EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

# Generic algorithms for bounded discrete logarithm vectors

van Roij, A.

*Award date:*
2010

Link to publication

**Disclaimer**
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

# Generic Algorithms for Bounded Discrete Logarithm Vectors

by

A. van Roij

Supervisor:

Berry Schoenmakers (TU/e)

Eindhoven, April 2007

# Contents


**1 Introduction** **3**
1.1 Background and Motivation . . . 3
1.2 Problem Description . . . 3
1.3 Related Work . . . 5
1.4 Road Map . . . 5

**2 Preliminaries** **7**
2.1 Ranking and Generation of Tuples . . . 7
2.2 The Birthday Paradox . . . 8
2.3 Random Walks and Checkpoints . . . 8
2.4 Hash Tables . . . 10
2.4.1 Description . . . 10
2.4.2 Abstract Data Type and Implementation . . . 12
2.5 Multi-base Exponentiation . . . 13
2.5.1 Exponentiation of Random Vectors . . . 13
2.5.2 Incremental Vector Changes . . . 14
2.5.3 An Amortized Variant for Testing . . . 15

**3 Algorithms** **16**
3.1 Heuristic Lower Bound . . . 16
3.2 Basic Algorithms . . . 17
3.2.1 Brute Force . . . 17
3.2.2 Precomputation . . . 17
3.3 Baby-step Giant-step . . . 17
3.3.1 Description . . . 17
3.3.2 Generalizations . . . 18
3.3.3 Optimizations . . . 19
3.4 Pollard-$\rho$ . . . 20
3.5 Pollard-$\lambda$ . . . 22
3.5.1 Description . . . 22
3.5.2 Analysis and Optimization . . . 23
3.5.3 Concurrent Pollard-$\lambda$ . . . 25

**4 Data Structure for Collision Search** **28**
4.1 Description . . . 28
4.2 Abstract Data Type and Implementation . . . 29
4.3 Example . . . 31
4.4 Efficiency . . . 31

**5 Advanced Algorithms** **33**
5.1 Precomputation for Pollard-$\lambda$ . . . 33
5.1.1 A simple algorithm . . . 33
5.1.2 An advanced algorithm . . . 34
5.2 A Hybrid of Baby-step Giant-step and Pollard-$\lambda$ . . . 36
5.2.1 Baby-Giant $\lambda$ . . . 36
5.3 Exploiting a priori Knowledge . . . 38
5.3.1 Constraints on coordinates . . . 38
5.3.2 Case: sum of coordinates known . . . 39

**6 Parallel Algorithms** **41**
6.1 Parallelization of Baby-step Giant-step . . . 41
6.2 Parallelization of Pollard-$\rho$ . . . 42
6.3 Parallelization of Pollard-$\lambda$ . . . 44
6.3.1 A First Technique . . . 44
6.3.2 An Advanced Technique . . . 45

**7 Conclusions** **48**
7.1 Useful Results . . . 48
7.2 Further Research . . . 48

**A Java Package** **51**
A.1 Discrete Logarithm Settings . . . 51
A.2 Class Hierarchy . . . 51
A.3 Class Description . . . 52
A.3.1 Peripheral . . . 52
A.3.2 Mathematics . . . 52
A.3.3 ReStore . . . 53
A.3.4 Setting . . . 53
A.3.5 Evaluator . . . 53
A.3.6 Algorithms . . . 53
A.3.7 Demonstration . . . 54

**B Description and Validation of Experiments** **55**
B.1 Introduction . . . 55
B.2 Pollard-$\lambda$ . . . 56
B.3 ReStore . . . 57
B.4 Pollard-$\lambda$ Attractors . . . 58
B.5 Advanced Parallel Pollard-$\lambda$ (not confirmed) . . . 58

# Chapter 1

# Introduction

## 1.1 Background and Motivation

In cryptography, modular exponentiation by an integer containing a message is often used as a one-way function. Computing the inverse function, that is, computing the discrete logarithm to a given base is supposed to be infeasible. For certain applications it is desirable that the discrete logarithm is relatively easy to compute. The question how to find discrete logarithms efficiently is called the *discrete logarithm problem* and it is the subject of this thesis.

A typical example for this problem is the electronic voting system proposed in [1], where an election result is encoded in a discrete logarithm. Since most elections need more than two options to vote for, we generally need a one-way function that encodes a vector of indices. The indices now are used as exponents to different bases, and by multiplying these exponentiation results, we obtain a one-way function for vectors. In [2] it is described how an election result can be put inside a discrete logarithm. The problem is how to retrieve the discrete logarithm vector efficiently in terms of time and space.

This problem raises more questions. The discrete logarithm is usually limited to a small subset of possible vectors, since the total number of votes is limited or even known exactly, it can be used for finding the election result. Also, an exit poll can help finding the discrete logarithm. A priori knowledge helps to reduce both time and space requirements for finding an election result, or generally, a discrete logarithm containing a vector. Parallelization is another possible way to speed up the finding of discrete logarithms.

The space requirements for finding discrete logarithms raise another question, since in [2] a variant of Baby-step Giant-step using vectors, needs space for about the square root of the total number of possible election results. Even in classroom size elections, this easily requires Gigabytes of memory[1]. One way of reducing this space, is to make a tradeoff between computation time and necessary space. Another way of reducing space is to develop new algorithms for finding the discrete logarithm.

## 1.2 Problem Description

The group generated by $g_1, g_2, ..., g_d$ is written $\langle g_1, g_2, ..., g_d \rangle$.

For $q > 2$ prime, let $(\mathbb{Z}_q, +, 0)$ be the additive group of integers mod $q$ and $(G_q, *, 1)$ be a multiplicative group of order $q$ such that $G_q$ is isomorphic to $\mathbb{Z}_q$. Let $g \in G_q \backslash \{1\}$. Because $q$ is prime, we have that $\langle g \rangle = G_q$.

[1] If 16 class-mates vote for one class-representative, then there are $16^{16} = 2^{64}$ election results. Suppose that all results are equiprobable. Using Baby-step Giant-step, we need space for $2^{64/2}$ results. If each result is identified by $|c|/2$ bits, we need $2^{64/2} 64/2 = 2^{37}$ bits, which is 16 Gigabyte. Because for every Giant step we have to do a lookup in memory, fast access is crucial. Note that for Baby-step Giant-step, the growth in memory is exponential.

**Definition 1.1** *Let $g$ of prime order $q > 2$ generate $G_q$ as above. Given that $h \in G_q$, the* discrete logarithm problem *is to find $z \in \mathbb{Z}_q$ such that $h = g^z$.*

For *integer intervals*, we have that $[0, c) = \{0, 1, 2, ..., c - 1\}$. Now restrict integer $z \in [0, c)$ for $c \in \mathbb{Z}_q$, where usually $c \ll q$.

**Definition 1.2** *Let $y \in [0, c)$ and $h = g^y \in G_q$. The* bounded discrete logarithm *is $y \in \mathbb{Z}_q$.*

In [1] the idea of working with generators $g_1, g_2, ..., g_d \in G_q$ is mentioned, each with bounded indices $v_i \in [0, c_i),\ i = 1, 2, ..., d$ such that we have a one way function that encodes the vector $(v_1, v_2, ..., v_d)$ as a product $g_1^{v_1} g_2^{v_2} ... g_d^{v_d} \in G_q$. The next definitions lead to an easy description of the problem.

**Definition 1.3** *We speak of a* bounded discrete logarithm vector problem*, if given $h = \prod_{i=1}^{d} g_i^{v_i}$, we have to find $v_i \in [0, c_i)$ , $i = 1, 2, ..., d$, where $c = \prod_{i=1}^{d} c_i \ll q$ vectors are possible.*

In Section 2.2 we show that for $c \ll \sqrt{q}$ each vector has unique $h$ in probability, such that we speak of a *one way* function[2].

**Definition 1.4** *Let $\mathbf{v} = (v_1, v_2, ..., v_d)$ and $\mathbf{g} = (g_1, g_2, ..., g_d)$. We write $\mathbf{g}^{\mathbf{v}} = g_1^{v_1} g_2^{v_2} ... g_d^{v_d}$.*

Note that for $g \in G_q$ there exist unique $r_i \in \mathbb{Z}_q$ such that $g_i = g^{r_i}$. From this it is immediate that $\mathbf{g}^{\mathbf{v}} = g^{\Sigma_{i=1}^{d} r_i v_i}$, which is a discrete logarithm problem. From an elegant proof by Schoenmakers[3] in [2], we know that the discrete logarithm vector problem for $v_i \in [0, q)$ *unbounded* (only bounded by the additive group order), is equivalent to the discrete logarithm problem. For a set of $c \ll q$ possible vectors, we have the *bounded discrete logarithm vector problem* with $v_i \in [0, c_i)$, that is different from the discrete logarithm vector problem. For our purpose, we also want this encoding to be a one way function.

With the previous definitions we can briefly formulate our problem.

*Given $h = \mathbf{g}^{\mathbf{v}}$ with $v_i \in [0, c_i)$ and $\prod_{i=1}^{d} c_i \ll q$, find $\mathbf{v}$ efficiently.*

By efficiently we mean by minimizing time and space, if necessary by a tradeoff. If possible, we also use a priori knowledge about $\mathbf{v}$. For parallel computers, we want to distribute the workload.

We will now explain how our problem setting is related to a broader context.

**Fact 1.5** *A* finite abelian group *is isomorphic to a direct product of cyclic groups $\mathbb{Z}_{p_1^{e_1}} \times \mathbb{Z}_{p_2^{e_2}} \times ... \times \mathbb{Z}_{p_d^{e_d}}$, where the primes $p_i$ are not necessarily distinct and the $e_i$ are positive integers.* ♡

Our setting reveals a weakness when applied to this structure. For example, consider the direct product of groups, each of different prime order $p_i$. Now $\langle g_i \rangle = G_{p_i},\ i = 1, 2, ..., d$, and we have a group structure isomorphic to $\mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times ... \times \mathbb{Z}_{p_d}$. Because $\gcd(p_i, p_j) = 1,\ i \neq j$ and $g_i^{p_i} = 1$, we have

$$(\mathbf{g}^{\mathbf{v}})^{p_2 p_3 ... p_d} = g_1^{(p_2 p_3 ... p_d) v_1} 1^{v_2} 1^{v_3} ... 1^{v_d} = g_1^{(p_2 p_3 ... p_d) v_1}$$

For $g_1' = g_1^{(p_2 p_3 ... p_d)}$ we have to find the discrete logarithm of $g_1'^{v_1}$, which is a major reduction. Analogously we solve the other $g_i'^{v_i}$ for $i = 2, 3, ..., d$. This so called *breaking of logarithms* is exploited in the Pohlig-Hellman algorithm. Compared to the general case of a finite abelian group, our cyclic group $G_q$ has the simplest structure of finite abelian groups. It is the computational complexity of finding $\mathbf{v},\ v_i \in [0, c_i)$ given $h = \mathbf{g}^{\mathbf{v}}$ and $c \ll q$ that makes our problem interesting.

[2] A *one way* function is not necessarily injective, but in many applications it can be used as such.

[3] Independently, Pfitzmann also gave a proof in her PhD thesis.

**Definition 1.6** *For $i \neq j$, let $g_i, g_j \in G_q$. Integer $r_{ij} \in \mathbb{Z}_q$ such that $g_i^{r_{ij}} = g_j$ is called the* relative discrete logarithm.

