MATHEMATICS OF COMPUTATION
Volume 67, Number 224, October 1998, Pages 1637–1663
S 0025-5718(98)00968-5

A SPACE EFFICIENT ALGORITHM FOR GROUP STRUCTURE COMPUTATION

EDLYN TESKE

ABSTRACT. We present a new algorithm for computing the structure of a finite abelian group, which has to store only a fixed, small number of group elements, independent of the group order. We estimate the computational complexity by counting the group operations such as multiplications and equality checks. Under some plausible assumptions, we prove that the expected run time is $O(\sqrt{n})$ (with n denoting the group order), and we explicitly determine the O-constants. We implemented our algorithm for ideal class groups of imaginary quadratic orders and present experimental results.

1. INTRODUCTION

Let G be a finite abelian group, written multiplicatively, for which we assume the following:

- For $a, b \in G$ we can compute $c = a * b$ and we can test whether $a = b$.
- We know the neutral element $1 \in G$.
- There is a computable function $f : G \to \{1, \dots, 20\}$ such that
$$\sum_{i=1}^{20} \left| \#\{a \in G : f(a) = i\} - \frac{|G|}{20} \right| = O(\sqrt{|G|}),$$
where $|G|$ denotes group order.

Throughout the paper, we refer to the function f of the third assumption as the *equidistributing function.* Let us already note that this function will serve to produce random walks in G. The number 20 entered the definition empirically.

For any subset S of G, denote by $\langle S \rangle$ the subgroup of G generated by S. If $\langle S \rangle = G$ then S is called a *generating set* of G. If $S = \{g\}$ then we write $\langle g \rangle$ instead of $\langle S \rangle$.

Given a generating set of G, we want to compute the *structure* of G. By computing the structure of G we mean computing positive integers $m_1, \dots, m_q$ with $m_1 > 1$, $m_i | m_{i+1}$, $1 \le i < q$ and an isomorphism $\phi : G \to \mathbb{Z}/m_1\mathbb{Z} \times \dots \times \mathbb{Z}/m_q\mathbb{Z}$. This isomorphism is given in terms of the images of the generators. The integers m_i are the uniquely determined *invariants* of G.

Note that once we have an algorithm to compute the group structure, we immediately can solve two other computational problems, which are the following:

Received by the editor February 7, 1997 and, in revised form, April 23, 1997.
1991 *Mathematics Subject Classification.* Primary 11Y16.
Key words and phrases. Generic algorithms; group structure computation; Pollard's ρ-method; class groups.

- Given $g \in G$, compute $|\langle g \rangle|$, the *order* of g in G, which is the least positive integer x such that $g^x = 1$.
- Given $g, d \in G$, decide whether d belongs to the cyclic subgroup $\langle g \rangle$ of G generated by g. If $d \in \langle g \rangle$, find $\log_g d$, the *discrete logarithm* of d to the base g, i.e., the least non-negative integer x such that $g^x = d$.

In this paper, we present a generic algorithm to compute the structure of a finite abelian group. By "generic" we mean that the algorithm does not exploit any special properties of the group operations or the encodings of the group elements. The algorithm we present works for every finite abelian group satisfying the three assumptions stated above. To determine the computational complexity of such a generic algorithm we count the number of group operations such as multiplications and equality checks.

Here is our main result.

Theorem 1.1. *There is an algorithm for computing the structure of a finite abelian group G from a generating set S that has to store*

$$|S| + 30$$

group elements and vectors in $\{1, \dots, 5|G|^{5/2}\}^{|S|}$. Under some plausible assumptions, it executes an expected number of at most

$$(1.41|S| + 5)\sqrt{|G|} + 5|G|^{1/4}(\lceil \log\log_{10}|G|\rceil - 1)$$
$$+20|S|\,(2\log|G| + 1)\,(|S| + 1 + 2\lceil \log\log_{10}|G|\rceil) + O(|S|)$$

group multiplications, for $|G| \geq 4$. It executes an expected number of at most

$$8(1.41|S| + 5)\sqrt{|G|} + 28|G|^{1/4}(\lceil \log\log_{10}|G|\rceil - 1) + O(|S|)$$

equality checks.

(Note that throughout this paper, $\log b$ stands for $\log_2 b$.)

Let us stress that this algorithm always terminates and that its output is always correct, regardless of whether our assumptions (stated in Conjectures 6.1 and 6.2) do hold or not. It is only the expected run time that depends on their validity.

Shoup [Sho96] has proved an $\Omega(p^{1/2})$ lower bound for the computational complexity of every generic algorithm computing discrete logarithms in a finite abelian group, where p denotes the largest prime divisor of the group order. The same holds for the group structure computation. Thus, in case of groups with prime group order, no algorithm can asymptotically do better than our algorithm.

There is another generic algorithm for group structure computation [BJT97], which is based on Shanks' Baby-Step Giant-Step method [Sha71]. It has run-time complexity $O(\sqrt{|G|})$, but it has the disadvantage that it has high storage requirements. We have shown [BJT97] that it uses tables of group elements of size $\Omega(\sqrt{|G|})$. Therefore, for example on a SPARCstation ULTRA170 with 170 MB RAM and with our implementation for ideal class groups of imaginary quadratic orders, we did not succeed in computing groups with group order larger than 10^{11} because of the memory constraints of the machine.

On the other hand, we know space efficient algorithms for computing the element order and the discrete logarithm, which use Pollard's rho method [Pol78] [SS85] [McC90]. These algorithms store only a small, constant number of group elements, but their run-time complexities can no longer be rigorously proved. Their expected run time is $O(\sqrt{|G|})$, under the assumptions that a random walk in the group can be simulated and that an upper bound for the group order (for the order algorithm) or the group order itself (for the discrete logarithm algorithm) is known.

In this paper we show, both theoretically and experimentally, that Pollard's rho method can also be used to compute the group structure. As already mentioned, the algorithm we present has constant storage requirements and expected run time $O(\sqrt{|G|})$. We do not need any prior knowledge about the group order.

Our paper is organized as follows. First, we present the results from the theory of finite abelian groups on which our group structure algorithm is based, and we give an outline of the algorithm. In Section 3 we show how to realize random walks in finite abelian groups. There the role of the equidistributing function will also become clear. In Sections 4 and 5 we explain in detail how to compute the relations needed for our group structure computation and how to minimize them. In Section 6 we explicitly state the two assumptions on which our run-time analysis is based, and we prove our complexity results. Finally, in Section 7 we present a selection of experimental results that we obtained with our implementation for ideal class groups of imaginary quadratic orders.

2. Description of the algorithm

Given a generating set of a finite abelian group G that is given as described in the introduction, we want to find the structure of G. By this we mean finding positive integers $m_1, \dots, m_q$ with $m_1 > 1$, $m_j | m_{j+1}$, $1 \leq j < q$, and an isomorphism

$$\phi : G \longrightarrow \mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_q\mathbb{Z}. \tag{2.1}$$

This isomorphism will be given in terms of the images of the elements of the generating set S. The integers m_i are the *invariants* of G.

Since our algorithm may behave differently if the generators are input in different orders, we speak in the following of S as a *generating sequence*, i.e., as a finite sequence $S = (g_1, \dots, g_l)$ of group elements such that $\{g_1, \dots, g_l\}$ is a generating set of G. For $\vec{z} = (z_1, \dots, z_l) \in \mathbb{Z}^l$ we write

$$S^{\vec{z}} = \prod_{i=1}^{l} {g_i}^{z_i}.$$

A *relation on* S is a vector $\vec{z} \in \mathbb{Z}^l$ such that $S^{\vec{z}} = 1$. Consider the surjective homomorphism

$$\Theta : \mathbb{Z}^l \longrightarrow G, \quad \vec{z} \mapsto S^{\vec{z}}.$$

Then for the set $L(S)$ of all relations on S we have

$$\operatorname{kernel}(\Theta) = \left\{\vec{z} \in \mathbb{Z}^l \ : \ S^{\vec{z}} = 1\right\} = L(S).$$

Hence,

$$G \cong \mathbb{Z}^l / L(S),$$

which implies that $L(S)$ is a lattice in $\mathbb{Z}^l$ of dimension l and $|G| = |\det(L(S))|$.

Our algorithm computes a basis $B = (\vec{b}_1, \dots, \vec{b}_l)$ of $L(S)$. This basis is identified with a matrix of column vectors $\vec{b}_j = (b_{1j}, \dots, b_{lj})^T$, where ^{T} denotes transposition of vectors. The matrix is in Hermite normal form. Then the order of G is $|\det B|$. The structure of G we find by computing the Smith normal form $\mathrm{SNF}(B)$ of B, because if $\mathrm{SNF}(B)$ is the diagonal matrix $\mathrm{diag}(1, \dots, 1, m_1, \dots, m_q)$ where $m_1 > 1$, then $m_1, \dots, m_q$ are the invariants of G. The isomorphism (2.1) is derived from the

Smith normal form and the corresponding transformation matrices. See [BJT97] for details.

In order to explain how to compute the relations, we need the following facts and definitions. If W is a set, we indicate by $w \in_R W$ that w is a randomly chosen element of W. Let $F : G \to G$ be a mapping chosen at random in the sense of Knuth [Knu75, p.8], i.e., each of the $|G|^{|G|}$ possible functions on G is equally probable. Now consider the sequence (h_k) in G formed by the rule

$$h_0 \in_R G, \quad h_{k+1} = F(h_k),\ k \in \mathbb{N}_0.$$

This sequence is ultimately periodic. Let λ denote the *period* and μ the *length of the non-periodic segment* of the sequence, i.e., $h_0, \dots, h_{\mu+\lambda-1}$ are pairwise distinct and $h_\mu = h_{\mu+\lambda}$. Figuratively, the sequence (h_k) is ρ-shaped, with tail of length μ and cycle of circumference λ. The expected values of λ and μ are both close to $\sqrt{\pi|G|/8}$ [Knu75, Ex. 3.1.12] [FO90].

If $h_\mu = h_{\mu+\lambda}$ for some $\mu \in \mathbb{N}_0$, $\lambda \in \mathbb{N}$, we also have $h_k = h_{k+\lambda}$ for every $k \geq \mu$. Our strategy is that for a given set S of generators we define a sequence $(h_k)_{k\in\mathbb{N}_0}$ such that every match $h_k = h_i$ with $i < k$ yields a non-trivial relation on S. For this, the sequence will be defined such that for each term h_k we also know an exponent vector $\vec{y}_k$ with $S^{\vec{y}_k} = h_k$. Then, if $h_k = h_i$, a relation is given by $\vec{y}_k - \vec{y}_i$.

Hence, the function F we use to define the sequence (h_k) has to satisfy the following requirements. First, it must enable us to easily keep track of the exponent vectors corresponding to the terms of the sequence. Second, it should produce sequences with similar values for λ and μ like a random mapping does. It is this second requirement that motivated the use of the equidistributing function $f : G \to \{1, \dots, 20\}$ and, in particular, the choice of the size of the image of f. It has been shown under certain assumptions [SS85] that the image of f should consist of at least 8 elements. The size 20 has been chosen empirically. See Section 3 for a further discussion.

