in stage s, only edges linked to points present at the end of stage $s-1$ are inserted. The algorithm in [Nag07] can be interpreted as follows: The computation is performed in stages, and in stage s first a list of relations is generated. After that such a list is used to enlarge the tree. Here again, only edges linked to points present at the end of stage $s-1$ are inserted.

In [Die11b] it is proven that in a randomized RAM model, one can solve the discrete logarithm problem in the degree 0 class groups of curves of a fixed genus g in an expected time of $\tilde{O}(q^{2-\frac{2}{g}})$. We wish to obtain such an expected running time in the Turing model too. Now the construction of the tree of large prime relations in [Die11b] does not immediately lead to a suitable algorithm in the Turing model. The reason is that it is not efficient to insert one edge into the tree at one time. The algorithm can however easily be modified. Let us in order that we can describe the modification recall some information from [Die11b].

In the algorithm in [Die11b, Section 2], a "potential generating systems" $s_1, \ldots, s_u$ of $\mathrm{Cl}^0(\mathcal{C})$ is used. To construct the tree of large prime relations, $c_1, \ldots, c_u$ are drawn uniformly at random modulo the group order and reduced divisors D with $[D] - \deg(D) \cdot [P_0] = \sum_j s_j c_j$ are computed. Let us call such an equality a relation (without a condition on D). It is proven that in each stage of the construction of the tree of large prime relations, the expected number of relations which have to be considered is $\leq 16(g-1) \cdot q^{1-\frac{1}{g}} \cdot (q+1)^{2-\frac{2}{g}}$.

The modification for adaption to the Turing model is as follows: In stage s we first generate $\lceil 32(g-1) \cdot q^{1-\frac{1}{g}} \cdot (q+1)^{2-\frac{2}{g}} \rceil$ relations, and we store the relations such that D splits completely into rational points in a list. Then we copy the tree, and via a sorting algorithm applied to the tree and the list, we insert new edges into the tree such that the result is as follows: We again have a tree, and for each large prime P which is not yet in the tree for which there exists a relation linking P to the tree of stage $s-1$, exactly one new edge occurs in the enlarged tree. If the enlarged tree is large enough (in the sense of the algorithm in [Die11b]), the next stage is considered. Otherwise the stage is repeated with the old tree. Note here that with these modifications, the algorithm can be seen as a close variant of Nagao's algorithm ([Nag07]).

The probability that the tree is enlarged with one repetition is then $\geq \frac{1}{2}$. (Consider Markov's inequality applied to the number of relations until the tree is large enough.) Thus the expected number of repetitions in each stage is ≤ 2. We obtain:

Theorem 14 *Let $g \in \mathbb{N}$ with $g \geq 2$ be fixed. Then there exists a randomized Turing machine with the following specification:*

- *Input: A prime power q, a curve $\mathcal{C}/\mathbb{F}_q$ of genus g and $a, b \in \mathrm{Cl}^0(\mathcal{C})$ with $b \in \langle a \rangle$*

- *Output: The discrete logarithm of b with respect to a*
- *Expected running time: $\tilde{O}(q^{2-\frac{2}{g}})$*
- *Space requirements: $\tilde{O}(q^{1-\frac{1}{g}+\frac{1}{g^2}})$*

Note here that for fixed genus g, we have $\# \mathrm{Cl}^0(\mathcal{C}) \sim q^g$.

Remark 7 For fixed $g \geq 3$ the expected running time is smaller than the one given in Theorem 12. For $g = 2$ the expected running time is up to logarithmic factors equal to the running time one obtains with Theorem 12. However, now the space requirements are smaller: In Theorem 12 they are $\tilde{O}(q)$ whereas here they are $\tilde{O}(q^{\frac{3}{4}})$.