In [2] it is proved, that finding the discrete logarithm vector for $v_i \in [0, q)$ is as easy as the discrete logarithm problem. So for $z \in \mathbb{Z}_q$, it is computationally infeasible to find either relative discrete logarithms $r_i$ such that $g_i = g^{r_i}$ or equivalently, a discrete logarithm vector. However, for $v_i \in [0, c_i)$ , $i = 1, 2, ..., d$ and $c = \prod_{i=1}^{d} c_i \ll q$, finding the bounded discrete logarithm vector is different. In particular by Fact 2.5 in Section 2.2, it is very unlikely to find two vectors $\mathbf{v}_1 \neq \mathbf{v}_2$ from a set of $c \ll \sqrt{q}$ such that $\mathbf{g}^{\mathbf{v}_1} = \mathbf{g}^{\mathbf{v}_2}$. By a simple restriction[4] we allow randomness in the $g_i$, while we have that $\mathbf{g}^{\mathbf{v}}$ is injective for $v_i \in [0, c_i)$.

## 1.3 Related Work

In the past thirty years, only the case of one generator has been investigated thoroughly. The first algorithm that finds the discrete logarithm in $\Theta(\sqrt{c})$ time, is Baby-step Giant-step, attributed to Daniel Shanks. A disadvantage of this algorithm is, that it needs $\Theta(\sqrt{c})$ space such that even for moderate cases, computation needs too much space. In [4, §3], Pollard applies a so-called $\lambda$ algorithm to the case $z \in [0, c)$ and a solution based on heuristic assumptions takes $\Theta(\sqrt{c})$ multiplications. The amount of space is reduced to only $\Theta(\log(c))$ elements, such that even computers with a small space can determine the discrete logarithm in $\Theta(\sqrt{c})$ multiplications. Wiener and Van Oorschot in [5, §5] extend this $\lambda$ method to parallel computers, using *distinguished points*. In [6, §2], Shoup proves for a cyclic group of prime order $q$, and an injective function that maps each group element to a binary string, that $\Omega(\sqrt{q})$ element mapping operations[5] are needed to find the discrete logarithm. Unfortunately, Shoup does not mention lower bounds for space.

First in [1] and more explicitly in [2], Schoenmakers explains a Baby-step Giant-step algorithm for $d \geq 2$, and in this manuscript the term *discrete logarithm vector* is used for the first time[6]. Therefore, [2] is the starting point for this thesis.

## 1.4 Road Map

This thesis consists of four parts. An introductory part (Chapters 1, 2 and 3), an intermezzo with two useful data structures (Chapter 4), new developments (Chapters 5 and 6) and finally some conclusions and recommendations (Chapter 7).

Chapter 2 explains how tuples with different radix per coordinate are ranked and unranked. Then a cryptographic setting is presented, since we need a setting for testing algorithms. Then, after the Birthday Paradox, the use of different types of checkpoints for keeping track of random walks is explained. The next section is about hash tables. Finally, an efficient evaluation technique for multiplicative group elements is explained and more importantly, a fast evaluation technique for vectors is explained. Chapter 3 starts with a heuristic statement about a lower bound on the bounded discrete logarithm vector problem. It is followed by different algorithms to find the discrete logarithm that are generalized, optimized and combined with time-space tradeoffs. In Chapter 4 we explain an efficient data structure for collision search, as is the case in Baby-step Giant-step and Pollard-$\lambda$. It can be read without a complete understanding of previous chapters.

[4] Let $\tau_1 = 1$ and $\tau_i = \prod_{j=1}^{i-1} k_j$, where $k_j$ are chosen at random such that $k_j > c_j$ and $\tau_d < q$. Define $g_i = g^{\tau_i}$.

[5] These *mapping operations* are called *oracle queries*.

[6] Since [7], using a product of multiple generators and exponents from the same cyclic group is considered, but for other reasons than ours. In that publication, the term *Relaxed Discrete Logarithms* was used for a $\mathbf{v}$ satisfying $\mathbf{g}^{\mathbf{v}} = h$, but in a different context than we have. The term *Relaxed* was used for a presumed relaxation on restrictions. To date, we know that finding a discrete logarithm vector is as hard as finding a discrete logarithm. Because we even *restrict* $v_i \in [0, c_i)$, the term *Relaxed Discrete Logarithm* would be very counter intuitive in our setting. A few years later, in [8], the terminology *The Representation Problem* is used for finding a $\mathbf{v}$ satisfying $\mathbf{g}^{\mathbf{v}} = h$. This last terminology is also unclear, because more than one vector can represent the discrete logarithm modulo $q$.

In Chapter 5 an alternative for Pollard-$\lambda$ algorithm guarantees a collision within a narrow range of steps. Next, a new and time efficient algorithm for the bounded discrete logarithm vectors is proposed. In case of a priori knowledge about the solution vector, a heuristic statement gives a minimum amount of time and space. As a special case, a priori knowledge is worked out for a fixed sum of coordinates. Chapter 6 describes how to split up the workload over a multitude of processors. Finally, Chapter 7 contains conclusions and recommendations for further research.

# Chapter 2

# Preliminaries

To have some tools ready when explaining and modifying algorithms, we start by explaining subjects that are useful throughout this thesis. We use the settings and definitions of Section 1.2. To get started, the ranking and generation of vectors from a finite set is explained in Section 2.1. Next the Birthday Paradox is explained in Section 2.2, since it is essential to understanding the principle of collision search that is used for finding discrete logarithm vectors. After that, in Section 2.3, different ways of leaving markers on a random walk are shown. In Section 2.4, hash tables follow. Finally, in Section 2.5 we explain how evaluation of $\mathbf{g}^{\mathbf{v}}$ is done efficiently, by incremental changes of a vector rank.

## 2.1 Ranking and Generation of Tuples

In some cases we need to do an exhaustive search through all combinatorial objects from a universe. To order these objects, we can use functions that assign a *rank* to them. For certain problems, it is necessary that we generate these objects systematically. By using the inverse of ranking, we can visit all ranks and generate each object by *generation* of each rank. In [16], Knuth uses *rank* to identify a tuple by an ordinal number. Next to that, *generation* is defined in [15] for making an object for given rank. Also in [15], the term *traversal* denotes a strategy for *visiting* all objects successively.

**Example 2.1** A computer clock contains a counter $n \in \mathbb{N}$ , that counts milliseconds. By doing $s = (n \; div \; 1000) \; mod \; 60$, $m = (n \; div \; 60{\cdot}1000) \; mod \; 60$ and $h = (s \; div \; 60{\cdot}60{\cdot}1000) \; mod \; 24$, we have generated the time as a tuple **[h: m: s]** in hours, minutes and seconds. Also, we have $\mu = n \; mod \; 1000$ milliseconds. Vice versa, from a 24 hour clock the time of a day in milliseconds can now be ranked by $t_{ms} = \mu + 1000(s + 60(m + 60h))$. Note that counting milliseconds is a time traversal.

In the bounded discrete logarithm vector problem as defined in Section 1.2, the vectors we have are in fact tuples. In our settings, for vector $\mathbf{v}$, $v_i \in [0, c_i)$ and $c = \prod_{i=1}^{d} c_i < q$, we can order vectors by rank $n = 0, 1, 2, ..., c-1$ if we have a bijective function translating $n$ into vector $\mathbf{v}$ and vice versa. Set $\tau_1 = 1$ and $\tau_i = \prod_{j=1}^{i-1} c_j$.

**Definition 2.2** *For vector* $\mathbf{v}$ *we have* rank $n(\mathbf{v}) = \sum_{i=1}^{d} \tau_i v_i \in [0, c)$.

**Definition 2.3** *The* generation $\mathbf{v}(n)$ *for* $n \in [0, c)$ *is the bijective mapping* $\mathbf{v} : [0, c) \to \{(v_1, \; v_2, ..., \; v_d) \mid v_i \in [0, c_i)\}$ *given by* $\mathbf{v}(n) = (v_1(n), \; v_2(n), ..., \; v_d(n)) = ((n \; div \; \tau_1) \; mod \; \tau_2, \; (n \; div \; \tau_2) \; mod \; \tau_3, ..., \; (n \; div \; \tau_{d-1}) \; mod \; \tau_d, \; n \; div \; \tau_d)$.

## 2.2 The Birthday Paradox

If 23 randomly selected people are together, the probability that at least two of them have the same birthday is over 50%. This number is counter-intuitive at first sight, hence the name *Birthday Paradox*. To find the discrete logarithm vector, some algorithms in this thesis use a collision search that relies on the Birthday Paradox. The following results from literature, together with a proposition, are important for the analysis of this collision principle.

Consider a bag with $c$ items numbered $0, 1, 2, ..., c-1$. We randomly draw $O(\sqrt{c})$ items from it, with replacement.

**Definition 2.4** *We speak of a* collision *when we have drawn an item twice.*

**Fact 2.5** *From [10, §2.1.5], we have that the probability of at least one collision after drawing* $n = O(\sqrt{c})$ *items is* $P_n \rightarrow 1 - e^{-n^2/(2c)}$, *for* $c \rightarrow \infty$ ♡

Hence for $n = \lceil k\sqrt{c} \rceil = O(\sqrt{c})$, we have numerically

$$P_{\lceil k\sqrt{c} \rceil} \approx 1 - e^{-\frac{1}{2}k^2} \approx 1 - 0.61^{k^2}$$

and for the probability of *no collision* after drawing $\lceil k\sqrt{c} \rceil = O(\sqrt{c})$ items

$$\overline{P}_{\lceil k\sqrt{c} \rceil} \approx 0.61^{k^2}$$

We also use the expected number of random elements until a collision occurs.

**Fact 2.6** *Let $n$ be the random variable for the number of elements taken with replacement from a set of $c$ different elements, before collision. From [10, §2.1.5], we have that the expected value for $n$, is*

$$E(n) \rightarrow \sqrt{\frac{\pi c}{2}} \qquad (c \rightarrow \infty)$$

♡

As a consequence of this collision we have that a pseudo random sequence, whose next state only depends on the actual state, starts a cycle of which the average length is known.

**Fact 2.7** *From [10, §2.1.6]. Let $\{x\}_t$ be a pseudo random sequence, with a function $f : \mathbb{Z}_c \rightarrow \mathbb{Z}_c$ for stepping, defined by $x_0 \in \mathbb{Z}_c$ with $x_{t+1} := f(x_t)$. After a collision, a cycle starts with an average length of $\sqrt{\frac{\pi c}{8}}$ steps.* ♡

## 2.3 Random Walks and Checkpoints

To prevent yourself from getting lost in the forest, you can drop stones now and then. If the distance between the stones is not too great, or if you only drop stones at particular places like crossings of paths, then it will be easier to find your way back. At the same time, you make it easier for your friends to find you.

On a computer, saving markers can be applied to random walks that move by a fixed decision rule, but start at a random position. First in [11], Sedgewick et al. use markers to detect when a sequence repeats a cycle shortly after it starts. The idea of using *distinguished points* for markers stems from Ronald Rivest, and it is used by Quisquater and Delescaille in [12] for finding DES collisions. In [5], Van Oorschot and Wiener use *distinguished points* as markers for finding discrete logarithms faster. We give an overview of two types of markers, and we will explain the advantages and disadvantages of each of them.

Consider the pseudo random walks $\{x\}_t,\ \{y\}_t$ that each start at different seed positions $x_0, y_0 \in \mathbb{N}$. Both walks have the same encoding $\mathcal{E}(x_t),\ \mathcal{E}(y_t)$ that is used for a stepping rule $\mathcal{S}(..)$, such that the next state is defined by $x_{t+1} = \mathcal{S}(\mathcal{E}(x_t))$. In this context, the *state* of a sequence at $t$ is the same as the integer *position* $x_t$ for time index $t$. In the next chapters we will show algorithms that use the joining of such walks for finding the discrete logarithm vector. In order to detect early when two walks have joined and to recover a discrete logarithm vector, it is necessary to store encodings together with their intermediate states $x_t$ after $t \in \mathbb{N}$ steps for at least one of the two walks $\{x\}_t$ and $\{y\}_t$.

**Definition 2.8** *A* checkpoint *is a point in a walk that is stored as a marker. This marker contains information about the* position of the walk[1] *and a (truncated) value of the encoding.*

There are two different strategies for storing checkpoints. The first strategy is to store a checkpoint every fixed number of steps.