We now describe how we compute the j-th relation ($j \in \{1, \dots, l\}$). For this computation we only use the generators $g_1, \dots, g_j$. Set

$$S_j = (g_1, \dots, g_j), \qquad 1 \leq j \leq l.$$

We will define a sequence $(h_k) \subset \langle S_j \rangle$. For this, we use an *exponent bound* E. We set $E = 10$ at the beginning of the computation of the first relation, and increase this bound dynamically following a strategy we explain below. To initialize the sequence (h_k), let $e \in_R \{1, 2, \dots, E\}$. We set

$$\vec{y}_0 = (0, \dots, 0, e),$$

where the last position is the j-th position, and

$$h_0 = {g_j}^e (= {S_j}^{\vec{y}_0}).$$

Then we create a *list of exponent vectors* containing 20 vectors

$$\vec{e}_s = (e_{1s}, \dots, e_{js}), \quad e_{is} \in_R \{1, 2, \dots, E\},\ 1 \leq i \leq j,\ 1 \leq s \leq 20,$$

which are used to compute a *list of multipliers* containing 20 group elements $M_1, \dots, M_{20}$ formed by the rule

$$M_s = {S_j}^{\vec{e}_s}, \quad 1 \leq s \leq 20.$$

The successive terms of the sequences (h_k) and $(\vec{y}_k)$ are then computed as follows. We take the equidistributing function $f : G \to \{1, \dots, 20\}$ and define

$$F : G \to G, \quad F(h) = h * M_{f(h)}, \quad h \in G. \tag{2.2}$$

Then we set

$$h_{k+1} = F(h_k) \quad \text{and} \quad \vec{y}_{k+1} = \vec{y}_k + \vec{e}_{f(h_k)}, \quad k \in \mathbb{N}_0. \tag{2.3}$$

We recursively compute $h_1, h_2, h_3, \dots$ and $\vec{y}_1, \vec{y}_2, \vec{y}_3, \dots$ until we have found a match $h_k = h_i$ with $i < k$. For the search of such a match we use the procedure COMPARE-AND-ADJUST, which we describe in Section 4. If this procedure has found a match $h_k = h_i$ with $i < k$, we have found a non-trivial relation on S, which is

$$\vec{b}_j = ((\vec{y}_k - \vec{y}_i) \circ \underbrace{(0, \dots, 0)}_{l-j})^T,$$

where $\circ$ denotes the concatenation of vectors. In particular, we have $b_{jj} > 0$. If $b_{jj} \neq 1$, we call the procedure MINIMIZE to find a j-th *minimal relation* on S. By this we mean a relation $\vec{x} = (x_1, \dots, x_j, 0, \dots, 0)$ on S whose j-th component, x_j, is the smallest positive integer such that $g_j{}^{x_j}$ belongs to the subgroup $\langle S_{j-1} \rangle$ of G generated by $\{g_1, \dots, g_{j-1}\}$. We store this minimal relation as the j-th vector of the basis B.

It remains to describe the dynamic handling of the exponent bound E. First observe that if $E \geq |G|$ and $\vec{e} \in_R \{1, \dots, E\}^j$, then $S_j{}^{\vec{e}}$ is an (at least almost) randomly chosen element of $\langle S_j \rangle$. Second, we prove in Section 6 that in our implementation, the number of iterations until the procedure COMPARE-AND-ADJUST finds a match is bounded by $1.25 \max(\lambda/2, \mu) + \lambda$ (Remark 6.2). Therefore, if $E \geq |G|$, then according to Conjecture 6.2 we may assume that in (almost) all cases a match is found within $5\sqrt{|G|}$ iterations (Remark 6.4). This leads to the following strategy. As already said, in the beginning we set $E = 10$ and start to compute the first relation. After each successless call of the procedure COMPARE-AND-ADJUST we check whether the number of iterations is larger than $5\sqrt{E}$. If this is the case, we conclude that the actual exponent bound has been chosen too small. We interrupt the computation of the relation, square the exponent bound, compute new lists of multipliers and exponent vectors and compute a new sequence (h_k) until a match is found or the number of iterations exceeds $5\sqrt{E}$. The final exponent bound with which our algorithm has computed the j-th relation is then used as initial exponent bound for the computation of the $(j+1)$-th relation. Note that it is not essential that the final exponent bound E is a tight bound on the group order, or on the order of the current subgroup $\langle S_j \rangle$ in which the computation of the j-th relation takes place. This bound determines the size of the exponents of the multipliers and, consequently, the sizes of the exponent vectors corresponding to the terms h_k. Both precomputing the multipliers and administering the exponent vectors have computational complexity $O(\log E)$.

To speed-up our algorithm in practice, for the computation of the j-th relation we use only those generators g_i in $\{g_1, \dots, g_j\}$ that really enlarge the subgroup $\langle S_{i-1} \rangle$, i.e. for which $b_{ii} > 1$. For the bookkeeping of the generators g_i with $b_{ii} = 1$ we use the set T where we store the respective indices.

Algorithm 2.1 determines the Hermite normal form basis B for the relation lattice $L(S)$.

Algorithm 2.1.

This algorithm computes the Hermite normal form basis for the lattice of relations on a generating sequence for a finite abelian group.

INPUT: A generating sequence $S = (g_1, \dots, g_l)$ of G, an equidistributing function $f : G \to \{1, \dots, 20\}$.
OUTPUT: A basis $B = (\vec{b}_1, \dots, \vec{b}_l)$ of the lattice of relations on S in upper triangular form.

```
$B = ()$, $T = \emptyset$, $E = 10$
for ($j = 1, \dots, l$) do
  while (match not found) do
    for ($s = 1, \dots, 20$) do                           /* compute list of
      $M_s = 1$                                            multiplier and
      for ($i = 1, \dots, j$; $i \notin T$) do              exponent vectors. */
        choose random number $e$, $1 \le e \le E$
        $M_s = M_s * g_i^e$
        $e_{is} = e$
      od
    od
    choose random number $e$, $1 \le e \le E$
    $h = {g_j}^e$                                          /* Initialize the
    $\vec{y} = (0, \dots, 0, e)$                              sequences */
    $k = 0$                                               /* $k$ = current index */
    Initialize COMPARE-AND-ADJUST with $h$, $\vec{y}$
    while ((match not found) and ($k < 5\sqrt{E}$)) do     /* Compute the
      segment = $f(h)$                                     successive terms
      $h = h * M_{\text{segment}}$                               of the sequences */
      $\vec{y} = \vec{y} + \vec{e}_{\text{segment}}$
      $k = k + 1$
      COMPARE-AND-ADJUST
        Input: $h$, $\vec{y}$, $k$
        Output: $\vec{y}^* \ne \vec{y}$ such that $S^{\vec{y}-\vec{y}^*} = 1$,
         if match has been found
    od
    if (($k \ge 5\sqrt{E}$) and (match not found)) then      /* take larger
      $E = E^2$                                            exponent bound */
    fi
  od
  $\vec{b}_j = ((\vec{y} - \vec{y}^*) \circ (0, \dots, 0))^T$
  if ($b_{jj} \ne 1$) then
    MINIMIZE
      Input: relation $\vec{b}_j$, relation matrix
        $B = (\vec{b}_1, \dots, \vec{b}_{j-1})$
      Output: minimized relation $\vec{b}_j$
  fi
  $B = (B, \vec{b}_j)$
  if ($b_{jj} == 1$) then                                /* i.e., $g_j$ does not
    $T = T \cup \{j\}$                                     enlarge the group */
  fi
od
return ($B$)
```

Remark 2.1. Note that the number 5 in the conditions "$k < 5\sqrt{E}$" and "$k \geq 5\sqrt{E}$" is actually a parameter, which depends on the parameters chosen within the procedure COMPARE-AND-ADJUST. See Section 6, Remark 6.5 for the general case.

Remark 2.2. Algorithm 2.1 can easily be adapted to run on a distributed system. With only a little loss of efficiency the minimization of the relations can be shifted to that moment in the algorithm when *all* relations have been computed. Thus, all relations can be computed completely independent from one another.

3. RANDOM WALKS IN FINITE ABELIAN GROUPS

For a finite set G, Knuth [Knu75] has analyzed the average behavior of sequences (h_k) in G defined by

$$h_0 \in_R G, \quad h_{i+1} = F(h_i), \; i = 0, 1, 2, \dots ,$$

under the condition that F is a random mapping. Then the expected values for the length of the period, λ, and the length of the non-periodic segment, μ, are close to $\sqrt{\pi|G|/8}$. We want to make use of this fact in our algorithm, so we have to make sure that the function F we are using for our recursion generates a sufficiently randomized sequence, in the sense that the expected values for λ and μ do not differ too much from the random case.

In this section we explain why we choose the function F defined in (2.2) for our algorithm and why we are convinced that this choice fulfills our needs. In particular, we explain the role of the equidistributing function f in this context. We define two equidistributing functions we use in our implementation for class groups of imaginary quadratic orders. We propose a prototype of an equidistributing function for the general use, thereby motivating why the assumption of the existence of an equidistributing function should not pose a problem for concrete implementations.

Our choice of F is based on the following observation. Given a mapping $F : G \to G$, we can write it as

$$F : G \to G, \qquad F(a) = a * M(a), \qquad a \in G, \tag{3.1}$$

namely, using $M : G \to G$, $M(a) = a^{-1} * F(a)$. This correspondence $M \leftrightarrow F$ induces a bijection on the set of all mappings from G to G to itself. So F is chosen at random if and only if M is chosen at random. Hence, we may define F by means of a mapping $M : G \to G$ and according to (3.1). Our aim is to define this mapping M so that it is as space and run-time efficient as possible. We follow a method that has already been successfully applied by Schnorr and Lenstra [LS84] in the case of cyclic groups. The idea is to restrict the image of M to a small set of elements of G, say $M_1, \dots , M_r$. Then we set

$$F(a) = a * M_{f(a)}, \qquad a \in G,$$

with some function $f : G \to \{1, \dots , r\}$. Observe that for $i, k \in N$ we have

$$h_k = h_i \iff h_{k-1} = h_{i-1} M_{f(h_{i-1})} M^{-1}_{f(h_{k-1})}.$$

Thus, the more distinct multipliers M_s we use, and the better the values $f(a)$ $(a \in G)$ are distributed over the set $\{1, \dots , r\}$, the sooner matches $h_k = h_i$ $(i < k)$ occur! This is where our concept of the equidistributing function comes into action.

If we use such a function F in our recursion, how many terms $M_1, \dots , M_r$ are necessary to generate a sufficiently randomized sequence (h_k)? Sattler and Schnorr [SS85] conducted a theoretical investigation of this question in case of a cyclic group G. They concluded that if $M_1, \dots, M_r \in G$ are randomly chosen and $f : G \to \{1, \dots, r\}$ is a pseudo-random function, then every $r \geq 8$ will do. However, in practice we cannot apply their result, since the constants hidden in their asymptotic bounds are too large in comparison with the group orders with which we deal.