Using the results in [Die11b, Section 4], we now obtain also:

Theorem 15 *Let $g_0 \in \mathbb{N}$ with $g_0 \geq 3$ be fixed. Then there exists a randomized Turing machine with the following specification:*

- *Input: A prime power q, a curve $\mathcal{C}/\mathbb{F}_q$ of genus $g \geq g_0$ and $a, b \in \mathrm{Cl}^0(\mathcal{C})$ with $b \in \langle a \rangle$*
- *Output: The discrete logarithm of b with respect to a*
- *Expected running time: $\tilde{O}(\# \mathrm{Cl}^0(\mathcal{C})^{\frac{2}{g_0}(1-\frac{1}{g_0})})$*
- *Space requirements: $\tilde{O}(\# \mathrm{Cl}^0(\mathcal{C})^{\frac{1}{g_0}(1-\frac{1}{g_0}+\frac{1}{g_0^2})})$*

In particular, for $g_0 = 3$, we obtain an expected running time of $\tilde{O}(\# \mathrm{Cl}^0(\mathcal{C})^{\frac{4}{9}})$ with space requirements of $\tilde{O}(\# \mathrm{Cl}^0(\mathcal{C})^{\frac{14}{27}})$.

In [Die11a] we consider the discrete logarithm problem in the degree 0 class groups of curves which are represented by plane models of a fixed degree. The algorithms in [Die11a] immediately lead to satisfying Turing machines. (One should keep in mind the changes under the headline "On the storage requirements" at the end of Section 2.)

We therefore obtain:

Theorem 16 *Let $d \in \mathbb{N}$ with $d \geq 4$ be fixed. Then there exists a randomized Turing machine with the following specification:*

- *Input: A prime power q, a curve $\mathcal{C}/\mathbb{F}_q$, represented by a plane model of degree d, where $d = 4$ or the plane model is reflexive, and $a, b \in \mathrm{Cl}^0(\mathcal{C})$ with $b \in \langle a \rangle$*
- *Output: The discrete logarithm of b with respect to a*

- *Expected running time:* $\tilde{O}(q^{2-\frac{2}{d-2}})$
- *Space requirements:* $\tilde{O}(\max\{q^{1-\frac{1}{d-2}}, q^{1-\frac{1}{g}+\frac{1}{(d-2)g}}\})$

7.3 The discrete logarithm problem in elliptic curves

The discrete logarithm problem in elliptic curves over finite fields is a special case of the discrete logarithm problem in the degree 0 class groups of curves. However, no randomized RAM is known which can solve the discrete logarithm problem for elliptic curves in an expected time of $o(E(\mathbb{F}_q)^{\frac{1}{2}})$, where $E/\mathbb{F}_q$ is the input curve.

On the other hand, for some sequences of finite fields, such a result can be obtained for all elliptic curves over these fields. Following ideas by Gaudry ([Gau09]), we have shown in [Die11b] that for any fixed $n \geq 2$, the discrete logarithm problem in elliptic curves over fields of the form $\mathbb{F}_{q^n}$, q being a prime power, can be solved in an expected time of $\tilde{O}(q^{2-\frac{2}{n}})$ on a randomized RAM.

By an adaption of the considerations for Theorem 14, we obtain the following theorem.

Theorem 17 *Let $n \in \mathbb{N}$ with $n \geq 2$ be fixed. Then there exists a randomized Turing machine with the following specification:*

- *Input: A prime power q, an elliptic curve $E/\mathbb{F}_{q^n}$, $a, b \in E(\mathbb{F}_{q^n})$ with $b \in \langle a \rangle$*
- *Output: The discrete logarithm of b with respect to a*
- *Expected running time:* $\tilde{O}(q^{2-\frac{2}{n}})$
- *Space requirements:* $\tilde{O}(q^{1-\frac{1}{n}+\frac{1}{n^2}})$

As a final remark we mention that there also exists a sequence of finite fields of strictly increasing cardinality over which the elliptic curve discrete logarithm problem can be solved in an expected time of $\exp(O(l(q)^{\frac{2}{3}}))$, where q is the cardinality of the ground field ([Die11c]). As no statement on the constant in the exponent is made, clearly this result holds in the Turing model too.