**Definition 2.9** *An* equidistant point *is a checkpoint that is stored directly after a fixed number of steps.*

The name *equidistant points* is chosen to emphasize that these checkpoints are based on a constant number of steps between checkpoints. Since $\{x\}_t,\ \{y\}_t$ have pseudo random step sizes, the distance stepped between every two equidistant points is not exactly constant. Suppose that we store checkpoints of $\{x\}_t$. For the first and second strategy, we need to verify whether $\{y\}_t$ joins $\{x\}_t$ *every step*, otherwise $\{y\}_t$ probably misses every checkpoint of $\{x\}_t$. Note that by storing equidistant points and verifying for a collision every step, we always mark steps in a cycle and we will always detect when a walk cycles. This way we avoid that our computer is *running idle* (repeating the same cycle, without stopping). Looking in a hash table is cheap compared to an encoding. In some algorithms it is possible to avoid look ups every step.

If we store checkpoints when the encoding $\mathcal{E}(x_t)$ or $\mathcal{E}(y_t)$ shows a certain property, then we only have to look whether $\{x\}_t$ and $\{y\}_t$ have joined or one of them is doing cycles when this property is met.

**Definition 2.10** *A* distinguished point *is a checkpoint with a particular property.*

*Distinguished points* are explained in [5, §4]. They can be defined for example as checkpoints that have the $b \in \mathbb{N}$ least significant bits of their encoding cleared. An important advantage of *distinguished points* is, that we only have to check for a collision if we find a *distinguished point*. It means, that for collision search, we can use parallel computers that are connected by internet. Now and then, a distinguished point has to be sent to the other computers. To detect that a walk has entered a cycle, we need to have at least one distinguished point inside this cycle. Notice that we can't make sure that this really happens, hence there is always a possibility that a walk ends in an undetected cycle. We will analyze this probability in Section 3.4.

There is another important difference between distinguished points and equidistant points, which is explained in [5, §4, Footnote 7]. Equidistant points are on average only half the number of steps away from the point of joining, that is, joining is on average in the middle between two equidistant points. Distinguished points however, have a geometric distribution. The joining of sequences is more probable between two distinguished points that have more steps in between. As a result, the average number of steps from the point of joining to the next distinguished point is equal to the average distance between two distinguished points. Hence, the efficiency of distinguished points in terms of steps for finding when sequences have joined, is half the efficiency of equidistant points.

[1] For a walk starting at an unknown position, we have a *relative* position. To tell whether a checkpoint has a relative position or an absolute position, we add an extra bit.

## 2.4 Hash Tables

For fast storage of data, which has to be accessed fast for lookups, a hash table is very useful. We describe the basics of a hash table first. After that we give an abstract data type. In this section, we explain hashing more abstractly with the help of [14] and [13].

### 2.4.1 Description

If $m$ unique objects $(\alpha(n), \beta(n)),\ n \in \mathbb{Z}_m$ have to be stored, we can take the identifying part $\alpha(n)$ of each of them as a key. Let $\mathcal{I} = \{\alpha(n) \mid n \in \mathbb{Z}_m\}$ be the set of identifiers and suppose there is a storage available for $s \geq m$ references to objects. As a result of the Birthday Paradox in Section 2.2, it is practically impossible to find a function $h : \mathcal{I} \rightarrow \mathbb{Z}_s$ on the objects that returns a unique index $h(\alpha(n))$ for storing each object. However, by hashing the bits of an object we can construct a pseudo random number that is unique with constant probability. Later we will explain what to do when more objects hash to the same number. First some definitions.

**Definition 2.11** *A* hash key *is part of an object that identifies the object.*

By our definition, $\alpha(n)$ is the hash key for object $n$. By hashing the key of an object, we obtain an address of a reference table that contains a pointer to the object in memory.

**Definition 2.12** *A* hash value *is a number assigned to an object that identifies the object with positive probability.*

**Definition 2.13** *A* hash function *is a function that modifies* hash keys *to* hash values *by* hashing *(doing binary operations on) the information in the hash key.*

A hash function consists of fast operations such as bit shifts, logical $AND$ and logical $OR$ that are applied to each *hash key.*

For example, if we have a database with personal information, the *hash key* can be the passport number of a person. A hash function now changes this passport number into an address of a reference table. This table refers to the memory location where the other data of the person can be, or is already, stored. In illustration 2.14 , we have *hash function* $h(\ )$ that modifies *hash key* $\alpha(n)$ from object number $n$ into *hash value* $h(\alpha(n))$. By putting the memory address of object number $n$ in $R[h(\alpha(n))]$, we now have a means to quickly address the objects that are stored.

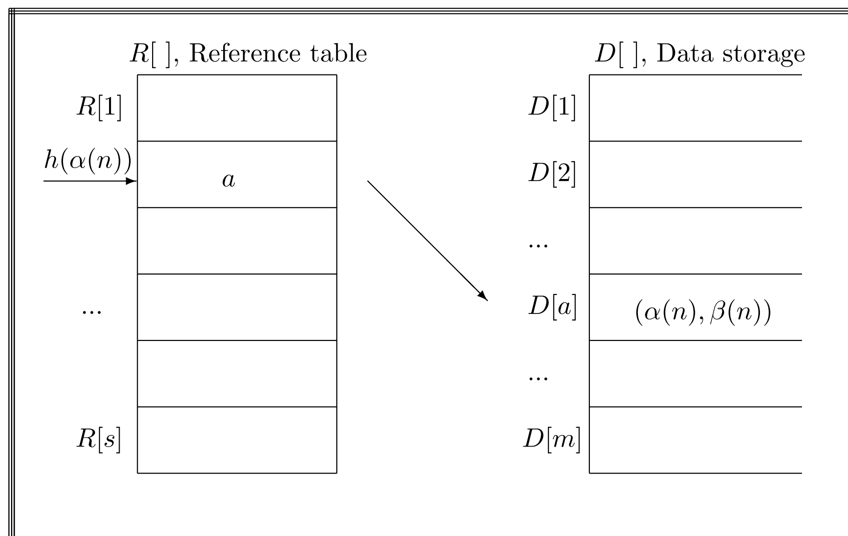

**Illustration 2.14** : Hash table with data storage table

For our purpose, we use a simpler hash table.

**Definition 2.15** Open addressing *is, when the address to store an object, is taken directly from the hash function.*

The advantage of *open addressing* is that we only need a *reference table* to directly store objects of fixed size. We will only store data of a fixed size so we restrict to this *open addressing*. Since we have to be economical with memory and *hash keys* contain redundant information, a hash function needs to truncate the result while operating on the bits of the *hash key*. Because of this truncation it is unavoidable that sometimes, different *hash keys* have the same *hash values*.

**Definition 2.16** *A* hash collision *is, when for different* hash keys $\alpha(n_1) \neq \alpha(n_2)$ *we have the same* hash value $h(\alpha(n_1)) = h(\alpha(n_2))$.

Note, that *hash collisions* are unwanted collisions. They are different from collisions that are used for finding discrete logarithms in that the latter are wanted.

**Definition 2.17** *The* load factor *of a hash table is the amount of data to be stored divided by the size of the reference table.*

We have a load factor of $\frac{m}{s}$. If this *load factor* is close to 1, then we have many *hash collisions*. To avoid that data is lost, these *hash collisions* have to be resolved. For *open addressing*, we have the following solution for *hash collisions*.

**Definition 2.18** *Let* $r \in \mathbb{N}$. Double hashing *is the use of a combined hash function* $h_d(\alpha(n), r) = h_1(\alpha(n)) + rh_2(\alpha(n)) \ mod \ s$, *where* $r$ *takes the values* $0, 1, 2, ...$ *until either* $r \geq r_{max}$ *or we find that* $R[h_d(\alpha(n), r)] = 0$ *for some* $r < r_{max}$.

This *double hashing* is *generic*, since we did not define the hash functions $h_1(\ )$, $h_2(\ )$. In practice, $h_1(\alpha(n))$ returns the address where object $n$ is stored if possible. If it succeeds we have that $r = 0$. If not, then for $r = 1, 2, ..., r_{max} - 1$, repeated addition by the the second hash function $h_2(\alpha(n))$ changes the address until a free address is found. A special case of *double hashing* is *linear probing*.

**Definition 2.19** Linear probing *is the search for a relative index* $r = 0, 1, ..., r_p - 1$ *until either* $r \geq r_p$ *or we find an* $r < r_p$ *such that* $R[h_1(\alpha(n)) + r \ mod \ s] = 0$.

An advantage of linear probing is that we keep looking in a computer cache nearby the first hash address, which is many times faster than looking in RAM that is not cached. A disadvantage is *clustering* (concentrations of non empty places in $R[\ ]$). To reduce this clustering we can also use *quadratic probing*, where we search for $r$ such that $R[h_1(\alpha(n)) + r^2 \ mod \ s] = 0$.

By *double hashing* without the limitation $r_{max}$, we will eventually find a free entry in the reference table, while we also keep the size of the reserved memory fixed. Of course, this only works if we have a *load factor* that is significantly less than 1.

To model the efficiency of *double hashing*, we need that for each new *hash key*, a *hash function* returns a pseudo random reference address.

**Convention 2.20** *We assume that hash functions are pseudo random.*

Under this assumption we can investigate the probability of collisions, and the distribution of the necessary number of rehashes to resolve a *hash collision*. First we need the distribution for the number of *hash collisions*.

**Fact 2.21** *From [17]. Suppose that we draw* $m$ *numbers randomly from* $\{0, 1, 2, ..., s-1\}$ *with replacement. Take a fixed* $n \in \{0, 1, 2, ..., s-1\}$. *Let* $m, s \to \infty$ *such that* $\lambda = \frac{m}{s}$ *is a positive constant. If* $K$ *is the random variable for the number of draws that have value* $n$, *then* $K$ *has the Poisson distribution*

$$P(K = k) = P_k(\lambda) = e^{-\lambda} \frac{\lambda^k}{k!} \ , \quad 0 \leq k \leq m$$

♡

Now we determine the average number of double hashing tries before a free address is found. Let $R$ be the random variable that denotes the number of rehashes including the first hash, until a collision is found. If we have stored $j$ data entries in a hash table then $\lambda = \frac{j}{s}$. With probability $P_0(\frac{j}{s})$, our storage table is empty at a random address. The next data entry can be stored after $r$ rehashes with probability

$$P(R = r) = P_0(\frac{j}{s})(1 - P_0(\frac{j}{s}))^{r-1}$$

Hence the number of rehashes is distributed geometrically with average $\frac{1}{P_0(\frac{j}{s})}$. Note, that during the storage of data, $\frac{j}{s}$ increases, such that we obtain a *worst case* average performance if we take $j = m$ which leads to an average of $\frac{1}{P_0(\lambda)}$ rehashes. If $\lambda = 1 - O(\frac{1}{s})$, the expected number of rehashes raises to $\Omega(s)$, which is equivalent to linear search. Therefore, in practice $\lambda$ should be not greater than 0.8.

Note, that there is always a possibility, that after *double hashing* $r_{max}$ times, still no free address is found. For reasons of efficiency, we can't make $r_{max}$ too great because we have to store the rehash number together with the objects and this reduces space efficiency. Since hash tables are meant for lossless storage, the objects that can't be stored after double hashing up to $r_{max}$ times, are put for example in a linear bin.