So we let experience decide and state that $r = 20$ works very well, using randomly chosen multipliers $M_1, \dots, M_{20}$, and an equidistributing function $f : G \to \{1, \dots, 20\}$ to determine which multiplier is used for which group element.

We are convinced that this method of computing the sequence (h_k) generates a sufficiently randomized sequence. However, we cannot prove this, so for our complexity analysis in Section 6 we have to state this conviction as conjecture (see Conjecture 6.1).

How should one define the equidistributing function for a concrete implementation? We suggest using the scheme of multiplicative hashing. Let us first explain this in the case of class groups of imaginary quadratic orders. Let $\mathcal{O}_\Delta$ denote the imaginary quadratic order of discriminant Δ. Let Cl_Δ denote the class group of $\mathcal{O}_\Delta$. Every ideal class in Cl_Δ is represented by a uniquely determined pair of integers (a, b), and we have $|b| \leq \sqrt{|\Delta|/3}$ (cf. [Coh93, p. 227]). Let p be the smallest prime number larger than $2|\Delta|^{1/4}$. We define

$$H : \mathrm{Cl}_\Delta \to [0, 1), \qquad (a, b) \mapsto H((a, b)) = b/p \bmod 1,$$

where $c \bmod 1$ denotes the (non-negative) fractional part of c, namely $c - \lfloor c \rfloor$. Then we define

$$f : \mathrm{Cl}_\Delta \to \{1, \dots, 20\}, \qquad (a, b) \mapsto \lfloor H((a, b)) \cdot 20 \rfloor + 1. \tag{3.2}$$

In other words,

$$f((a, b)) = d \iff \frac{b \bmod p}{p} \in \left[\frac{d-1}{20}, \frac{d}{20}\right), \quad d = 1, \dots, 20.$$

The choice of the size of the prime p in this definition is based on tests with samples of different primes and different discriminants, combined with the knowledge of the upper bound on $|b|$ as given above. Extensive tests support our assumption that this definition leads to an equidistributing function. For all tests we made, 20 as O-constant would work.

Let us also consider the function

$$H^* : \mathrm{Cl}_\Delta \to [0, 1), \qquad (a, b) \mapsto H^*((a, b)) = (A \cdot b) \bmod 1, \tag{3.3}$$

with A being approximately $(\sqrt{5} - 1)/2$ (i.e., the golden ratio). Note that it is sufficient to compute A with a precision of $\lceil \log_{10} \sqrt{|\Delta|/3} \rceil + 3$ digits, since $|b| \leq \sqrt{|\Delta|/3}$. From the theory of multiplicative hash functions we know [Knu73] that among all numbers between 0 and 1, choosing A as a rational approximation of $(\sqrt{5} - 1)/2$ with a sufficiently large denominator (i.e., in comparison with the input size) leads to the most uniformly distributed hash values, even for non-random inputs.

We use both H and H^* in our implementation. Both functions yield very good performances, which differ only slightly (see Table 2 in Section 7).

The function H^* is of particular interest for us since it provides a pattern for implementations of other groups. We suggest proceeding as follows. Given a group G whose elements are encoded as bit strings, we have to decide how to use this encoding to define the value corresponding to b of Definition (3.3), and what precision is required when computing A as approximation of $(\sqrt{5}-1)/2$. We then can define a suitable function $H : G \to [0,1)$, which we use to define f analogously to Definition (3.2). As a result we obtain a promising candidate for an equidistributing function.

Note that the requirements on the equidistributing function can be weakened. For instance, if the encodings of the group elements are not unique, we may define f on the set of all encodings instead of defining it on the group itself. This just implies that the expected length of the non-periodic segment and the period are not determined by the group order itself, but rather by the number of different encodings of all group elements. This observation could enable us to use our algorithm to compute, for instance, class groups of real quadratic orders.

4. Generating relations

As soon as our algorithm has found a match $h_k = h_i$ with $i < k$, it can compute a relation by taking the difference of the two exponent vectors $\vec{y}_k$ and $\vec{y}_i$ corresponding to h_k and h_i. Matches occur as soon as the sequence (h_k) has completed its first cycle. Then we have $h_k = h_i$ whenever $k - i$ is a multiple of the period.

In this section we show how our algorithm finds matches. We have improved a method used by Schnorr and Lenstra [LS84], which goes back to a family of cycle-finding algorithms developed by Brent [Bre80].

As before, let λ denote the period of the sequence (h_k) and μ the length of the non-periodic segment. Then for each $k \geq \mu$ we have that $h_{k+\lambda} = h_k$. Let $v > 1$ be such that $\lambda + \mu \leq v\mu$. Then, for each $k \geq \mu$ we have $k + \lambda \leq vk$. If we store some h_k with $k \geq \mu$ and compare all successive terms in the sequence with this term h_k, we are guaranteed to find a match $h_k = h_l$ with $k < l \leq vk$. We make use of this fact in our search algorithm.

Our method now is to store a fixed number $t \geq 2$ of terms $h_{\sigma_1}, \dots, h_{\sigma_t}$. Their indices $\sigma_1, \dots, \sigma_t$ have the property that $\sigma_{i+1} \approx R\sigma_i$ for some constant R, for $i = 1, \dots, t-1$ and σ_i large enough. In fact, since we are also interested in the corresponding exponent vectors, we store the terms and their exponent vectors as pairs $(h, \vec{y})$. We explain how we choose the indices $\sigma_1, \dots, \sigma_t$. First we choose a number $v > 1$. We set $\sigma_i = 0$ for all $i = 1, \dots, t$ and store $(h_0, \vec{y}_0)$. Then we compute $h_{\sigma_t+1}, h_{\sigma_t+2}, \dots$. For each newly computed term, we check whether it matches with one of the stored terms. If this is the case, we return the exponent vector corresponding to the stored matching term. Otherwise, we check whether $k \geq v\sigma_1$. If this is the case, we set $\sigma_i := \sigma_{i+1}$ for $i = 1, \dots, t-1$ and $\sigma_t = k$, i.e. we store $(h_k, \vec{y}_k)$ instead of $(h_{\sigma_1}, \vec{y}_{\sigma_1})$. It follows from the results in Section 6 (Lemma 6.1) that by this method we have $\sigma_{i+1} \approx R\sigma_i$ for some appropriate R and for all σ_i large enough.

In Section 6 we discuss the influence of the parameter v in more detail. For the moment, we just mention the following facts. The larger v is, the larger R is. A large value of R leads to a larger number of iterations until a match is found. Hence, from this point of view, the smaller v is, the better. On the other hand, the larger v is, the higher is the proportion of pairs (λ, μ) for which our algorithm finds the first match $h_{\sigma_i} = h_{\sigma_i+\lambda}$ that arises. It finds such a match if $\mu \leq \sigma_1$ and $\sigma_1 + \lambda \leq v \cdot \sigma_1$, and the last inequality holds whenever $\lambda \leq (v-1)\mu$. This means that the larger v is, the better. So there is some trade-off point determining the optimal choice for v. To compute this optimal theoretical value requires a thorough analysis involving the probability densities of λ and μ, which would exceed the scope of this paper.

Once again, we let experience decide. We made experiments with different rational parameters v between 1 and 5. We obtained the best performance when v was between 3 and 4, and we decided to use $v = 3$ in our implementation.

The number t of stored terms may be and should be kept rather small. It follows from Theorem 6.3 in Section 6 that the worst-case number of terms to be computed until a match is found decreases at most with $1 + O(1/t)$. (It follows from Lemma 6.1 that the same holds for the best-case number of terms.) On the other hand, the work needed to compare the current term with the stored terms increases linearly with t. The optimal number of stored terms depends on the ratio of the time needed for one equality check and the time needed for the computation of one term of the sequence.

In our experiments with class groups of imaginary quadratic fields we worked with t ranging from 2 to 10. The run-time differences were very small. On average, we got the best results with $t = 8$, so we chose this value for the experiments in Section 7.

Algorithm 4.1 is the algorithm to find a match.

Algorithm 4.1. COMPARE-AND-ADJUST

This algorithm searches for a match $h_k = h_i$ with the help of a set of t stored terms, and it administers this set. The parameters t and v have to be chosen in advance.

INPUT: The current terms h and $\vec{y}$, the current index k.
OUTPUT: A vector $\vec{y}^* \neq \vec{y}$ such that $S^{\vec{y}-\vec{y}^*} = 1$, if comparison was successful

```
if ($k = 0$) then                                                  /* Initialize */
  $(H_1, \vec{Y}_1) = (h, \vec{y}), \dots, (H_t, \vec{Y}_t) = (h, \vec{y})$
  $\sigma_1 = 0, \dots, \sigma_t = 0$
else
  if (there is $H_s$ such that $h = H_s$) then                     /* Compare */
    $\vec{y}^* = \vec{Y}_s$
    return ($\vec{y}^*$)
  fi
  if ($k \geq v\sigma_1$) then                                       /* Adjust */
    $(H_1, \vec{Y}_1) = (H_2, \vec{Y}_2), \dots, (H_{t-1}, \vec{Y}_{t-1}) = (H_t, \vec{Y}_t)$
    $(H_t, \vec{Y}_t) = (h, \vec{y})$
    $\sigma_1 = \sigma_2, \dots, \sigma_{t-1} = \sigma_t$
    $\sigma_t = k$
  fi
fi
```

5. Computing minimal relations

Given a relation $\vec{b}_j = (b_{1j}, \dots, b_{jj}, 0, \dots, 0)$ on $S = (g_1, \dots, g_l)$, we want to find a j-th minimal relation. By this, we mean that we compute a relation $\vec{x} = (x_1, \dots, x_j, 0, \dots, 0)$ with the property that x_j is the smallest positive integer such that ${g_j}^{x_j}$ belongs to the subgroup $\langle S_{j-1} \rangle$ of G generated by $\{g_1, \dots, g_{j-1}\}$.

The big advantage of computing minimal relations instead of working with the non-minimized relations is the following. Having generated an $l \times l$ upper triangular matrix of minimal relations, we can be sure that these relations really form a basis of the relation lattice $L(S)$. So l minimal relations are definitely enough to compute the structure of the group generated by S, which in general is not true for non-minimized relations. Our experiments show that minimizing a relation can be done very fast in comparison with the effort necessary to compute one relation.

In this section we describe how to minimize a relation.

We say that a relation $\vec{x} = (x_1, \dots, x_j, 0 \dots, 0)$ on S is *smaller* than a relation $\vec{y} = (y_1, \dots, y_j, 0 \dots, 0)$ on S if $0 < x_j < y_j$. If $\vec{x}$ is a j-th minimal relation, then $x_j \mid y_j$, since x_j is the order of $g_j\langle g_1, \dots, g_{j-1}\rangle$ in the factor group $\langle g_1, \dots, g_j\rangle / \langle g_1, \dots, g_{j-1}\rangle$. So when minimizing $\vec{b}_j$ we can restrict ourselves to those relations $\vec{x}$ that are smaller than $\vec{b}_j$ and for which $x_j \mid b_{jj}$. In fact, we will restrict ourselves to those relations that are smaller than $\vec{b}_j$ and for which $b_{jj}/x_j = p$ for some prime divisor p of b_{jj}. We then say that $\vec{x}$ is *p-smaller* than $\vec{b}_j$. If for some $p \mid b_{jj}$ there is no p-smaller relation of $\vec{b}_j$, we say that p is an *impossible divisor* of $\vec{b}_j$.