References

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Ann. of Math.*, 160(2):781–793, 2004.

[BCS91] P. Bürgisser, M. Clausen, and M.A. Shokrollahi. *Algebraic Complexity Theory.* Springer-Verlag, 1991.

[Ber01] D. Bernstein. Circuits for integer factorization: a proposal. Available under cr.yp.to/papers/nfscircuit.pdf, 2001.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw-Hill and The MIT Press, 2001. Second Edition.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. *J. Symb. Comp.*, 9:251–280, 1990.

[Die08] C. Diem. On arithmetic and the discrete logarithm problem in class groups of curves, 2008. Habilitation thesis.

[Die11a] C. Diem. On the discrete logarithm problem for plane curves. Submitted, 2011.

[Die11b] C. Diem. On the discrete logarithm problem in class groups of curves. *Math. Comp.*, 80:443–475, 2011.

[Die11c] C. Diem. On the discrete logarithm problem in elliptic curves. *Compos. Math.*, 147:75–104, 2011.

[Die11d] C. Diem. On the notion of bit complexity. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS*, 103:35–52, 2011. In the "Complexity Column".

[EG02] A. Enge and P. Gaudry. A general framework for subexponential discrete logarithm algorithms. *Acta. Arith.*, 102:83–103, 2002.

[EK97] W. Eberly and E. Kaltofen. On randomized Lanczos algorithms. In W. Küchlin, editor, *Proceedings ISSAC 1997*, pages 176–183. ACM Press, 1997.

[Für07] M. Fürer. Faster integer multiplication. In D. Johnson and U. Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing*, pages 57–66. ACM 2007, 2007.

[Gau09] P. Gaudry. Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem. *J. Symb. Comput.*, 44:1690–1702, 2009.

[GG03] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge Unversity Press, 2003.

[GTTD07] P. Gaudry, E. Thomé, N. Thériault, and C. Diem. A double large prime variation for small genus hyperelliptic index calculus. *Math. Comp.*, 76:475–492, 2007.

[Heß01] F. Heß. Computing Riemann-Roch spaces in algebraic function fields and related topics. *J. Symb. Comput.*, 11, 2001.

[Heß05] F. Heß. Computing relations in divisor class groups of algebraic curves over finite fields. Preprint, ca. 2005.

[LMD90] G. Lachaud and M. Martin-Deschamps. Nombre de points des jacobiennes sur un corps fini. *Acta. Arith.*, 56:329–340, 1990.

[LP92] H.W. Lenstra and C. Pomerance. A rigorous time bound for factoring integers. *J. Amer. Math. Soc.*, 5, 1992.

[Nag07] K. Nagao. Index calculus attack for Jacobian of hyperelliptic curves of small genus using two large primes. *Japan J. Indust. Appl. Math.*, 24, 2007.

[Pap94] C. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[San12] R. Santhanam. Ironic complicity: satisfiablility algorithms and circuit lower bounds. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS*, 106:32 – 52, 2012. In the "Computational Complexity Column".

[SGV94] A. Schönhage, A. Grotefeld, and E. Vetter. *Fast algorithms – a multitape Turing machine implementation.* BI Wissenschaftsverlag, Mannheim, 1994.

[SS71] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[Vet98] J. Vetter. External memory algorithms and data structures. In Abello and Vetter, editors, *DIMACS Workshop External Memory Algorithms and Visualization*, Series in Discrete Mathematics and Theoretical Computer Science, pages 1–38. AMS, 1998.

[Vet01] J. Vetter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Computing Surveys*, 33:209271, 2001.

Claus Diem
Universität Leipzig
Mathematisches Institut
Augustusplatz 10
04109 Leipzig
Germany
diem@math.uni-leipzig.de