### 2.4.2 Abstract Data Type and Implementation

The abstract presentation of list 2.22 shows how a hash table works. Rehashing and the use of a linear bin is only shown in the implementation. We store object number $n$ belonging to hash key $\alpha(n)$ such that we can always verify if in future, a lookup is really successful. In the implementation, we use *open addressing*, hence we only need a reference table. We also implemented a linear bin for the case that $r_{max}$ *double hashes* do not resolve all *hash collisions.*

**List 2.22**: Abstract data type of a Hash Table

`new`$(m,\ \lambda,\ r_{max})$ : $S \leftarrow \emptyset$

`put`$(\ (\alpha(n),\ \beta(n))\ )$ : $S \leftarrow S\ \cup\ \{\ (h(\alpha(n)),\ \alpha(n),\ \beta(n))\ \}$

`get`$(\ \alpha(n)\ )$ : `output` $\leftarrow \{\ \beta(n)\ |\ (h(\alpha(n)),\ \alpha(n),\ \beta(n)) \in S\ \}$

**new**$(m,\ \lambda,\ r_{max})$ defines a *double hash* function $h_d(n, r) : [0, m) \rightarrow [0, \lceil \frac{m}{\lambda} \rceil)$ for $n \in [0, m)$ and $r \in [0, r_{max} - 1)$, and a table $R[0], R[1], .., R[\lceil \frac{m}{\lambda} \rceil]$. Also, a linear bin $B[\ ]$ is initialized by means of an array with a pointer to the first entry $p_B := 0$.

**put**$(\ (\alpha(n),\ \beta(n))\ )$ by rehashing, find the smallest $r \in [0, 1, ..., r_{max})$ such that $R[h_d(\alpha(n), r)]$ is empty. Then we set $R[h_d(\alpha(n), r)] := (\alpha(n),\ \beta(n),\ r)$. If such $r$ is not found, then we store this entry in the linear bin, by doing $B[p_B + +] := (\alpha(n),\ \beta(n))$.

**get**$(\ \alpha(n)\ )$ by rehashing, find $j \in \mathbb{Z}_m$ and $r \in [0, 1, ..., r_{max})$ such that $R[h_d(\alpha(n), r)] = (\alpha(j),\ \beta(j),\ r)$. Then return $\beta(j)$. If such $r$ is not found, then search the linear bin $B[0],\ B[1], ...,\ B[p_B - 1]$. If we have a $B[..] = (\alpha(j),\ \beta(j))$ such that $\alpha(j) = \alpha(n)$, then return $\beta(n)$.

## 2.5 Multi-base Exponentiation

For basic definitions, we refer to Section 1.2. In Section 2.1 we explained *rank*, *generation* and *traversal* and in [15], Knuth uses *mixed-radix generation* to generate all tuples by *visiting* them in order of their rank. We will generalize this in Section 2.5.2 for visiting ranks by different step sizes.

We take for $G_q$ the cyclic group such that multiplication can be undone by multiplication with the inverse element. In Section 2.1, we introduced a *tuple ranking* of vector $\mathbf{v}$ by integer $n(\mathbf{v})$. To find $\mathbf{v}$ given $h = \mathbf{g}^{\mathbf{v}}$, we need to *generate* $\mathbf{v}(n)$ and then evaluate $\mathbf{g}^{\mathbf{v}(n)}$ for at least some $n \in [0, c)$ by doing multiplications in the cyclic group $G_q$. The goal of this section is, to simplify vector evaluation for use in other chapters. In Section 2.5.1 we show that this number of multiplications is $O(|c|)$ for random $n \in [0, c)$. In Section 2.5.2 we show that by choosing appropriate steps for $n$, we can reduce evaluation of $\mathbf{g}^{\mathbf{v}(n)}$ to only one multiplication per step. Finally, in Section 2.5.3 we give a very simple technique for vector evaluations that is precisely two times slower than optimum.

### 2.5.1 Exponentiation of Random Vectors

We use the rank of a vector as in Section 2.1, such that $\tau_i = \prod_{j=1}^{i-1} c_j$ with $\tau_1 = 1$. First a few definitions and conventions.

**Definition 2.23** *Let $x \in \mathbb{N}$. Then $|x| = \lceil \log_2(x+1) \rceil$ is called the* bit length *of $x$.*

**Convention 2.24** *The binary representation of an integer $n \in \mathbb{N}$, is written $n = (b_{|c|-1}(n), b_{|c|-2}(n), ..., b_0(n))$, where $b_i(n) \in \{0, 1\}$.*

If we consider the binary representation of only one integer $n$, then we write $n = (b_{|c|-1}, b_{|c|-2}, ..., b_0)$.

**Convention 2.25** *The signed binary representation of an integer $n \in \mathbb{N}$, is written $n = (b_{|c|-1}(n), b_{|c|-2}(n), ..., b_0(n))$, where $b_i(n) \in \{-1, 0, 1\}$.*

This signed binary representation is useful for expressing additive inverses of integer numbers. Note that the signed binary representation is not unique, so we have to be careful when using it.

**Convention 2.26** *For example, by $(0^i \star (-1)^j 1^k)$ we mean a signed binary representation with the first $i$ coordinates $0$, the next coordinate don't care, the following $j$ coordinates $-1$ and the last $k$ coordinates $1$.*

**Convention 2.27** *By the* evaluation *of $\mathbf{g}^{\mathbf{v}(n)}$, we mean the computation of $\prod_{i=1}^{d} g_i^{v_i(n)}$.*

We explain how to evaluate $\mathbf{g}^{\mathbf{v}(n)}$ for every $n \in \mathbb{N}$. Let $\mathbf{v}(n) = (v_1, v_2, ..., v_d)$. Per coordinate, we have the binary representation $v_i = (b_{|c|-1}(v_i), b_{|c|-2}(v_i), ..., b_0(v_i))$. For coordinate $i$, we can evaluate and store $f_{i,j} := g_i^{2^j}$ for $j = 0, 1, 2, ..., |c_i| - 1,\ i = 1, 2, ..., d$. This can be done by setting $f_{i,0} := g_i$ and a successive evaluation of $f_{i,j+1} := f_{i,j}^2,\ j = 0, 1, 2, ..., |c_i| - 2$ for $i = 1, 2, ..., d$. It takes at most $\sum_{i=1}^{d}(|c_i| - 1) \leq |c|$ multiplications and also at most $|c|$ extra space. Now we can evaluate $\mathbf{g}^{\mathbf{v}(n)}$ by

$$\mathbf{g}^{\mathbf{v}(n)} = \prod_{i=1}^{d} \prod_{j=0}^{|c_i|-1} g_i^{(2^j b_j(v_i))} = \prod_{i=1}^{d} \prod_{j=0}^{|c_i|-1} f_{i,j}^{b_j(v_i)}$$

For this evaluation, given $\mathbf{g}$ and $\mathbf{v}(n)$ we need at most $(\sum_{i=1}^{d} |c_i|) - 1 \leq |c| + d - 1$ multiplications, since we do not have to multiply the first factor. As a total, we need at most $2|c| + d - 1$ multiplications and $|c|$ extra space[2]. We summarize this.

[2] For one single evaluation we do not need extra space for $f_{i,j},\ j = 1, 2, 3, ..., |c_i| - 1,\ i = 1, 2, ..., d$. In that case we use $f_{i,j}$ only when $b_j(v_i)$ is set, and continue by the substitution $f_{i,j+1} := f_{i,j}^2,\ j = 1, 2, ..., |c_i| - 2$ directly after each multiplication is done. Then we only need 1 extra space.

**Heuristic 2.28** *Suppose that we have stored $|c|$ precomputed $f_{i,j},\ j=1,2,...,|c_i|-1,\ i=1,2,...,d$. Then for every $n\in[0,c)$, evaluation of $\mathbf{g}^{\mathbf{v}(n)}$ takes at most $|c|+d-1$ multiplications.*

We can also evaluate inverse multiplicative binary evaluations $f_{i,j}^{-1}$, to clear one bit of a coordinate $v_i,\ i=1,2,...,d$ by addition of $-2^j,\ j=0,1,...,|c_i|-1$. Inverse evaluations can reduce the maximum number of multiplications for every rank $n\in\mathbb{N}$. First concentrate on one coordinate $v_i,\ i\in\{1,2,...,d\}$ with binary representation $v_i\equiv(b_{|c_i|-1},b_{|c_i|-2},...,b_0)$. Store evaluations for coordinate representations with weights $2^j$ and $2^{-j}$ for $j=0,1,...,|c_i|-1$ and $2^{|c_i|}-1$. This takes $2|c_i|-1$ extra space, if we already have that $f_{1,0}=g_i$. If coordinate $v_i$ has at most $\frac{|c_i|}{2}$ bits set, then in additive notation $v_i=\sum_{j=0}^{|c_i|-1}b_j2^j$. Correspondingly, this costs at most $\frac{|c_i|}{2}$ multiplications. If more than $\frac{|c_i|}{2}$ bits are set, then in additive notation we evaluate the coordinate by $v_i=(2^{|c_i|}-1)-\sum_{j=0}^{|c_i|-1}(1-b_j)2^j$. This takes $\frac{|c_i|+1}{2}$ multiplications. Altogether, we need at most $(\sum_{i=1}^{d}\frac{|c_i|+1}{2})-1=\frac{|c|+d}{2}-1$ multiplications, since the first bit needs no evaluation. Now this can be summarized as follows.

**Heuristic 2.29** *Suppose that we have stored $2|c|$ precomputations for coordinate representations with weights $2^j,\ j=1,...,|c_i|-1$ and $2^{-j},\ j=0,1,...,|c_i|-1$ and $2^{|c_i|}-1,\ i=1,2,...,d$. Evaluation of $\mathbf{g}^{\mathbf{v}(n)}$ costs at most $\frac{|c|+d}{2}-1$ multiplications.*

### 2.5.2 Incremental Vector Changes

Both in [15] and [2] it is explained how to use Gray Codes for a traversal. In [2], Schoenmakers also proposes a generalization of Gray Codes that traverses over all tuples by doing one multiplication per step. We explain another way to traverse all tuples, and it needs 1 multiplication per step.

In Section 2.1 we explained *rank*, *generation* and *traversal*. We use the same conventions and definitions as in Section 2.5.1. Here we will generalize rank generations for visiting ranks selectively by doing different step sizes. The advantage of this way is, that we simply work with the coordinates as we did for evaluation of random vectors. This is especially an advantage, if we have to restart partial traversals at random positions or if we need that more than one step size takes 1 multiplication. We also have nice restrictions on space requirements.

From Section 2.5.1 we know that changing one bit in coordinate $v_i$ takes only one addition by $2^j,\ j=0,1,...,|c_1|-1$, and equivalently one multiplication in $G_q$. Now an increment of the vector rank to $n'=n+1$ leads to a rippling carry bit if $b_0(v_1(n))=1$, until the carry bit arrives at the first bit that is cleared. So if $v_1=(\star^{|c_1|-j-1}01^j)\ ,j\in\{0,1,...,|c_1|-1\}$, then $v_1+1=(\star^{|c_1|-j-1}10^j)$. This rippling carry bit can go from one coordinate to another. In that case, we get $v_1'=0,\ v_2'=v_2+1$. Generally, if $v_i=c_i-1,\ i=1,2,...,k<d$ and $v_k<c_k-1$, then $v_1'=v_2'=v_3'=...=v_k'=0,\ v_{k+1}'=v_{k+1}+1$.

*Per coordinate $i$*, we can do increments including carry ripple by addition of $(0^{|c_i|-j-1}1(-1)^j)\ ,j=0,1,...,|c_i|-1$. Equivalently, we store the multiplicative evaluations for these binary vectors. Also, we can manage coordinate overflows by adding one of the vectors $(1-c_1,1,0,...,0),(1-c_1,1-c_2,1,0,...,0),...,(1-c_1,1-c_2,...,1-c_d)$ to $\mathbf{v}(n)$. Now each increment is equal to exactly one multiplication.