Our strategy is that we iteratively replace $\vec{b}_j$ by p-smaller relations. Such an iteration is ultimately finite, and it stops exactly when $\vec{b}_j$ is minimal.

We first consider the case $j = 1$. Then $\vec{b}_1 = (b_{11}, 0, \dots, 0)$, i.e. ${g_1}^{b_{11}} = 1$. For each prime divisor p of b_{11} we check whether ${g_1}^{b_{11}/p} = 1$. If this is the case for some prime divisor p, then we replace b_{11} by b_{11}/p and repeat the procedure. If ${g_1}^{b_{11}/p} \neq 1$ for all prime divisors p of b_{11}, then $\vec{b}_1$ is minimal.

Let us now minimize $\vec{b}_j$, under the condition that $\vec{b}_1, \dots, \vec{b}_{j-1}$ are already minimal. First, for $i = 1, \dots, j-1$ we reduce b_{ij} modulo b_{ii} by elementary column operations. Then we choose a prime p with $p \mid b_{jj}$ and we put $x_j = b_{jj}/p$. We show how to determine $x_1, \dots, x_{j-1}$. This is done recursively, starting with x_{j-1}. From the conditions $S^{\vec{x}} = 1$ and $S^{\vec{b}_j} = 1$ we get

$$S^{\vec{b}_j - p\vec{x}} = 1. \tag{5.1}$$

The j-th component of $\vec{b}_j - p\vec{x}$ is zero, as well as the $(j+1)$-th, $\dots$, l-th components. We conclude that the $(j-1)$-th component satisfies

$$b_{j-1,j} - p x_{j-1} \equiv 0 \bmod b_{j-1,j-1}, \tag{5.2}$$

since $\vec{b}_{j-1}$ is a minimal relation. We have either no or $\gcd(p, b_{j-1,j-1})$ different solutions of (5.2). Let x_{j-1} be such a solution. Let

$$m_{j-1} = \frac{b_{j-1,j} - p x_{j-1}}{b_{j-1,j-1}}.$$

From the equations $S^{m_{j-1}\vec{b}_{j-1}} = 1$ and (5.1) we get

$$S^{\vec{b}_j - p\vec{x} - m_{j-1}\vec{b}_{j-1}} = 1.$$

In the vector $\vec{b}_j - p\vec{x} - m_{j-1}\vec{b}_{j-1}$, all components from the $(j-1)$-th component on are zero. Therefore, x_{j-2} is one of the $\gcd(p, b_{j-2,j-2})$ different solutions of

$$b_{j-2,j} - px_{j-2} - m_{j-1}b_{j-2,j-1} \equiv 0 \bmod b_{j-2,j-2}.$$

By induction we get for $k = j-1, \dots, 1$ that x_k must satisfy the equation

$$b_{kj} - px_k - \sum_{i=k+1}^{j-1} m_i b_{ki} \equiv 0 \bmod b_{kk}, \tag{5.3}$$

where

$$m_i = \frac{b_{ij} - px_i - \sum_{n=i+1}^{j-1} m_n b_{kn}}{b_{ii}}. \tag{5.4}$$

Such a solution exists if and only if

$$b_{kj} - \sum_{i=k+1}^{j-1} m_i b_{ki} \equiv 0 \bmod \gcd(p, b_{kk}). \tag{5.5}$$

In this case, all solutions of (5.3) are given by the formula

$$x_k = x_{k,r_k} \equiv \frac{b_{kj} - \sum_{i=k+1}^{j-1} m_i b_{ki}}{\gcd(p, b_{kk})} \cdot c_k(p) + \frac{b_{kk}}{\gcd(p, b_{kk})} \cdot r_k \bmod b_{kk}, \tag{5.6}$$

where $c_k(p) \in \mathbb{Z}$ is such that

$$\gcd(p, b_{kk}) = p \cdot c_k(p) + b_{kk}a_k \tag{5.7}$$

for some integer a_k, and

$$r_k = 0, \dots, \gcd(p, b_{kk}) - 1.$$

So to compute $\vec{x}$, for $k = j-1, \dots, 1$ we recursively have to solve equation (5.3). For this, we use the following strategy. Having computed a $j-k$-tuple $x_{j-1}, \dots, x_k$ and the corresponding numbers $m_{j-1}, \dots, m_k$, we check whether (5.5) holds for $k-1$. If this is the case, we use (5.6) and (5.4) to compute a solution x_{k-1} and the corresponding m_{k-1} and check whether (5.5) holds for $k-2$. Otherwise, we choose another solution x_k according to (5.6), compute the corresponding m_k, and check again whether (5.5) holds for $k-1$ using the new solution tuple. If there is no other solution x_k, or if all those solutions have already unsuccessfully been tried for further computation, we have to replace x_{k+1} by another solution, and so forth. In the language of graph theory, we deal with a rooted tree of height $\le j-1$ and with root associated with $x_j = b_{jj}/p$. For $k = j-1, \dots, 2$, the solutions x_{k,r_k} of (5.3), given by (5.6), are associated with the children of x_{k+1}. So each node associated with a component x_{k+1} is either a leaf or has degree $\gcd(p, b_{kk})$. Hence, the computation described above means to find a path from the root to a leaf of depth $j-1$, and the traversing method of the tree is depth-first search. Whenever we have found a leaf of depth $j-1$, the corresponding path from the root to this leaf, $(x_j, \dots, x_1)$, yields exactly such a vector $(x_1, \dots, x_j, 0, \dots, 0)$ as we have been looking for. If we do not find any leaf of depth $j-1$, we conclude that there is no p-smaller relation of $\vec{b}_j$.

Suppose we have succeeded in computing a complete vector $\vec{x}$. Then we have to check whether $\vec{x}$ is indeed a relation, since the modular equations represent only necessary but not sufficient conditions on $\vec{x}$ to be a relation. If $S^{\vec{x}} = 1$, then we replace $\vec{b}_j$ by $\vec{x}$ and start the whole thing again with this new relation. Otherwise,

Algorithm 5.1. MINIMIZE

This algorithm computes a minimal relation on a set of generators.

INPUT: A set of generators $S = (g_1, \ldots, g_l)$, a relation $\vec{b_j} = (b_{1j}, \ldots, b_{jj}, 0, \ldots, 0)^T$ on S, an upper triangular matrix $B = (\vec{b_1}, \ldots, \vec{b}_{j-1})$ of minimal relations on S.
OUTPUT: A j-th minimal relation $\vec{b_j}$ on S.

```
$P = \{q; q$ prime, $q \mid b_{jj}\}$
$ID = \emptyset$                                              /* $ID$ = set of
for ($k = 1, \ldots, j-1$) do                                 impossible divisors */
  reduce $b_{kj}$ mod $b_{kk}$ by element. column
  oper.
od
while ($b_{jj}$ is not minimal) do
  if (there is $p$ in $P \setminus ID$ ) then
    $x_j = b_{jj}/p$
  else
    return ($\vec{b_j}$)                                      /* $b_{jj}$ is minimal */
  fi
  for ($k = 1, \ldots, j-1$) do
    compute a solution $c_k(p)$ of (5.7)
  od
  repeat
    use depth-first search to find the next
    $j-1$-tuple $x_{j-1}, \ldots, x_1$ satisfying the
    equations (5.3)
    if (such a $j-1$-tuple $x_{j-1}, \ldots, x_1$ has
      been found) then
      $\vec{x} = (x_1, \ldots, x_{j-1}, x_j, 0, \ldots, 0)$
      if ($S^{\vec{x}} == 1$) then                            /* $\vec{x}$ is a $p$-smaller
        $\vec{b_j} = \vec{x}^T$                               relation */
        if ($b_{jj} == 1$) then                               /* $b_{jj}$ is minimal */
          return ($\vec{b_j}$)
        fi
      fi
    fi
  until (no next $j-1$-tuple has been found, or
      a $p$-smaller relation has been found)
  if (no $p$-smaller relation has been found)
    then
    $ID = ID \cup \{p\}$
  fi
od
```

we try to compute a new complete vector, i.e., we continue the depth-first search of our tree to find another path $(x_j, \ldots, x_1)$. If we do not find such a path, we conclude that there is no p-smaller relation of $\vec{b_j}$.

Algorithm 5.1 is the algorithm to minimize a relation.

Remark 5.1. In all our experiments the greatest common divisors we encountered, i.e., the numbers of different solutions at each stage, were very small. In most cases, they were just 1, and in almost all cases, they were 1 or 2. Also, the number of checks whether $S^{\vec{x}} = 1$ was always small in comparison with the number of iterations needed to find a match $h_k = h_i$. See the tables in Section 7.

Remark 5.2. The disadvantage of the method presented above is that its worst-case complexity is worse than $O(|G|)$ due to the non-zero probability that one of the greatest common divisors that arises equals the group order. However, an average-case complexity analysis seems to be infeasible.

An alternative approach to get minimal relations is the following. First compute all l relations without minimizing any of them. Let B denote the resulting triangular matrix. Compute the Smith normal form $W := \mathrm{SNF}(B)$ of B and transformation matrices U and V such that $W = UBV$. Compute a new generating sequence $\tilde{S}$ of G such that the columns of W are relations on $\tilde{S}$. For this, let $X = (x_{ij})$ denote the inverse matrix U^{-1} of U. Then the new generating sequence is given by $\tilde{S} = (\tilde{g}_1, \dots, \tilde{g}_l)$, where

$$\tilde{g}_j = \prod_{i=1}^{l} g_i^{x_{ij}}, \qquad j = 1, \dots, l.$$

Note that with $W = (w_{ij})$, we have $\tilde{g}_j = 1$ for $1 \le j < \min\{i : w_{ii} > 1\}$.

Now factor the diagonal entries of the Smith normal form and compute p-smaller relations such that the diagonal entries of the resulting matrix $\tilde{W}$ are just the orders of the new generators. Only then apply the procedure MINIMIZE to each nontrivial column of $\tilde{W}$, using the new generating sequence $\tilde{S}$. As a result you get the minimized relations on $\tilde{S}$. The group structure is obtained by a second SNF computation.

This method has the advantage that its complexity analysis is simpler, since the procedure MINIMIZE is called at a later stage in the group structure algorithm. However, in practice the performance is much worse. This is due to the fact that the matrix entries become extremely large during the first SNF computation. Hence, the SNF computation usually takes more time than the whole minimizing procedure in the original minimizing algorithm. Even worse, minimizing the new relations on the new generating sequence is much more time-consuming than before. Therefore, we gave preference to the use of Algorithm 5.1, in spite of its bad theoretical properties.