**Heuristic 2.30** *Suppose that we have precomputations for the first coordinate $(0^{|c_1|-j-1}1(-1)^j)\ ,j=0,1,...,|c_1|-1$ and overflow correction evaluations for the vectors $(1-c_1,1,0,...,0),(1-c_1,1-c_2,1,0,...,0),...,(1-c_1,1-c_2,...,1-c_d)$. Given $\mathbf{g}^{\mathbf{v}(n)}$, evaluation of $\mathbf{g}^{\mathbf{v}(n+1)}$ takes 1 multiplication, using $|c_i|+d$ space.*

Now we will explain how we can do ranked steps of size $2^k\tau_i,\ k=0,1,2,...,|c_i|-1,\ i=1,2,...,d$ by only one multiplication. Without overflow, by addition of the rank to $n'=n+2^k\tau_i$ we only change coordinate $v_i$. Here we have that $(\star^{|c_i|-k-j-1}01^k\star^{j-1})+(0^{|c_i|-j-1}10^{j-1})=(\star^{|c_i|-k-j-1}10^k\star^{j-1})$, where $k\in\{0,\ 1,...,\ |c_i|-1\},\ j\in\{0,\ 1,...,\ k-2\}$. We have that only the coordinates $v_i,v_{i+1},...,v_d$ can have carry overflows. We can handle these overflows by adding one of the vectors

| # steps $\approx 2\sqrt{c}$ | Stable decimal # bit changes per step in 50 runs |
|---|---|
| $2^6$ | 2 |
| $2^7$ | 2 |
| $2^8$ | 2.0 |
| $2^9$ | 2.0 |
| $2^{11}$ | 2.0 |
| $2^{13}$ | 2.00 |
| $2^{15}$ | 2.00 |
| $2^{17}$ | 2.000 |
| $2^{19}$ | 2.0000 |
| $2^{21}$ | 2.0000 |

**Table 2.32**: Average # multiplications per step in Pollard-$\lambda$

$(0,0,...,0,1-c_i,1,0,...,0),(0,0,...,0,1-c_i,1-c_{i+1},1,0,...,0),...,(0,0,...,0,1-c_i,1-c_{i+1},...,1-c_d)$ to $\mathbf{v}(n)$. Overflows happen in $\frac{2^k}{c_i}$ of the cases. For an overflow, coordinate $v_{i+1}$ has on average $\frac{k}{2}$ bits changed. The evaluations for coordinate $i$ additions cost $\binom{|c_i|-1}{2}$ space, and evaluations for carry overflows cost $d-i$ space. This brings us the following heuristic.

**Heuristic 2.31** *Let $\tau_i$ be defined as in Section 2.5.1. Store precomputations for the vectors $(0^{|c_i|-k-1}1(-1)^k0^{j-1})$, $k=0,1,...,|c_i|-1,\ j\in\{0,\ 1,...,\ k-2\}$ and overflow correction evaluations for the vectors $(1-c_1,1,0,...,0),(1-c_1,1-c_2,1,0,...,0),...,(1-c_1,1-c_2,...,1-c_d)$. Given $\mathbf{g}^{\mathbf{v}(n)}$, evaluation of $\mathbf{g}^{\mathbf{v}(n+2^k\tau_i)}$, $k=0,1,...,|c_i|-1$ takes on average $1+\frac{2^k}{c_i}\frac{k}{2}$ multiplications using $\binom{|c_i|-1}{2}+d-i$ space.*

Note that if at any time, we want to choose arbitrary $k\in\{0,1,..|c_i|-1\},\ i\in\{1,..d\}$ for making a step of size $2^k\tau_i$, then we need to prepare $\sum_{i=1}^{d}\binom{|c_i|-1}{2}$ binary vectors and $\binom{d}{2}$ vectors for coordinate overflows. To do approximately 1 multiplication each step, we need that $k2^{k-1}\ll c_i$.

These larger steps can be used for example by Baby-step Giant-step, where the Baby makes ranked steps of size 1 and the Giant makes ranked steps of size $2^k\tau_i$, for appropriate $k,\ i$.

### 2.5.3 An Amortized Variant for Testing

In our implementation of Appendix A.3, we use a variant that needs an amortized number of two multiplications for every coordinate $i$ increment by $2^k\ll c_i,\ k\in\mathbb{N},\ i=1,2,...,d$. To do this, we distinguish the bits of coordinate $v_i$ that have to be changed to arrive at $v_i+2^k$. Also we determine the bits of coordinates $v_{i+1},v_{i+2},...,v_d$ that change if we have a carry overflow.

From the incremental sequence $0,\ 1,\ 10,\ 11,\ 100,\ 101,...$ it follows that we need $\sum_{j=0}^{n}2^j-1=2^{n+1}-n-1$ bit changes to count from 0 to $2^n$. Therefore, a counter has an *average* number of $\frac{2^{n+1}-n-1}{2^n}=2\ \ (n\to\infty)$ bit changes per step. This also holds for counters that only run on the last part of $m<n$ bits. So by selective addition of these counters for $m=n,n-1,n-2,...,2,1$, we can make any natural number in $[0,2^{n+1}-1]$ by an amortized 2 bit changes per step. Now for fixed step size $2^k\ll c_i$ in coordinate $i$, we have a counter that is simply *shifted to the left* by $k$ bits. Hence, we already have an amortized number of 2 multiplications for any fixed step size. A shifted counter can be used for Giant steps of size $2^k\tau_i,\ k=0,1,2,...,|c_i|-1,\ i=1,2,...,d$ in a ranked version of Baby step Giant step.

Since $\sum_{j=0}^{\infty}2^{-j}=2$ for rank additions by every $2^k\tau_i,\ k=0,1,2,...,|c_i|-1,\ i=1,2,...,d$, we also have an *average* of 2 multiplications per step. Table 2.32 confirms an average of 2 multiplications per step with a negligible error. The significance is the decimals that have not changed in 50 runs. These random step sizes $2^k$ are used for example in Pollard-$\lambda$.

# Chapter 3

# Algorithms

This chapter contains an overview of *generic* algorithms for finding discrete logarithm vectors. *Generic* algorithms do not exploit any particular knowledge about the group operation other than how to perform the operation itself. First in Section 3.1 we show by means of a heuristic, that there is a lower bound to the number of operations for efficient algorithms. Most algorithms that we will explain in the following sections are time efficient. In Section 3.2 however, we start by basic algorithms that are very inefficient. Then in Section 3.3 we explain a fast algorithm called Baby-step Giant-step. Unfortunately Baby-step Giant-step needs a huge amount of space. As an alternative Pollard-$\rho$ is explained in Section 3.4. A typical disadvantage of Pollard-$\rho$ is, that it doesn't exploit when discrete logarithms are *bounded*. For *bounded* discrete logarithms, Pollard-$\lambda$ in Section 3.5 is an efficient algorithm.

## 3.1 Heuristic Lower Bound

To search for a lower bound for the generic bounded discrete logarithm vector problem as it is described in Section 1.2, we recall theorem 1 from [6, §2], without repeating the proof. To understand this theorem, first a definition.

**Definition 3.1** *A generic algorithm in the context of the discrete logarithm problem, is an algorithm that does not exploit any special properties of the encoding of group elements.*

For our purpose, the following theorem is simplified to avoid too many new definitions.

**Theorem 3.2** *(Theorem 1 from [6, §2]) Suppose for prime $q$, that we have indices $n \in \mathbb{Z}_q$ for cyclic group elements, and an injective mapping*

$$f : \mathbb{Z}_q \mapsto S$$

*that maps these indices into a set of binary strings in order to make elements of $S$ comparable. Now given that $h \in S$ for any generic algorithm, after evaluation[1] of $m$ elements in $S$, the probability of finding $n$ such that $f(n) = h$, is $O(m^2/q)$.*

This means that after evaluating $m = \Omega(\sqrt{q})$ elements in $S$, the probability for finding the discrete logarithm is at least a nonzero constant. For the setting in this thesis, we search for the discrete logarithm vector $\mathbf{v}$ such that $f(\mathbf{v}) = \mathbf{g}^{\mathbf{v}} = h$ for $h,\ g_i \in G_q$ and $v_i \in [0, c_i)$. By means of a heuristic argument, we suppose that a similar result holds for our setting. Unfortunately, no proof for this is found, and the proof of Shoup for prime group order $q$ does not translate to proof for a subset of size $c \in \mathbb{Z}_q$.

[1] Evaluation of elements is left to a so-called oracle in [6, §2], that has direct access to all previously evaluated elements in $S$. For each consultation the oracle applies only one operation in $S$, equivalent to one multiplication in an abelian group, to arrive at the evaluation of a new vector from previously evaluated vectors.

**Heuristic 3.3** *Let $g_i \in G_q$ and $v_i \in [0, c_i)$ with $c = \prod_{i=1}^{d} c_i$ as in Section 1.2. Let $f(\mathbf{v}) = \mathbf{g}^{\mathbf{v}}$ be injective*[2]*. Finding a discrete logarithm vector out of $c$ equiprobable vectors takes $\Theta(\sqrt{c})$ multiplications in $G_q$.*

**Argument** We think that the above heuristic holds, because for a random mapping the best known strategy is to use the Birthday Paradox, see Section 2.2. ♣

Note that it says nothing about the need for space. For Baby-step Giant-step in Section 3.3, we need $O(\sqrt{c})$ space.

## 3.2 Basic Algorithms

### 3.2.1 Brute Force

We can try all $c$ possible exponents by subsequently testing whether $g^v = g\ g^{v-1} = h$, without storing the results. This takes $\Theta(c)$ multiplications and $\Theta(1)$ space for group elements.

### 3.2.2 Precomputation

**Definition 3.4** *Let $N : G_q \to \mathbb{N}$ be an injective mapping. For $e \in G_q$, the mapping $N_s : G_q \to \mathbb{N}_s$ given by $N[e]_s = N[e] \bmod s$ is an integer representation of $e$ modulo $s$.*

We call this mapping a *truncation* of an element. Suppose we already have a table of all pairs $\{(g^v, v) \mid v \in [0, c)\}$. In this classical approach, space depends on the binary string length per element, which can be very inconvenient when elements of $G_q$ have thousands of bits. Using a hash table, we only need $\Theta(|c|)$ bits per element to find the discrete logarithm after collision in probability. For efficiency reasons, these pairs are stored in a hash table as $\{(N[g^v]_c, v) \mid v \in [0, c)\}$. By using element truncation and vector ranking, the space for this table is $\Theta(c|c|)$ bits. Now we can find the discrete logarithm for $h$ on average in $\Theta(1)$ lookups and $\Theta(1)$ evaluations. Remark that truncations of different elements can be the same. When the hash table returns an integer after a lookup, we have to verify whether this really is the discrete logarithm. Since all elements have to be calculated at least once, this can only be useful if the table is used many times.

## 3.3 Baby-step Giant-step

Shanks combined the above algorithms to a tradeoff between time and space. First we describe a simple version. After that it is generalized to a version for more coordinates. Finally, we give an optimization for the ratio of Baby steps to Giant steps.

### 3.3.1 Description

To find a discrete logarithm in $[0, c)$, we set $\mu = \lceil c^{1/2} \rceil$. Consider $h = g^v$ the encoded integer $v$. First we take one subset $S_B$ of $\mu$ elements $hg^{-v_B}$ for $v_B = 0, 1, 2, ..., \mu - 1$. These are the so-called Baby steps. For the Giant steps let $g_G := g^\mu$ and take $\mu$ elements $(g_G)^{v_G}$ for $v_G = 0, 1, 2, ..., \lceil \frac{c}{\mu} \rceil$ such that the $\mu v_G$ are spread equidistantly over $c$. Now we have that $\mu \lceil \frac{c}{\mu} \rceil \geq c$, which means that at least one of the $(g_G)^{v_G}$ is contained in $S_B$. Given that $g^v = h$ for $v \in [0, c)$, we have $v = \log_g(h) = v_B + \mu v_G$ for the right integers $v_B, v_G \in [0, \mu]$. See algorithm 3.5.

Illustration 3.6 shows how the algorithm works. When a collision is found, we can reconstruct the exponent $v$ for $h$. Since we first run all the Baby steps, we find on average the right Giant step halfway. So $v$ is on average found after $\frac{3}{2}\mu$ steps and $\frac{1}{2}\mu$ lookups, using $\mu$ space.

[2]We can restrict the $g_i$ as in footnote 1.2 of Section 1.2 to have $f(\mathbf{v}) = \mathbf{g}^{\mathbf{v}}$ injective. For $c < \sqrt{q}$, by the Birthday Paradox, $f(\mathbf{v}) = \mathbf{g}^{\mathbf{v}}$ is injective in probability

**Algorithm 3.5**: Baby-step Giant-step

1. ***input*** $g,\ h$
2. $g_G := g^{\mu}$ ; $v_G := 0$ ; $S_B := \{(hg^{-v_B}, v_B) \mid v_B := 0, 1, ..., \mu\ \}$
3. ***while*** $g_G^{v_G} \not\in S_B$ ***do*** $v_G := v_G + 1$
4. ***output*** $v := v_B + \mu v_G$

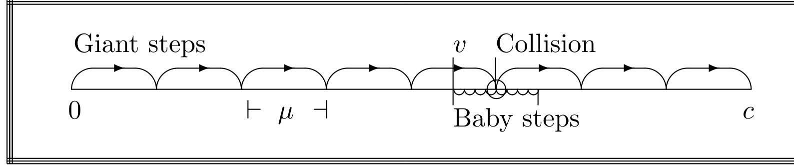

**Illustration 3.6** : Baby step - Giant step

### 3.3.2 Generalizations

In [2, §3], Schoenmakers proposes a very simple variation for Baby-step Giant-step in more than one coordinate. For $0 < k < d$, rewrite the bounded discrete logarithm vector problem $\mathbf{g}^{\mathbf{v}} = h$ for finding $v_i \in [0, c_i)$ such that $\prod_{i=k+1}^{d} g_i^{v_i} = h \prod_{i=0}^{k} g_i^{-v_i}$. After evaluation and storage of all possibilities for one side, we can find a collision by evaluating all possibilities for the other side. By clever evaluation as described in Section 2.5, we need approximately one multiplication per vector. This strategy is efficient if the subsets are approximately of the same size, that is, $\prod_{i=k+1}^{d} c_i \approx \prod_{i=0}^{k} c_i$.

**Example 3.7** We take the case of two coordinates. Given $h$, we search for $v_1 \in [0, c_1),\ v_2 \in [0, c_2)$ such that $h = g_1^{v_1} g_2^{v_2}$. First we evaluate the Baby steps $hg_1^{-v_1},\ v_1 \in [0, c_1)$ and store these results in a set $S_B$ together with the coordinate $v_1$. Next we evaluate the Giant steps $g_2^{v_2},\ v_2 \in [0, c_2)$, until we find $v_2$ with $g_2^{v_2} = hg_1^{-v_1} \in S_B$. Now we have $\mathbf{v} = (v_1, v_2)$. See illustration 3.8. Only if $c_1 \approx c_2$, this strategy is efficient.

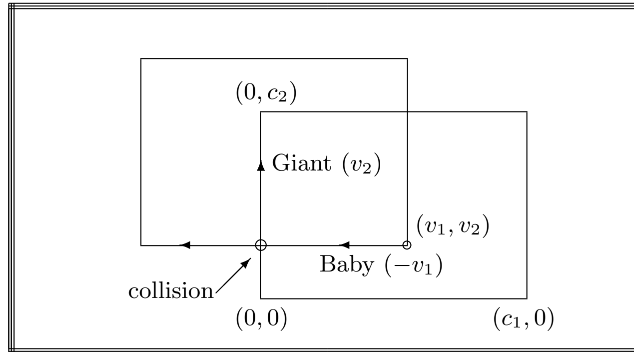

**Illustration 3.8** : One Baby and one Giant coordinate

In [2, §3], another very straightforward algorithm is explained. It is efficient for large enough $c_i$ such that $\prod_{i=1}^{d} \lceil \sqrt{c_i} \rceil \approx \sqrt{c}$. See Algorithm 3.9. In line 1, we store pairs $(h\mathbf{g}^{\mathbf{v_B}},\ \mathbf{v}_B)$. In practice, we use a hash table for truncated evaluations $(N[h\mathbf{g}^{\mathbf{v_B}}]_{\lceil\sqrt{c}\rceil},\ n(\mathbf{v_B}))$. This costs $\Theta(\sqrt{c}|c|)$ bits for space, but we have to check whether a collision is real. By traversing the vectors cleverly by rank $n(\mathbf{v_B})$ from Section 2.1 and evaluation as presented in Section 2.5.2, we need approximately one multiplication to go from one vector evaluation to another. Therefore this algorithm needs $\Theta(\sqrt{c})$ multiplications. Note that by Section 2.5.2, we can rank the Giant steps apart from the Baby steps.

**Algorithm 3.9**: Extended Baby-step Giant-step for $d$ coordinates.

1. *Evaluate the* $h\mathbf{g}^{-\mathbf{v_B}}$ *for Baby step vectors*

$$\mathbf{v_B} \in [0, \lceil\sqrt{c_1}\rceil] \times [0, \lceil\sqrt{c_2}\rceil] \times ... \times [0, \lceil\sqrt{c_d}\rceil]$$

*and store the pairs* $(h\mathbf{g}^{-\mathbf{v_B}},\ \mathbf{v_B})$ *in a table.*

2. *For* $\mathbf{c_G} = (\lceil\sqrt{c_1}\rceil,\ \lceil\sqrt{c_2}\rceil,\ ...,\ \lceil\sqrt{c_d}\rceil)$ *evaluate* $\mathbf{g_G} = \mathbf{g}^{\mathbf{c_G}}$.
3. *Evaluate* $\mathbf{g_G}^{(v_1,\ v_2,\ ...,\ v_d)}$ *for Giant step vectors*

$$\mathbf{v_G} \in \{v_1,\ v_2,\ ...,\ v_d) \mid v_i \in [0, \lceil\sqrt{c_i}\rceil],\ i = 1, 2, ..., d\}$$

*until a collision is found in the table when* $h\mathbf{g}^{-\mathbf{v_B}} = \mathbf{g}_\mathbf{G}^{\mathbf{v_G}}$.

4. *The solution is* $\mathbf{v_B} + (v_{G,1}\lceil\sqrt{c_i}\rceil,\ v_{G,2}\lceil\sqrt{c_2}\rceil,\ ...,\ v_{G,d}\lceil\sqrt{c_d}\rceil)$.

### 3.3.3 Optimizations

We start by minimizing the runtime of Baby-step Giant-step. After that, we use this optimum to make a time-space tradeoff.

As we saw in Algorithm 3.5, we need on average $\frac{3}{2}\mu$ steps until a collision is found. Ideally, for $r \in \mathbb{R}$, we can do Baby steps $0, 1, 2, ..., r\mu$ and Giant steps $0, r\mu, 2r\mu, ..., \lceil(\frac{\mu}{r} - 1)r\mu\rceil < c$. In the average case we now need $r\mu + \frac{\mu}{2r}$ multiplications. The following theorem says what is on average the best choice for $r$.

**Proposition 3.10** *The average number of multiplications for Baby-step Giant-step algorithm 3.5 is at least* $\sqrt{2c}$*, with equality for* $\sqrt{\frac{c}{2}}$ *Baby steps and* $\sqrt{2c}$ *Giant steps. For this minimum, we need* $\sqrt{\frac{c}{2}}$ *space.*

**Proof** Let $\mu = \sqrt{c}$. Suppose that for any $r \in [0, 1]$ we have $r\mu$ Baby steps and $\frac{\mu}{r}$ Giant steps such that $r\mu\frac{\mu}{r} = c$. Now we need on average $r\mu + \frac{\mu}{2r}$ multiplications and $r\mu$ space. For the optimal number of multiplications we have that $\frac{\partial}{\partial r}(r\mu + \frac{\mu}{2r}) = \mu - \frac{\mu}{2r^2} = 0$. It follows that $r = \frac{1}{\sqrt{2}}$. ◇

This result can be used to optimize any Baby-step Giant-step variant. In particular for Extended Baby-step Giant-step in Section 3.3.2 we now split each coordinate in $\mu_{B,i}$ Baby steps and $\mu_{G,i}$ Giant steps such that $\mu_{B,i}\mu_{G,i} \approx c_i$. By choosing appropriate $\mu_{B,i}$ and $\mu_{G,i}$, we have $\prod_{i=1}^{d} \mu_{B,i} \approx 2\prod_{i=1}^{d} \mu_{G,i}$ which is close to an optimum.

If we use Baby-step Giant-step to find $n \in [1, c]$ different discrete logarithms $v_1,\ v_2, ..., v_n$ given $h_1 = g^{v_1},\ h_2 = g^{v_2},\ ...,\ h_n = g^{v_n}$ in the same setting, then we can do much faster than Proposition 3.10. The next proposition can also be extended to the case of vectors.

**Proposition 3.11** *The average number of multiplications for finding* $n \in [1, c]$ *discrete logarithms by Baby-step Giant-step in one setting is at least* $\sqrt{\frac{2c}{n}}$ *multiplications. Also, we need* $\sqrt{\frac{nc}{2}}$ *space.*

**Proof** See Proposition 3.10 for the case $n = 1$. In this case we turn around the roles of Baby steps and Giant steps, which makes it different from Algorithm 3.5. For $\mu = \sqrt{c}$ and $r \in \mathbb{R}$, we do $r\mu$ Giant steps first, and the results $1, g^{\frac{\mu}{r}}, g^{2\frac{\mu}{r}}, ..., g^{\lceil(\frac{\mu}{r}-1)r\mu\rceil}$ are stored, where $\lceil(\frac{\mu}{r}-1)r\mu\rceil \in [c-r\mu, c)$. After the Giant is done, for each of the discrete logarithms given by $h_j,\ j = 1, 2, ..., n$, we do at most $\frac{\mu}{r}$ Baby steps to obtain $h_j, h_j g^{-1}, h_j g^{-2}, ...$ but we stop when a collision with a Giant step result is found. So per discrete logarithm we do on average $\frac{\mu}{2r}$ Baby steps.

The average number of steps for finding all discrete logarithms together, is $r\mu + n\frac{\mu}{2r}$. This is minimized for $r = \sqrt{\frac{n}{2}}$. So first we store $\sqrt{\frac{nc}{2}}$ Giant steps. Next, for each of the $v_i,\ i = 1, 2, ..., n$ we do $\frac{1}{2}\sqrt{\frac{2c}{n}} = \sqrt{\frac{c}{2n}}$ Baby steps to find a collision with one of the stored Giant steps. Since

**Algorithm 3.13**: Pollard-$\rho$

1. ***input*** $h$
2. $f(x_t, a_t, b_t) := \begin{cases} (x_t^2,\ 2a_t\ mod\ q,\ 2b_t\ mod\ q) & if\ \ N[x_t]_3 \equiv_3 0 \\ (gx_t,\ a_t + 1\ mod\ q,\ b_t) & if\ \ N[x_t]_3 \equiv_3 1 \\ (hx_t,\ a_t,\ b_t + 1\ mod\ q) & if\ \ N[x_t]_3 \equiv_3 2 \end{cases}$
3. $t := 0\ ;\ x_0 := 1\ ;\ a_0 := 0\ ;\ b_0 := 0\ ;\ r := 0$
4. ***do*** $\{$ ***while*** $(x_t \neq x_{2t})$ ***do*** $\{\ (x_{t+1}, a_{t+1}, b_{t+1}) := f(x_t)\ ;$
5. $(x_{2t+2}, a_{2t+2}, b_{2t+2}) := f(f(x_{2t}))\ ;\ t := t + 1\ \}$
6. $h := hg\ ;\ r := r + 1$
7. $\}$ ***until*** $b_t \neq b_{2t}$
8. ***output*** $v := (\ (a_{2t} - a_t)(b_t - b_{2t})^{-1}$ ***mod*** $q\ )\ - r$

these steps cost one multiplication, we do on average $\frac{1}{n}(\sqrt{\frac{nc}{2}} + n\sqrt{\frac{c}{2n}}) = \sqrt{\frac{2c}{n}}$ multiplications per discrete logarithm. ♢

Until now we optimized for time performance. As a result we are confident that using more space does not give a further speedup. The question remains what happens when we have less space at our disposal than needed for the fastest result. To find an answer, we use $\mu_B$ for the number of Baby steps and $\mu_G$ for the number of Giant steps. We only store evaluations $\mathbf{g^v}$ for Baby steps and $\mu_B\mu_G \approx c$. From Proposition 3.10, we know that we need on average $\mu_B + \frac{\mu_G}{2} = r\mu + \frac{\mu}{2r}$ multiplications. In a tradeoff, we can vary $r$ in order to use less space at the cost of more time.