6. Complexity

In this section we consider the computational complexity of Algorithm 2.1. We are not interested in the overall bit complexity, because it depends on the particular group with which we are working. Instead, we count the number of group operations such as multiplications and equality checks, the number of evaluations of the equidistributing function f, and the number of random numbers needed. Moreover, we determine the storage requirements in terms of group elements and l-dimensional vectors of integers. We ignore the time and space for doing index calculations.

We estimate the work necessary to compute l linearly independent relations on the generating sequence (Theorem 6.1). We do not consider the complexity of the procedure MINIMIZE, or the computational complexity of the SNF-computation.

Recall that if (h_k) is a sequence given by

$$h_0 \in G, \quad h_{i+1} = F(h_i), \ i = 0, 1, 2, \dots,$$

with some mapping $F : G \to G$, then λ denotes the period of this sequence and μ denotes the length of the non-periodic segment. Our complexity bounds also depend on the two parameters of Algorithm 4.1, which are the number v that appears in the condition $k \geq v\sigma_1$ of Algorithm 4.1, and t, the number of elements stored by Algorithm 4.1. In the following, we always use λ, μ, v and t in this sense. Note that to simplify our analysis, we restrict ourselves to integer values of v, i.e., $v \in \mathbb{N}_{\geq 2}$.

We base our complexity analysis on the following two conjectures.

Conjecture 6.1. *Let the sequence (h_k) be given by equations (2.2) and (2.3) in Section 2, with the initial term h_0 and the multipliers $M_1, \dots, M_{20}$ randomly chosen. Then the expected values of λ and μ are close to*

$$\sqrt{\frac{\pi |G^*|}{8}},$$

where G^* is the group generated by those generators taking part in the computation of (h_k) (via the multipliers).

Conjecture 6.2. *For almost all sequences (h_k) defined by equations (2.2) and (2.3) (with the initial term h_0 and the multipliers $M_1, \dots, M_{20}$ randomly chosen) we have*

$$2 \leq 3\sqrt{|G|} \quad \textit{and} \quad \lambda + \mu \leq 4\sqrt{|G|}.$$

In other words, given the set of all possible sequences (h_k), the number of sequences for which $2 > 3\sqrt{|G|}$ or $2 + \mu > 4\sqrt{|G|}$ is negligible.

Conjectures 6.1 and 6.2 hold if the sequences (h_k) behave like the random sequences analyzed by Knuth [Knu75]. They completely agree with our experiments.

The main result of this section is the following theorem:

Theorem 6.1. *Let G be a finite abelian group with $|G| \geq 4$. Let S be a generating sequence of G. Let $l = |S|$. Let*

$$V = \max\left(3\left(1 + \frac{1}{t} + \frac{1}{v-1}\right), 4\left(1 + \frac{v-1}{t}\right)\right).$$

Conjectures 6.1 and 6.2 imply that Algorithm 2.1 computes a regular $l \times l$ upper triangular matrix whose columns are relations on S, performing an expected number of at most

$$\left(l\left(2 + \frac{v-1}{t}\right)\sqrt{\frac{\pi}{8}} + V\right)\sqrt{|G|} + V|G|^{1/4}(\lceil \log\log_{10}|G|\rceil - 1)$$
$$+20l\,(2\log|G| + 1)\,(l + 1 + 2\lceil \log\log_{10}|G|\rceil)$$

group multiplications,

$$t \cdot \left[\left(l\left(2 + \frac{v-1}{t}\right)\sqrt{\frac{\pi}{8}} + V\right)\sqrt{|G|} + V|G|^{1/4}(\lceil \log\log_{10}|G|\rceil - 1)\right]$$

equality checks, and

$$\left(l\left(2 + \frac{v-1}{t}\right)\sqrt{\frac{\pi}{8}} + V\right)\sqrt{|G|} + V|G|^{1/4}(\lceil \log\log_{10}|G|\rceil - 1).$$

evaluations of the equidistributing function f. It uses at most

$$10l^2 + 11l + (20l + 1)\lceil \log\log_{10}|G|\rceil$$

random integers taken from the set $\{1, \dots, |G|^2\}$. Regardless of the validity of Conjecture 6.1, Algorithm 2.1 has to store

$$l + t + 22$$

group elements and the same number of vectors in $\{1, \dots, V|G|^{5/2}\}^l$.

Corollary 6.1. *Assume that the number of checks whether $S^{\vec{z}} = 1$ in the procedure* Minimize *does not depend on the group order. Then Theorem 1.1 holds.*

Proof. Put $v = 3$ and $t = 8$ in Theorem 6.1. □

Remark 6.1. Note that the storage requirements of the procedure Minimize are very small. There is just one group element and one l-dimensional vector to be stored, and a fixed number of integers. Only the size of the list of impossible divisors depends on the group order – it is $O(\log |G|)$.

To estimate the work required to compute the relations we combine

- the computation of one term in the sequence (h_k) and
- the check whether this term matches with one of the previously computed terms $h_{\sigma_1}, \dots, h_{\sigma_t}$

in one operation. This operation is called one *iteration.* So we have to estimate both the work for one iteration and the number of iterations until the algorithm finds a match. Moreover, we must take into account the work for the precomputation, i.e., the work for computing the lists of multipliers and exponent vectors.

Theorem 6.2. *To perform one iteration, Algorithm 2.1 executes one group multiplication, one evaluation of the equidistributing function, one vector addition in $\mathbb{Z}^l$ and t equality checks.*

Proof. This follows immediately from the description of Algorithm 2.1 in Section 2. □

Our next aim (Theorem 6.3) is to give an upper bound on the number of iterations until Algorithm 2.1 finds a match using Algorithm 4.1, under the condition that λ and μ are already given. For this, we need the following statement on the distribution of the numbers $\sigma_1, \dots, \sigma_t$ that do the bookkeeping of the indices of the stored terms in Algorithm 4.1.

Lemma 6.1. *Let v, $t \geq 2$ and such that $(v-1) \mid t$. Then in each situation of Algorithm 4.1 with $\sigma_1 \geq t/(v-1)$ we have that*

$$1 + \frac{v-1}{t + (t-1)(v-1)} \leq \frac{\sigma_{i+1}}{\sigma_i} \leq 1 + \frac{v-1}{t}, \qquad i = 1, \dots, t-1.$$

Proof. After the first call of Algorithm 4.1 by Algorithm 2.1, we have $\sigma_i = 0$ for $i = 1, \dots, t$. Hence, the next t numbers to be stored are $1, \dots, t$. Then, as long as $\sigma_1 \leq t/(v-1)$, we have

$$\sigma_i = u + i, \qquad i = 1, \dots, t,$$

for some appropriate integer $u \geq 0$. This can be proved by induction over u. If $\sigma_1 = t/(v-1)$, hence $\sigma_i = \sigma_1 + i - 1$ for $i = 1, \dots, t$, then the next t numbers to be stored are

$$v \cdot \sigma_1, \dots, v \cdot \sigma_t,$$

i.e.,

$$v \cdot \frac{t}{v-1}, v\left(\frac{t}{v-1}+1\right), \dots, v\left(\frac{t}{v-1}+t-1\right).$$

It follows by induction that the next t-tuples of stored numbers are given by

$$v^s \cdot \frac{t}{v-1}, v^s\left(\frac{t}{v-1}+1\right), \dots, v^s\left(\frac{t}{v-1}+t-1\right), \qquad s = 2, 3, \dots .$$

So it remains to consider the quotients

$$\frac{\sigma_{i+1}}{\sigma_i} = \frac{v^s\left(\frac{t}{v-1}+i\right)}{v^s\left(\frac{t}{v-1}+i-1\right)} = 1 + \frac{v-1}{t+(i-1)(v-1)}, \quad i = 1, \dots, t-1;\ s \in \mathbb{N}_0,$$

and

$$\frac{\sigma_{i+1}}{\sigma_i} = \frac{v^{s+1} \cdot \frac{t}{v-1}}{v^s\left(\frac{t}{v-1}+t-1\right)} = 1 + \frac{v-1}{t+(t-1)(v-1)}, \qquad s \in \mathbb{N}_0.$$

Taking the minimum and the maximum of this collection of quotients yields the desired result. □

Theorem 6.3. *Let v and t be as in Lemma 6.1. Given a periodic sequence (h_k) with λ and μ as above, the number of iterations until Algorithm 4.1 finds a match $h_k = h_i$ with $i < k$ is bounded above by*

$$\left(1 + \frac{v-1}{t}\right) \max\left(\frac{\lambda}{v-1}, \mu\right) + \lambda.$$

Proof. We first consider the case that $\lambda \leq (v-1)\mu$. There exists a number σ_i such that $\sigma_{i-1} < \mu \leq \sigma_i$. We denote this number by σ. Since

$$\frac{\sigma}{\mu} < \frac{\sigma_i}{\sigma_{i-1}} \leq 1 + \frac{v-1}{t},$$

we have

$$\sigma \leq \left(1 + \frac{v-1}{t}\right)\mu.$$

At some point in the algorithm, $\sigma_1 = \sigma$. Therefore, since $\sigma+\lambda \leq \sigma+(v-1)\mu \leq v\cdot\sigma$, Algorithm 4.1 detects the match $h_{\sigma+\lambda} = h_\sigma$, after at most

$$\left(1 + \frac{v-1}{t}\right)\mu + \lambda$$

iterations.

If $\lambda > (v-1)\mu$, let σ denote the smallest number σ_i such that $\frac{\lambda}{v-1} < \sigma_i$. Then

$$\sigma \leq \left(1 + \frac{v-1}{t}\right)\frac{\lambda}{v-1}.$$

Since $\sigma > \mu$, we have a match $h_{\sigma+\lambda} = h_\sigma$. Since $\sigma + \lambda < v \cdot \sigma$, Algorithm 4.1 finds this match, after at most

$$\left(1 + \frac{v-1}{t}\right)\frac{\lambda}{v-1} + \lambda$$

iterations.

Taking the maximum of these two bounds yields the desired result. □

Corollary 6.2. *Let v and t be as in Lemma 6.1. Conjecture 6.2 implies that the number of sequences (h_k) for which Algorithm 2.1 does* not *compute a relation within*

$$\sqrt{|G|}\max\left(3\left(1+\frac{1}{t}+\frac{1}{v-1}\right),4\left(1+\frac{v-1}{t}\right)\right)$$

iterations, is negligible.

Remark 6.2. For our implementation, where we used $v = 3$ and $t = 8$, Theorem 6.3 means that Algorithm 4.1 finds a relation after at most $1.25\max(\lambda/2, \mu) + \lambda$ iterations.

Remark 6.3. For our implementation we chose the parameters v and t for the procedure COMPARE-AND-ADJUST according to our experiments with a collection of different pairs (v, t). In Theorem 6.3 we gave a worst-case formula for the number of iterations until this procedure finds a match when using certain values for these parameters. As already mentioned in Section 4, one should do a probability analysis to find the theoretically optimal parameters v and t. This would exceed the scope of this paper.