**Corollary 3.12** *If we use a fraction $r \in (0, 1]$ of $\sqrt{\frac{c}{2}}$ space, then the average number of multiplications is $f_t = \frac{1}{2}(r + \frac{1}{r})$ times greater than minimum $\sqrt{2c}$.*

**Proof** Suppose that in the optimal case we have $\mu$, $\mu_B$ and $\mu_G$ such that exactly $\mu_B\mu_G = \mu^2 = c$. We need space to store $\mu_B$ Baby steps and after that we do $\mu_G$ Giant steps. From Proposition 3.10 we know that the minimum expected time is $\mu_B + \frac{1}{2}\mu_G = \frac{\mu}{\sqrt{2}} + \frac{\mu}{\sqrt{2}} = \mu\sqrt{2}$, where $\mu_G = 2\mu_B$. If for $r \in [0, 1]$ we have $r\mu_B$ space, then we need $r\mu_B + \frac{1}{2r}\mu_G = r\frac{\mu}{\sqrt{2}} + \frac{\mu}{r\sqrt{2}} = \frac{1}{2}(r + \frac{1}{r})\mu\sqrt{2}$ time, such that $f_t = \frac{1}{2}(r + \frac{1}{r})$. ♢

## 3.4 Pollard-$\rho$

We start with an algorithm that solves the discrete logarithm problem. Given $h \in G_q$ it finds $v \in [0, q)$ such that $g^v = h$. From the Birthday Paradox in Section 2.2, we have results that apply to the Pollard-$\rho$ algorithm. For algorithm 3.13 , we were inspired by [3, §8.3.3]. Because we use only one type of sequence throughout this thesis, we use the following convention for pseudo random sequences.

**Convention 3.14** *By a* sequence *we mean a deterministic pseudo random sequence, that has the next pseudo random state as a function of its actual pseudo random state only.*

Now these sequences can be subdivided into two kinds.

**Definition 3.15** *An* unknown sequence *is a sequence that is started at an unknown seed state.*

**Definition 3.16** *A* known sequence *is a sequence that is started at a known seed state.*

In the algorithm, first a sequence $\{x\}_t,\ t = 0, 1, 2, ...$ is defined, that steps over $G_q$ by evaluating elements $g^{a_t}h^{b_t},\ t = 0, 1, 2, ...$, where $a_{t+1},\ b_{t+1}$ only depend on the previous state $(g^{a_t}h^{b_t},\ a_t,\ b_t)$. These exponents of $g$ and $h$ are updated together with element $x_t = g^{a_t}h^{b_t}$.

As a result of the Birthday Paradox in Section 2.2, after $\Theta(\sqrt{q})$ steps the sequence ends up in a cycle of length $\Theta(\sqrt{q})$ steps. This cycle is detected with help of *Floyd's cycle*, such that we only need space for two elements $x_t$ and $x_{2t}$. Since for certain $T \in \mathbb{N}$, the sequences $\{x\}_t$ and $\{x\}_{2t}$ both contain the subsequence $x_{2T}, x_{2T+2}, ...$ and since this sequence ends in a cycle, for certain $t > T$ we have that $x_t = x_{2t}$. Then, because $g^{a_t}h^{b_t} = g^{a_{2t}}h^{b_{2t}}$, we have

$$\log_g(h) = (a_{2t} - a_t)(b_t - b_{2t})^{-1} \; mod \; q$$

provided that $b_t - b_{2t} \not\equiv_q \quad 0$, which is so in $\frac{q-1}{q}$ of the cases. For the other $\frac{1}{q}$th of the cases, we rerun the algorithm for alternative $h' = hg^r$, $r = 0, 1, 2, ...$ until $b_t - b_{2t} \not\equiv_q \quad 0$ is found. The algorithm is discussed first in [4, §1]. Unfortunately, this algorithm takes roughly three times more multiplications than is strictly necessary. We will now concentrate on how to detect it when a *single* sequence starts doing cycles.

From Fact 2.5 in Section 2.2, we have that the number of steps until collision is on average $\sqrt{\frac{\pi q}{2}}$. Suppose that after doing $T$ steps first, a sequence starts to cycle with a cycle length of $n$ steps. Then for all $t \geq T$, there exist $(a_t, b_t)$ and $(a_{t+n}, b_{t+n})$ such that $g^{a_t}h^{b_t} = g^{a_{t+n}}h^{b_{t+n}}$ and the discrete logarithm is $v = \frac{a_{t+n} - a_t}{b_t - b_{t+n}}$ in $\frac{q-1}{q}$th of the cases.

In practice, we only store checkpoints such that we know whenever a cycle has started. It follows that Pollard-$\rho$ is efficient in terms of multiplications. From Fact 2.7, we know that a cycle has an average length of $\sqrt{\frac{\pi q}{8}}$ steps. If we use $u \in \mathbb{N}$ *equidistant points* every $\frac{1}{u}\sqrt{\frac{\pi q}{8}}$ steps, then after a sequence cycles, we detect it on average in $\frac{1}{2u}\sqrt{\frac{\pi q}{8}}$ more steps. Since we need on average $2u$ checkpoints, the average number of steps for $u$ equidistant points is

$$\sqrt{\frac{\pi q}{2}} + \frac{2}{2u}\sqrt{\frac{\pi q}{8}} = \sqrt{\frac{\pi q}{2}}(1 + \frac{1}{2u})$$

However, if we use *distinguished points* on average per $\frac{1}{u}\sqrt{\frac{\pi q}{8}}$ steps, then we risk that there is no distinguished point in a cycle, since there is a possibility that we find no distinguished points in $\sqrt{\frac{\pi q}{8}}$ steps. Also, as described in Section 2.3, the average number of steps between the start of a cycle and a distinguished point is twice the number of steps between the start of a cycle and an equidistant point.

First we have, that the cycle length can be too short to find one or more distinguished points. Let $K$ be the random variable for the number of steps before a cycle starts, and let $k > 0$. Then from Fact 2.5, for an average of $k\sqrt{\frac{\pi q}{2}}$ steps $P(K > k\sqrt{\frac{\pi q}{2}}) \approx 1 - e^{-\frac{\pi}{4}k^2}$ and for the probability density function, we have[3] that

$$P(K = k\sqrt{\frac{\pi q}{2}}) \approx k\frac{\pi}{2}e^{-\frac{\pi}{4}k^2}$$

In the second place distinguished points occur only in probability, such that by accident they do not happen within a cycle[4] of length $\frac{k}{2}\sqrt{\frac{\pi q}{2}} = k\sqrt{\frac{\pi q}{8}}$. Therefore, for $u \ll \sqrt{\frac{\pi q}{8}}$ let the probability to find a distinguished point in one step be $p = u\sqrt{\frac{8}{\pi q}}$. Let random variable $T$ denote the number of steps until a distinguished point. Then $T$ is distributed geometrically by

$$P(T = t) = p(1 - p)^{t-1}$$

with an expected number of $E(T) = \frac{1}{p}$ steps. For the probability of no distinguished point after $t = \frac{u}{p}$ steps, we have $P(T > t) = (1 - p)^t$. So approximately with a probability of $(1 - p)^{\frac{u}{p}} \approx e^{-u}$, no distinguished point is found within $\frac{u}{p} = \sqrt{\frac{\pi q}{8}}$ steps. Now we want to know what the probability

[3] We neglect that Fact 2.5 and $k\sqrt{\frac{\pi q}{2}}$ offer only *approximations*, while in fact sequences only do an integer number of steps.

[4] On average a cycle is half the number of steps done. This is obvious, since the first element that is drawn twice, is drawn pseudo randomly from a set of elements that are all drawn once.

is, that we find no distinguished point in a random cycle. For a cycle of length $k\sqrt{\frac{\pi q}{8}}$, this is $e^{-ku}$. By integration, we find an approximation of the overall probability that a random cycle has no distinguished point. We call this the probability of failure.

$$\frac{\pi}{2}\int_0^\infty ke^{-\frac{\pi}{4}k^2}e^{-ku}dk$$

For instance, by numerical integration, we have for $u = 5$ a failure probability of 0.054, while for $u = 50$ this is $6.3 \cdot 10^{-4}$. For $u = 200$, we have a failure probability of $3.9 \cdot 10^{-5}$. For increasing $u$, this failure probability drops fast. Therefore, we have no significant loss if we stop a sequence after a maximum number of steps[5]. As a result, for $u \gg 1$ the influence of cycles without distinguished point is negligible. From Section 2.3, we have that distinguished points need on average twice the number of steps of equidistant points until a cycle is detected. Since we need $2u$ distinguished points per $\sqrt{\frac{\pi q}{2}}$ steps, the average number of steps for Pollard-$\rho$ with $u$ distinguished points is

$$\sqrt{\frac{\pi q}{2}} + \frac{2}{u}\sqrt{\frac{\pi q}{8}} = \sqrt{\frac{\pi q}{2}}(1 + \frac{1}{u})$$

If $u$ is a small number then we have to make a tradeoff between equidistant points and distinguished points. On one hand, equidistant points need a verification every step. On the other hand, equidistant points are twice as efficient as distinguished points. Also, when using distinguished points there is a probability that sequences start a cycle without having a distinguished point, such that we either need to store more distinguished points or we risk that we regularly have to restart sequences after a while. The details of such tradeoff depend largely on the implementation.

## 3.5 Pollard-$\lambda$

### 3.5.1 Description

In this section we present an algorithm that solves the bounded discrete logarithm problem. Given $h \in G_q$ Pollard-$\lambda$ finds $v \in [0, c)$ for $c \in \mathbb{Z}_q$ such that $g^v = h$. The discrete logarithm $v$ is found in $\Theta(\sqrt{c})$ multiplications and $\Theta(|c|)$ space for elements. As shown in [5, §5] this $\lambda$ algorithm is faster than the $\rho$ algorithm if $c \in [0, 0.39q)$. In his original article [4, §3], Pollard gave rather heuristic arguments and a sketchy analysis for the performance of his $\lambda$ algorithm. First see algorithm 3.17 and illustration 3.18 .

We have encodings $g^v$ , $v \in [0, c)$. For a sequence of different $v$ the truncated encodings $N[g^v]_s$, $s \in \mathbb{N}$ are used for making step sizes $\mathcal{S}(i)$ pseudo randomly. We have defined a pseudo random stepping function $f(x) = x + \mathcal{S}(N[g^x]_s) \bmod q$ in line 3. In line 4 we run the known sequence $\{x\}_t$, starting at known position $x_0 = c$. In line 5 we run unknown sequence $\{y\}_t$, starting at an unknown position $y_0 = \log_g(h) + r$, for which we have stepping by $f(y) = y + \mathcal{S}(N[hg^{y_t}]_s)$ mod $q$. When $y_t > x_n$, or if $y_t$ collides with $x_n$, then we stop doing line 6. If no collision is found, then in line 7 we rerun sequence $\{y\}_t$ at a different starting position $r$.