Remark 6.4. For our implementation, Corollary 6.2 means that the number of sequences (h_k) for which Algorithm 2.1 does *not* compute a relation within $5\sqrt{|G|}$ iterations is negligible.

This observation is crucial for the dynamic handling of the exponent bound. It justifies our strategy, and it enables us to completely analyze the complexity of the precomputation and the computation of the l relations. It coincides with our experiments, where it never took more than $5\sqrt{|G|}$ iterations to compute one relation.

Remark 6.5. From Corollary 6.2 we immediately get the number that in general must replace the number 5 in the conditions "$k < 5\sqrt{E}$" and "$k \geq 5\sqrt{E}$" of Algorithm 2.1. It is given by

$$V := \max\left(3\left(1+\frac{1}{t}+\frac{1}{v-1}\right),4\left(1+\frac{v-1}{t}\right)\right).$$

Theorem 6.4. *Let Conjecture 6.2 hold. Let $|G| \geq 4$. To perform the precomputation, Algorithm 2.1 executes at most*

$$20l\,(2\log|G|+1)\,(l+1+2\lceil\log\log_{10}|G|\,\rceil)$$

multiplications in G. It uses at most

$$10l(l+1+2\lceil\log\log_{10}|G|\,\rceil)$$

random integers taken from the set $\{1,\dots,|G|^2\}$.

Proof. Let us first estimate the work assuming that the exponent bound does not change in the course of the computation of the relations. Then, for each of the l relations, Algorithm 2.1 computes 20 multipliers. For the j-th relation, these multipliers are products of random powers of at most j distinct generators. Each powering requires at most $2\lfloor\log E\rfloor + 1$ multiplications. Hence, to compute one multiplier for the j-th relation, Algorithm 2.1 executes at most $j(2\lfloor\log E\rfloor + 2)$ multiplications. Therefore, to compute 20 multipliers each for $j = 1,\dots,l$ requires at most

$$20l(l+1)(\lfloor\log E\rfloor+1)$$

multiplications in G. The number of required random integers is bounded by

$$\sum_{j=1}^{l} 20 \cdot j = 10l(l+1).$$

Now let us consider the reality of Algorithm 2.1. Initially, we have $E = 10$. The exponent bound is successively squared during the computation of the relations until l relations are found. Conjecture 6.2 implies that for the final exponent bound E we have

$$E_{fin} \leq |G|^2, \tag{6.1}$$

since if E has been squared at least once, for the penultimate exponent bound $F = \sqrt{E_{fin}}$ the condition $k \geq V\sqrt{F}$ must have held, where k denotes the final number of iterations for the current relation, and V is as in Remark 6.5. Since $k \leq V\sqrt{|G|}$ (Corollary 6.2) we get $F \leq |G|$, which implies (6.1). From (6.1) we conclude that at most

$$\lceil \log \log_{10} |G| \rceil$$

squarings may happen during the whole computation. This is also the upper bound for the number of lists of multipliers that are computed in vain. Hence, due to the dynamic handling of the exponent bound, the precomputation requires at most

$$20l \lceil \log \log_{10} |G| \rceil \cdot (2 \lfloor \log E_{fin} \rfloor + 2)$$

additional multiplications and at most

$$20l \lceil \log \log_{10} |G| \rceil$$

additional random numbers. Summing up and using the estimate (6.1), we get the desired result. □

We are now able to prove Theorem 6.1.

Proof of Theorem 6.1. From Theorem 6.3 we get that the expected number of iterations until one relation is computed is bounded above by

$$\left(2 + \frac{v-1}{t}\right) \cdot \sqrt{\frac{\pi |G|}{8}},$$

provided that the exponent bound does not change during the computation. So we have to take into account the iterations which are done in vain because the computation of a sequence is interrupted due to an insufficiently large exponent bound. We already saw in the proof of Theorem 6.4 that such an interruption happens at most $\lceil \log \log_{10} |G| \rceil$ times. We call the interrupted sequences the *vain sequences*. According to Corollary 6.2 we assume that each relation is computed within $V\sqrt{|G|}$ iterations. Thus, in the worst case the exponent bound of the last vain sequence equals $|G| - 1$. This implies that the last vain sequence is computed at most until the $V\sqrt{|G|-1}$-th term, with V as defined in Remark 6.5. Then the penultimate and all preceding vain sequences are computed at most until the $V(|G|-1)^{1/4}$-th term. Hence, the expected number of iterations until l relations are computed is bounded above by

$$\left(l\left(2 + \frac{v-1}{t}\right)\sqrt{\frac{\pi}{8}} + V\right)\sqrt{|G|} + V|G|^{1/4}(\lceil \log \log_{10} |G| \rceil - 1).$$

Together with Theorems 6.2 and 6.4 we get the asserted expected bounds on the number of group multiplications, equality checks and evaluations of f. As for the number of random integers, note that outside the precomputation random numbers are only used to initialize the computation of the sequences (h_k). This happens at most $l + \lceil \log\log_{10}|G| \rceil$ times. Thus, together with Theorem 6.4 we get the asserted bound.

It remains to consider the storage requirements. As for the group elements to be stored by Algorithm 2.1, there are the l generators, the 20 multipliers, the current term of the current sequence, and the return element of the procedure COMPARE-AND-ADJUST. This procedure itself has to store t group elements. Overall, there are $l + 20 + 2 + t$ group elements to be stored. Together with each group element (except the generators), Algorithm 2.1 and the procedure COMPARE-AND-ADJUST store the corresponding exponent vector. In addition, the l computed relations must be stored. Hence, there are $l + 20 + 2 + t$ l-dimensional vectors to be stored. Their entries are bounded by the product of the final exponent bound and the maximal number of iterations with constant exponent bound. So they are bounded by $V|G|^{5/2}$. This completes the proof of Theorem 6.1. □

From the proof of Theorem 6.1 and Conjecture 6.2 we immediately get upper bounds on the number of group multiplications etc., not just expectations. We only have to replace the term $(2 + (v-1)/t)\sqrt{\pi/8}$ in Theorem 6.1 by V:

Corollary 6.3. *Conjecture 6.2 implies that Algorithm 2.1 computes a regular $l \times l$ upper triangular matrix of relations performing in almost all cases at most*

$$\begin{aligned}(l+1)V\sqrt{|G|} + V|G|^{1/4}(\lceil \log\log_{10}|G| \rceil - 1)\\ +20l\,(2\log|G| + 1)\,(l + 1 + 2\lceil \log\log_{10}|G| \rceil)\end{aligned}$$

multiplications in G.

Note that even if Conjecture 6.2 does not hold, Algorithm 2.1 always terminates, since every sequence (h_k) we use becomes periodic within $|G| + 1$ iterations, and we have $\lambda, \mu, \lambda + \mu \le |G|$. Hence, as soon as $\sqrt{E} \ge |G|$, our algorithm detects a match $h_k = h_i$ with $i < k \le 2.25|G|$.

7. EXPERIMENTAL RESULTS

Using the LiDIA system [LiD96], we implemented our algorithm for ideal class groups of imaginary quadratic orders. Recall that this algorithm computes the structure of the group generated by a set S that is assumed to be *given*. So, given a discriminant Δ, we took the ten prime ideals of smallest norm in the corresponding imaginary quadratic order and let S be set of the ten corresponding (not necessarily pairwise distinct) equivalence classes. Then we used our algorithm to compute the subgroup generated by these equivalence classes. Note that due to the heuristics of Cohen and Lenstra [CL83] it is very likely that S generates the whole ideal class group Cl_Δ of $\mathcal{O}_\Delta$, and not just a subgroup of Cl_Δ. (And this indeed is the case in all our examples.) Except where otherwise stated, we used as equidistributing function the function $f : \mathrm{Cl}_\Delta \to \{1, \dots, 20\}$, $f((a,b)) = \lfloor (b/p \bmod 1) \cdot 20 \rfloor + 1$, as described in Section 3. As parameters in the procedure COMPARE-AND-ADJUST we used $v = 3$ and $t = 8$, where, as explained in Section 4, t indicates the number of elements being stored, and v measures the range covered by the indices $\sigma_1, \dots, \sigma_t$. During the computation, we counted the number of iterations until ten relations

TABLE 1. Group structure computation – one example

Discriminant: $\Delta = -4(10^{30}+1)$
Generators: $\{(2,2)\ (3,2)\ (5,4)\ (11,6)\ (17,10)$
$(19,14)\ (41,22)\ (43,2)\ (53,8)\ (59,6)\}$

j	Exponents of generators in the minimal relations										Iterations	Exp. bound
	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}		
1	2	0	0	0	0	0	0	0	0	0	1	10
2	0	4591263001512	0	0	0	0	0	0	0	0	7546962	10^{16}
3	0	4203702660072	2	0	0	0	0	0	0	0	2594586	10^{16}
4	0	1297599936828	1	4	0	0	0	0	0	0	11931582	10^{16}
5	0	2052377770760	1	0	2	0	0	0	0	0	21088172	10^{16}
6	1	71469877288	0	3	0	1	0	0	0	0	11605265	10^{16}
7	0	67059950988	1	0	0	0	2	0	0	0	16714761	10^{16}
8	0	1312014944900	1	2	0	0	0	2	0	0	40309091	10^{16}
9	0	2642574281313	0	1	0	0	1	0	1	0	50777998	10^{16}
10	0	1343052169238	1	0	0	0	0	0	0	2	80639226	10^{16}
sum											243207644	

Group structure (given by the invariants): $G = [2,2,2,2,2,8,4591263001512]$
Group order (= class number): $|G| = 1175363328387072$ (15 digits)
$\frac{\text{sum(Iterations)}}{\sqrt{|G|}} \approx 7.1$
Total run time: 15days 9hrs 30min 6.81sec (on a SPARCstation 4)
of which: time to minimize: 58.75 sec

were found. We also counted the number of checks whether $S^{\vec{z}} = 1$ in the procedure MINIMIZE. We measured the total run time of the algorithm and the time it took to minimize all relations. We worked with two series of discriminants, given by $\Delta = -(10^n + 3)$ and $\Delta = -4(10^n + 1)$, $n \in \{2, 3, \dots, 30\}$. For discriminants with up to 17 digits, we executed Algorithm 2.1 100 times; for discriminants with more than 17 digits, we did it 10 times.

First we show in Table 1 the result of the group structure computation of the ideal class group with discriminant $\Delta = -4(10^{30}+1)$. This computation would not have been possible using Shanks' Baby-Step Giant-Step method, because of the storage requirements of Shanks' algorithm and the memory constraints of the machines with which we work. The generators are represented by uniquely determined pairs of integers (a, b). The rows under the title "Exponents of generators in the minimal relations" represent the 10 relations after the procedure MINIMIZE. In the column "Iterations" the j-th entry is the number of iterations necessary to compute the j-th relation. In the column "Exp. bound" the j-th entry is the exponent bound after having finished the computation of the j-th relation. The last row contains the sum of all iterations. The run times were taken on a SPARCstation 4.

Tables 2 and 3 show the minimal, average and maximal number of iterations to compute 10 relations in their first three columns. The next three columns show the number of checks whether $S^{\vec{z}} = 1$ in the procedure MINIMIZE. The next column shows the final exponent bound after the computation of the tenth relation. The penultimate column of Table 2 and the last column of Table 3 show the quotient of the average number of iterations and the square root of the group order. For the series $\Delta = -(10^n + 3)$, we repeated the computation using the equidistributing

TABLE 2. Iterations, checks to minimize, for $\Delta = -(10^n + 3)$.

n	G	Iterations			Checks (minmz)			exp.	$\frac{aveIt}{\sqrt{\lvert G\rvert}}$	$\frac{aveIt}{\sqrt{\lvert G\rvert}}$
		min	ave	max	min	ave	max	bnd.		
2	[5]	20	33	42	11	22	30	10	14.8	15.2
3	[4]	19	29	40	11	21	30	10	14.5	14.5
4	[12]	28	44	64	13	29	53	10^2	12.7	13.0
5	[39]	52	86	125	21	35	57	10^2	13.8	13.4
6	[105]	76	137	191	33	49	65	10^4	13.4	13.3
7	[706]	265	391	539	23	46	347	10^4	14.7	14.5
8	[1702]	376	595	922	23	48	97	10^4	14.4	14.3
9	[2,1840]	426	833	1256	27	54	83	10^4	13.7	13.3
10	[10538]	701	1329	1995	32	58	422	10^8	12.9	12.9
11	[31057]	1552	2527	3597	32	57	1026	10^8	14.3	14.5
12	[2,62284]	2859	4567	6902	39	79	640	10^8	12.9	12.7
13	[2,2,124264]	5209	9611	14093	41	64	86	10^8	13.6	13.8
14	[2,2,356368]	7515	13257	18492	40	59	88	10^8	11.1	10.9
15	[3929262]	16985	27867	39945	31	51	72	10^8	14.1	14.3
16	[12284352]	28597	45624	68632	32	52	76	10^8	13.0	13.5
17	[38545929]	51413	88168	122559	34	47	71	10^8	14.2	14.6
18	[102764373]	91491	137517	256903	38	50	65	10^{16}	13.6	14.1
19	[2,2,2,78425040]	202538	287486	348269	78	98	130	10^{16}	11.5	10.9
20	[2,721166712]	328910	522644	724270	50	76	144	10^{16}	13.8	13.6
21	[3510898632]	602216	826743	1071913	53	65	77	10^{16}	13.9	14.7
22	[2,2,2,1159221932]	761318	1073395	1429794	72	105	146	10^{16}	11.1	11.0
23	[2,16817347642]	1560717	2594912	3452080	54	66	81	10^{16}	14.1	15.5
24	[2,2,37434472258]	3932389	4355120	4640687	67	79	87	10^{16}	11.2	13.0
25	[2,245926103566]	6424991	8562256	11061946	52	61	75	10^{16}	12.2	
26	[2,656175474498]	12682353	14301324	17852763	54	62	75	10^{16}	12.5	
27	[3881642290710]	19499900	25751685	31130041	50	63	79	10^{16}	13.1	
28	[2,2,2,1607591023742]	31471362	41832563	49104574	81	101	117	10^{16}	11.7	
29	[2,17634301773068]	60308823	71532179	84711420	61	81	119	10^{16}	12.0	

function $f((a,b)) = \lfloor((A \cdot b) \bmod 1) \cdot 20\rfloor + 1$, with $A \approx (\sqrt{5}-1)/2$ as described in Section 3. Since the experimental results differ only slightly, we just list the corresponding quotients of the average numbers of iterations and the square roots of the group orders. They are shown in the last column of Table 2.

We see that our experimental results completely agree with the theoretical results summarized in Theorem 1.1. Here, it is most honest to look at the class groups corresponding to the discriminants $-(10+3)^7$, $-(10+3)^{11}$, and $-(10+3)^{17}$, since for these groups, the first generator already generates the whole class group, so that the computation of each relation takes place in the whole class group (and not just in a smaller subgroup).

Tables 4 and 5 show the minimal, average and maximal run times for the whole group structure computation in their first three columns. The next three columns show the minimal, average and maximal run times for the whole minimizing procedure. All run times were taken on a SPARCstation ULTRA170.

TABLE 3. Iterations, checks to minimize, for $\Delta = -4(10^n + 1)$.

n	G	Iterations			Checks (minmz)			exp.	$\frac{aveIt}{\sqrt{\lvert G\rvert}}$
		min	ave	max	min	ave	max	bound	
2	[14]	33	47	63	16	30	46	10^2	12.6
3	[2,2,10]	43	73	99	28	56	83	10^2	11.5
4	[4,40]	93	154	248	33	56	81	10^4	12.2
5	[2,230]	148	255	392	25	53	79	10^4	11.9
6	[2,516]	250	426	626	31	55	94	10^4	13.3
7	[2,1446]	419	706	1074	34	58	282	10^4	13.2
8	[4,4104]	733	1359	2079	39	63	113	10^8	10.6
9	[2,2,2,2,2560]	1406	2265	3164	68	125	182	10^8	11.2
10	[2,2,48396]	2699	4988	7066	44	84	194	10^8	11.3
11	[2,2,2,56772]	4684	7509	11955	55	101	170	10^8	11.1
12	[2,4,117360]	5408	11137	18185	46	71	110	10^8	11.5
13	[2,2,742228]	11656	21389	30510	31	67	110	10^8	12.4
14	[2,2,4,1159048]	26099	42194	64341	52	83	119	10^8	9.8
15	[2,2,2,2,2,4,257448]	22533	43956	63994	163	310	634	10^8	7.7
16	[2,2,2,2,11809616]	90283	118944	167257	96	135	180	10^{16}	8.7
17	[2,2,2,46854696]	185550	237291	311824	73	105	148	10^{16}	12.2
18	[2,2,264135076]	248943	405015	561785	59	69	90	10^{16}	12.5
19	[2,1649441906]	559116	800177	1170851	48	66	83	10^{16}	13.9
20	[2,2,2,1856197104]	674526	1037680	1460502	71	89	101	10^{16}	8.5
21	[2,2,2,2,2,2,678293202]	1546123	1810912	2013223	221	242	265	10^{16}	8.7
22	[2,2,2,19870122100]	3379721	3650074	4007603	93	113	132	10^{16}	9.1
23	[2,2,2,2,23510740696]	4891053	6210273	7408870	115	134	152	10^{16}	10.1
24	[2,4,144373395240]	12318015	12736451	13802496	63	79	93	10^{16}	11.9
25	[2,2,2,2,186902691564]	16884070	19749328	25105267	108	127	153	10^{16}	11.4
26	[2,4,2062939290744]	35914919	45882067	53558193	73	81	87	10^{16}	11.3
27	[2,2,2,2,2,2,596438010456]	32147266	46568150	53200468	185	242	283	10^{16}	7.5

We see that the method to minimize the relations in order to get a complete relation lattice is efficient, especially for large discriminants.

In order to test the efficiency of our dynamic handling of the exponent bound, we did the same experiments as above but using Algorithm 2.1 with a *fixed* exponent bound, which we initially already chose as an upper bound of the group order. That is, we chose $E \approx (\sqrt{|\Delta|}\ln|\Delta|)/\pi$, in accordance with the fact [Coh93, p. 290] that $|\mathrm{Cl}_\Delta| < 1/\pi \cdot \sqrt{|\Delta|} \cdot \ln|\Delta|$ for $\Delta < -4$. The corresponding results, with $\Delta = -(10^n + 3)$, are shown in Tables 6 and 7.

We see that it does not make much difference whether we know an upper bound on the group order or not, at least for this sample of discriminants, and we expect that this holds in general.

TABLE 4. Sample run times for $\Delta = -(10^n + 3)$.

n	G	time (total)			time (to minimize)		
		min	ave	max	min	ave	max
2	[5]	0.32s	0.52s	0.66s	0.04s	0.26s	0.45s
3	[4]	0.39s	0.51s	0.65s	0.12s	0.25s	0.39s
4	[12]	0.77s	1.11s	1.73s	0.11s	0.34s	0.51s
5	[39]	0.71s	1.42s	1.59s	0.26s	0.44s	0.59s
6	[105]	2.99s	3.74s	3.93s	0.46s	0.67s	0.82s
7	[706]	2.58s	2.95s	4.0s	0.35s	0.69s	1.72s
8	[1702]	4.21s	4.58s	4.99s	0.53s	0.82s	1.27s
9	[2,1840]	3.80s	4.25s	4.70s	0.49s	0.97s	1.34s
10	[10538]	5.97s	9.64s	16.03s	0.79s	2.15s	8.68s
11	[31057]	10.32s	11.46s	29.55s	1.56s	2.41s	20.51s
12	[2,62284]	18.05s	20.07s	35.55s	1.89s	3.18s	19.06s
13	[2,2,124264]	19.20s	22.52s	26.34s	2.07s	2.94s	3.88s
14	[2,2,356368]	26.86s	31.43s	36.08s	2.34s	3.27s	4.42s
15	[3929262]	32.92s	43.31s	54.53s	2.44s	3.63s	4.84s
16	[12284352]	41.33s	57.25s	1m19.44s	2.43s	3.58s	4.94s
17	[38545929]	1m7.32s	1m45.71s	2m22.06s	2.79s	4.16s	5.55s
18	[2,2,264135076]	5m43.20s	8m6.20s	10m23.01s	6.66s	8.89s	11.47s
19	[2,2,2,78425040]	4m22.28s	5m57.59s	7m5.45s	7.38s	10.17s	12.56s
20	[2,721166712]	7m7.72s	10m43.21s	14m26.02s	7.75s	10.15s	17.43s
21	[3510898632]	12m38.45s	17m2.41s	21m57.59s	7.91s	9.41s	11.0s
22	[2,2,2,1159221932]	16m40.45s	23m7.50s	30m25.08s	10.65s	15.24s	20.11s
23	[2,16817347642]	34m20.94s	56m30.12s	1h14m45.13s	9.05s	10.94s	12.74s
24	[2,2,37434472258]	1h30m31.16s	1h40m50.31s	1h47m33.80s	12.31s	13.96s	15.63s
25	[2,245926103566]	2h27m49.11s	3h17m15.25s	4h13m55.91s	9.31s	11.27s	13.26s
26	[2,656175474498]	5h2m28.51s	5h43m34.55s	7h8m54.02s	11.68s	12.65s	14.09s
27	[3881642290710]	9h34m11.65s	11h28m26.99s	12h44m30.40s	13.50s	14.80s	16.21s
28	[2,2,2,1607591023742]	13h24m22.42s	18h32m43.84s	22h3m3.59s	17.69s	21.83s	25.37s
29	[2,17634301773068]	1d2h47m35.75s	1d8h18m26.45s	1d14h59m50.84s	12.47s	18.94s	28.31s

TABLE 5. Sample run times for $\Delta = -4(10^n + 1)$.

n	G	time (total)			time (to minimize)		
		min	ave	max	min	ave	max
2	[14]	0.42s	0.71s	1.08s	0.09s	0.30s	0.45s
3	[2,2,10]	1.07s	1.42s	1.55s	0.28s	0.45s	0.57s
4	[4,40]	1.75s	3.55s	3.80s	0.43s	0.64s	0.81s
5	[2,230]	2.40s	3.46s	3.81s	0.38s	0.65s	0.93s
6	[2,516]	3.01s	3.41s	4.04s	0.46s	0.82s	1.31s
7	[2,1446]	3.31s	3.77s	5.04s	0.58s	0.89s	2.14s
8	[4,4104]	13.33s	14.39s	15.80s	1.47s	2.10s	3.11s
9	[2,2,2,2,2560]	9.28s	10.87s	12.48s	1.39s	2.51s	3.76s
10	[2,2,48396]	15.36s	17.29s	21.13s	2.05s	3.35s	6.56s
11	[2,2,2,56772]	20.61s	23.56s	28.56s	2.28s	4.05s	6.13s
12	[2,4,117360]	23.38s	28.56s	34.69s	2.43s	3.60s	5.11s
13	[2,2,742228]	23.69s	33.43s	40.57s	2.17s	3.68s	5.85s
14	[2,2,4,1159048]	49.75s	1m6.16s	1m29.69s	2.78s	4.63s	6.76s
15	[2,2,2,2,2,4,257448]	48.99s	1m9.26s	1m40.08s	6.33s	13.68s	30.86s
16	[2,2,2,2,11809616]	2m7.68s	2m51.75s	3m33.90s	9.24s	12.71s	17.38s
17	[2,2,2,46854696]	3m59.50s	4m54.08s	6m11.14s	7.36s	10.42s	13.95s
18	[102764373]	1m58.94s	2m49.14s	5m3.14s	3.34s	4.75s	6.21s
19	[2,1649441906]	11m39.95s	16m21.83s	23m29.17s	6.45s	8.67s	10.37s
20	[2,2,2,1856197104]	14m45.22s	22m5.77s	30m50.56s	10.35s	12.16s	14.0s
21	[2,2,2,2,2,2,678293202]	34m17.85s	39m53.37s	44m8.20s	28.12s	29.26s	30.81s
22	[2,2,2,19870122100]	1h15m30.58s	1h21m20.47s	1h29m11.20s	14.50s	17.56s	20.99s
23	[2,2,2,2,23510740696]	1h54m10.96s	2h23m30.62s	2h51m45.95s	17.12s	18.59s	19.48s
24	[2,4,144373395240]	5h10m28.88s	5h24m25.41s	5h53m8.26s	11.82s	14.0s	15.96s
25	[2,2,2,2,186902691564]	6h49m11.30s	7h54m34.31s	9h57m15.03s	18.77s	21.63s	26.27s
26	[2,4,2062939290744]	15h3m50.03s	19h16m54.82s	22h23m43.79s	15.32s	16.48s	18.06s
27	[2,2,2,2,2,2,596438010456]	14h18m26.15s	20h36m24.53s	23h37m24.21s	30.97s	41.07s	50.04s

TABLE 6. Iterations, checks to minimize for $\Delta = -(10^n + 3)$; exponent bound $E \approx (\sqrt{|\Delta|} \ln |\Delta|)/\pi$.

n	G	Iterations			Checks (minmz)			exp.	$\frac{aveIt}{\sqrt{\lvert G\rvert}}$
		min	ave	max	min	ave	max	bound	
2	[5]	21	33	44	19	30	47	10^2	14.8
3	[4]	17	30	39	18	30	47	10^2	15
4	[12]	24	44	62	26	40	61	10^3	12.7
5	[39]	57	84	111	21	43	62	10^4	13.5
6	[105]	80	138	191	31	47	68	10^4	13.5
7	[706]	253	390	502	20	42	118	10^5	14.7
8	[1702]	296	570	813	29	52	110	10^5	13.8
9	[2,1840]	502	821	1210	32	59	91	10^6	13.5
10	[10538]	719	1385	2083	31	58	521	10^6	13.5
11	[31057]	1702	2538	3801	28	44	68	10^7	14.4
12	[2,62284]	3208	4498	6232	30	52	87	10^7	12.7
13	[2,2,124264]	5312	9630	13412	40	68	307	10^8	13.7
14	[2,2,356368]	8640	13146	18299	38	57	79	10^8	11.0
15	[3929262]	15409	27173	39547	34	50	74	10^8	13.7
16	[12284352]	29107	44936	64740	36	56	72	10^9	12.8
17	[38545929]	54900	90017	137832	30	49	64	10^9	14.5
18	[102764373]	99294	144596	175773	39	52	67	10^{10}	14.3
19	[2,2,2,78425040]	252322	325712	388455	81	113	167	10^{10}	13.0
20	[2,721166712]	304446	503664	612263	62	78	113	10^{11}	13.3
21	[3510898632]	638289	853722	1190061	44	57	75	10^{11}	14.4
22	[2,2,2,1159221932]	855706	1155522	1396567	80	100	119	10^{12}	12.0
23	[2,16817347642]	1889992	2427885	2968866	51	61	76	10^{12}	13.2
24	[2,2,37434472258]	3839274	5128883	6466268	61	77	97	10^{13}	13.3

TABLE 7. Sample run times for $\Delta = -(10^n + 3)$; exponent bound $E \approx (\sqrt{|\Delta|} \ln |\Delta|)/\pi$.

n	G	time (total)			time (to minimize)		
		min	ave	max	min	ave	max
2	[5]	0.63s	0.79s	0.88s	0.19s	0.35s	0.46s
3	[4]	0.57s	0.74s	0.84s	0.16s	0.35s	0.46s
4	[12]	1.83s	1.97s	2.05s	0.33s	0.46s	0.54s
5	[39]	1.60s	1.83s	1.99s	0.34s	0.52s	0.64s
6	[105]	3.11s	3.26s	3.43s	0.45s	0.62s	0.77s
7	[706]	2.55s	2.95s	3.78s	0.40s	0.68s	0.82s
8	[1702]	5.41s	5.71s	6.31s	0.68s	0.98s	1.50s
9	[2,1840]	4.30s	4.83s	5.69s	0.65s	1.14s	1.55s
10	[10538]	6.32s	7.05s	13.38s	0.96s	1.55s	7.76s
11	[31057]	7.41s	8.11s	10.02s	1.07s	1.54s	2.10s
12	[2,62284]	14.82s	16.02s	17.28s	1.47s	2.28s	2.91s
13	[2,2,124264]	15.78s	19.41s	23.51s	1.62s	2.60s	5.32s
14	[2,2,356368]	25.69s	29.44s	33.62s	2.21s	3.07s	3.79s
15	[3929262]	29.85s	40.15s	51.14s	2.54s	3.30s	4.34s
16	[12284352]	39.28s	55.44s	1m16.23s	2.68s	3.75s	4.92s
17	[38545929]	1m10.12s	1m40.15s	2m37.12s	3.25s	4.31s	5.30s
18	[102764373]	2m7.08s	53.25s	3m25.91s	3.74s	4.68s	5.40
19	[2,2,2,78425040]	5m4.70s	6m20.79s	7m28.33s	5.74s	8.31s	12.41s
20	[2,721166712]	6m22.69s	10m5.38s	12m13.67s	6.18s	7.80s	10.40s
21	[3510898632]	13m52.55s	18m30.90s	25m27.44s	5.59s	7.02s	9.50s
22	[2,2,2,1159221932]	19m22.23s	25m45.83s	30m52.47s	10.78s	12.86s	14.86s
23	[2,16817347642]	39m59.22s	50m53.60s	1h2m7.28s	7.30s	8.40s	10.35s
24	[2,2,37434472258]	1h22m50.77s	1h50m40.66s	2h18m29.73s	9.06s	11.31s	14.34s

Acknowledgments

The author wishes to thank Johannes Buchmann and Michael J. Jacobson, Jr. for the fruitful discussions on the subject presented in this paper.

References

[BJT97] J. Buchmann, M.J. Jacobson Jr., and E. Teske. On some computational problems in finite abelian groups. *Mathematics of Computation*, 66:1663–1687, 1997. MR **98a:**11185

[Bre80] R.P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20:176–184, 1980. MR **82a:**10007

[CL83] H. Cohen and H.W. Lenstra, Jr. Heuristics on class groups of number fields. In *Number Theory*, Lecture notes in Math., volume 1068, pages 33–62. Springer-Verlag, New York, 1984. MR **85g:**10007

[Coh93] H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag, Berlin, 1993. MR **94i:**11105

[FO90] P. Flajolet and A.M. Odlyzko. Random mapping statistics. In *Advances in Cryptology - EUROCRYPT '89*, Lecture Notes in Computer Sci., volume 434, pages 329–354, Springer-Verlag, New York, 1990. MR **91h:**94003

[Knu73] D.E. Knuth. *The art of computer programming. Volume 3: Sorting and searching.* Addison-Wesley, Reading, Massachusetts, 1973. MR **56:**4281

[Knu75] D.E. Knuth. *The art of computer programming. Volume 1: Fundamental algorithms.* Addison-Wesley, Reading, Massachusetts, 1975. MR **51:**14624

[LiD96] LiDIA Group, Universität des Saarlandes, Saarbrücken, Germany. *LiDIA - A library for computational number theory, Version 1.2*, 1996.

[LS84] H.W. Lenstra, Jr. and C.P. Schnorr. A Monte Carlo factoring algorithm with linear storage. *Mathematics of Computation*, 43(167):289–311, 1984. MR **85d:**11106

[LT82] H.W. Lenstra, Jr. and R. Tijdeman, editors. *Computational methods in number theory*, volume 154/155 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1982. MR **84d:**10004

[McC90] K. McCurley. The discrete logarithm problem. In *Cryptology and Computational Number Theory*, Proc. Symp. Appl. Math., vol. 42, pages 49–74. American Mathematic Society, 1990. MR **92d:**11133

[Pol78] J.M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978. MR **58:**10684

[Sch82] R.J. Schoof. Quadratic fields and factorization. In Lenstra, Jr. and Tijdeman [LT82], pages 235–286. MR **85g:**11118

[Sha71] D. Shanks. Class number, a theory of factorization and genera. In *Proc. Symp. Pure Math. 20*, pages 415–440. AMS, Providence, R.I., 1971. MR **47:**4932

[Sho96] V. Shoup. Lower bounds for discrete logarithms and related problems. In Advances in Cryptology–Eurocrypt '97, Lectures Notes in Computer Sci., Volume 1233, pp. 256–266, Springer-Verlag, New York, 1997.

[SS85] J. Sattler and C.P. Schnorr. Generating random walks in groups. *Ann.-Univ.-Sci.-Budapest.-Sect.-Comput.*, 6:65–79, 1985. MR **89a:**68108

Technische Universität Darmstadt, Institut für Theoretische Informatik, Alexanderstrasse 10, 64283 Darmstadt, Germany

Current address: Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

E-mail address: `teske@cdc.informatik.tu-darmstadt.de`