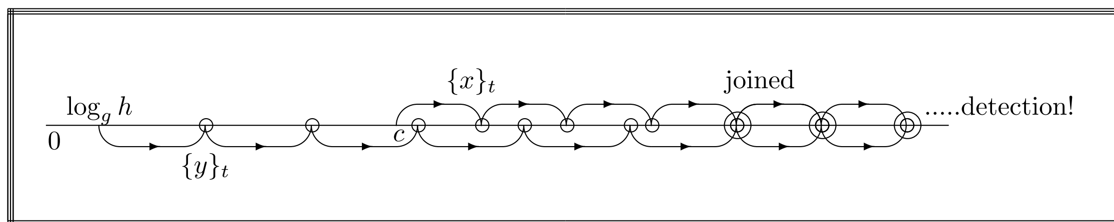


**Illustration 3.18** : Pollard-$\lambda$

[5] In [5, §4], sequences that ran for too long without finding a distinguished point are stopped. Because the probability that this happens is very small, on average we do not lose a significant number of steps.

**Algorithm 3.17**: Pollard-$\lambda$

1. ***input*** $h\ ;\ c\ ;\ n\ ;\ s\ ;\ \mathcal{S}(0),\ \mathcal{S}(1),...,\ \mathcal{S}(s-1)$
2. $r := 0\ ;\ x_0 := c$
3. $f(x,e) := x + \mathcal{S}(N[e]_s)\ mod\ q$
4. ***for*** $(t = 0\ to\ n)$ ***do*** $x_{t+1} := f(x_t, g^{x_t})$
5. ***do*** $\{\ r := r+1\ ;\ y_0 := r\ ;\ t := 0$
6. ***while*** $(hg^{y_t} \neq g^{x_n} \wedge y_t < c + x_n)$ ***do*** $\{y_{t+1} := f(y_t, hg^{y_t})\ ;\ t := t+1\}$
7. $\}$ ***until*** $(\ hg^{y_t} = g^{x_n})$
8. ***output*** $v := x_n - y_t$

### 3.5.2 Analysis and Optimization

We follow the original description of Pollard-$\lambda$ but we try to be a little bit more precise. In [4, §3], $\mathcal{S}(.)$ is supposed to be best when having a negative exponential distribution, which is known to be memoryless. After a few experiments, Pollard suggests that $\mathcal{S}(i) = 2^i,\ i \in \mathbb{Z}_s$ has a good performance. For a negative exponential distribution the average is equal to the standard deviation. From Section 2.5.2 we know that each step of size $2^i,\ i \in \mathbb{Z}_s$ corresponds to only one multiplication, which is the smallest effort for a random step. Moreover, for the negative exponential distribution we only need space for the precomputed $\Theta(|c|)$ step sizes.

**Definition 3.19** *The average of step sizes* $\mathcal{S}(i),\ i \in \mathbb{Z}_s$ *is given by* $\mu = \frac{1}{s}\sum_{i=0}^{s-1}\mathcal{S}(i)$.

Assume that the $\mathcal{S}(i),\ i \in \mathbb{Z}_s$ has average $\mu$ and variance $\mu^2$. Now sequence $\{x\}_t$ makes $n$ steps from $x_0$ to $x_n$. Assume that we have to find $v$. Then, on average after $r = \frac{c-v}{\mu}$ steps, the unknown sequence position $y_r$ is close to the starting position of the known sequence $x_0$. Since $v = \log_g(h) \in [0,c)$, sequence on average $r = \frac{c}{2\mu}$ steps before $\{y\}_t$ can join $\{x\}_t$. Now after $r$ steps say, $\{y\}_t$ can join $\{x\}_t$, that is, $y_r \in [x_0, x_0 + \mu)$. From step $r$ on, $\{y\}_{t+r}$ joins $\{x\}_t$ with probability $\frac{1}{\mu}$ per step. For $k > 0$, after $t = k\mu$ more steps, the probability of $\{y\}_{t+r}$ joining $\{x\}_t$ has the geometric distribution. Let $T$ be the random variable for the number of steps until collision. Now

$$P(T = t) = \frac{1}{\mu}(1 - \frac{1}{\mu})^{t-1}$$

which has an average of $\mu$ steps, such that on average $k = 1$. Note that in [4, §3], by approximation $P(T > k\mu) = (1 - \frac{1}{\mu})^{k\mu} \approx e^{-k}$, which also has average $k = 1$.

Suppose that $\{y\}_t$ has done $r$ steps, the next $k\mu$ steps[6] for both $\{x\}_t$ and $\{y\}_t$ until joining. Then for optimization we have on average a total of $r + 2k\mu$ steps, where $r$ has average $\frac{c}{2\mu}$. So we have to minimize $\frac{c}{2\mu} + 2k\mu$. For constant $\gamma = \frac{\mu}{\sqrt{c}}$, we minimize $\sqrt{c}(\frac{1}{2\gamma} + 2k\gamma)$. The minimum is at $\gamma = \frac{1}{2}k^{-1/2}$, giving $2k^{1/2}\sqrt{c}$ steps of average size $\mu = \frac{1}{2}k^{-1/2}\sqrt{c}$. The value $k = 1$ only holds when the sequences are not restarted, but run concurrently until they join. This is explained in Section 3.5.3.

**A First Pollard-$\lambda$ Variant with Checkpoints** We now optimize a Pollard-$\lambda$ variant, where the known sequence runs once while the unknown sequence runs repeatedly until it successfully joins the known. The first part of our analysis is from [5, §5.1], where Pollard-$\lambda$ needs on average $3.28\sqrt{c}$ steps. We repeat the analysis of [5, §5.1] and then optimize it for $u$ checkpoints.

Let a known sequence start at $c$ with element $g^c$ and make $\mu\beta$ steps of average size $\mu$. Next we start unknown sequences until a collision is found. After an unknown sequence passes $c$ it has a probability of $\frac{1}{\mu}$ to join the known sequence each step. After $\mu\beta$ more steps, it has probability of $1 - (1 - \frac{1}{\mu})^{\mu\beta} \approx 1 - e^{-\beta}$ to join the known sequence.

[6] In practice, the number of steps for $\{x\}_t$ and $\{y\}_t$ is not exactly the same.

| # checkpoints | average # steps |
|---|---|
| 1 | $3.28\sqrt{c}$ |
| 10 | $2.93\sqrt{c}$ |
| $\geq 100$ | $2.89\sqrt{c}$ |

**Table 3.20**: Classic Pollard-$\lambda$ using checkpoints

The average end for the known sequence is at $c + \mu^2\beta$. Because the average start for the unknown sequence is $\frac{c}{2}$, the expected number of steps for the last unknown sequence is $\frac{c}{2\mu} + \mu\beta$. The unknown sequence is expected to succeed once and fail $\frac{1}{1-e^{-\beta}} - 1$ times.

Now if we introduce constant $\gamma$ such that $\mu = \gamma\sqrt{c}$ then the total runtime is

$$\mu\beta - \frac{c}{2\mu} + (\mu\beta + \frac{c}{\mu})\frac{1}{1-e^{-\beta}} = \sqrt{c}(\ \gamma\beta - \frac{1}{2\gamma} + (\gamma\beta + \frac{1}{\gamma})\frac{1}{1-e^{-\beta}}\ )$$

in terms of multiplications. By approximation we find, that $\gamma = 0.51$ such that $\mu = 0.51\sqrt{c}$ and $\beta = 1.39$, with a total of $3.28\sqrt{c}$ steps.

We investigate how much faster this classical Pollard-$\lambda$ can be made if only the known sequence leaves $u$ checkpoints spread out equidistantly over $\mu\beta$ steps. In this case the distance between checkpoints is $\frac{\mu\beta}{u}$. The last sequence has to do an average of $\frac{c}{2\mu}$ steps first to pass $c$, and after that it does on average $\frac{\mu\beta}{u}\sum_{i=0}^{u} e^{-\beta i/u}$ steps until it collides with a checkpoint. Now the total runtime is

$$\mu\beta - \frac{c}{2\mu} + (\mu\beta + \frac{c}{\mu})(\frac{1}{1-e^{-\beta}} - 1) + \frac{c}{\mu} + \frac{\mu\beta}{u}\sum_{i=0}^{u-1} e^{-\beta i/u}$$

multiplications. In table 3.20 we show approximations of the performance for a different number of steps[7]. The result does not change significantly for $u \geq 100$. From this limited reduction in speed, we conclude that it only makes sense to use a few checkpoints.

**Optimum Step Sizes** By many test results[8], of which a typical example is shown in table 3.22 , we verify that it is best to take $s$ such that $\mu = \frac{1}{s}\sum_{i=0}^{s-1}\mathcal{S}(i) = \frac{2^s-1}{s} \approx \sqrt{c}$. So for step sizes $\mathcal{S}(i) = 2^i,\ i \in \{0, 1, 2, ..., s-1\}$, we have $s \approx \frac{|c|}{2} + \|c\| - 1$, where $\|c\| = \lceil \log_2(|c| + 1)\rceil$. As a result, optimally, the maximum step size is $M = \Theta(\sqrt{c}|c|)$.

Note that there is a great difference in performance if we take $s+1$ or $s-1$ instead of $s$. More importantly, if we take $\mu = \frac{2^s-1}{s}$, the maximum relative approximation error compared to $\sqrt{c}$ is $\frac{1}{\mu}\frac{1}{s}2^{s-1} \approx \frac{1}{2}$. This can influence the efficiency very much. As we derived earlier at the beginning of this subsection, the performance of Pollard-$\lambda$ is expressed by $\sqrt{c}(\frac{1}{2\gamma} + 2k\gamma)$ steps, with $\gamma = \frac{\mu}{\sqrt{c}}$ and on average $k = 1$, if we run the sequences concurrently until they join. Now the relative error is at most $\frac{1}{2}(\frac{1}{(1-\frac{1}{2})} + (1 - \frac{1}{2})) - 1 \approx \frac{1}{2}(2 + \frac{1}{2}) - 1 = 25\%$. So approximation errors may have an important influence on the performance of Pollard-$\lambda$. There are better approximations for the average step size $\sqrt{c}$.

**Example 3.21** If we have step sizes

$$\{1\} \cup \{2^i \mid i = 1, 2, ..., s_1\} \cup \{3^i \mid i = 1, 2, ..., s_2\} \cup \{5^i \mid i = 1, 2, ..., s_3\}$$

then the total number of step sizes is $s = 1 + s_1 + s_2 + s_3$. Now for $s_1, s_2, s_3 \gg 1$, the average step size is

$$\mu = \frac{1}{s}\sum_{i=0}^{s-1}\mathcal{S}(i) \approx \frac{1}{s}(2^{s_1+1} + \frac{3^{s_2+1}}{2} + \frac{5^{s_3+1}}{4})$$

Suppose that $3^{s_2}, 5^{s_3} < 2^{s_1}$. Then because each single step size is weighted by a factor $\frac{1}{s},\ s > s_1, s_2, s_3$, we have that the influence on the average of each separate step is smaller than if we only had step sizes $2^i,\ i = 0, 1, 2, ..., s$.

[7] For different $u$, the optimum values of $\mu$ and $\beta$ are a little bit different from previous values.

[8] see Appendix B.2

| **maximum step size** $2^{s-1}$ | **average # steps in** 50 **runs,** $\lvert c\rvert = 20,\ 22,\ 24,\ 26,\ 28,\ 30$ |
|---|---|
| $s = \frac{\lvert c\rvert}{2} + \lVert c\rVert - 1$ | 2113, 3248, 6743, 14703, 32475, 59121 |
| $s-1$ | 2232, 5228, 10518, 21821, 65419, 86332 |
| $s-2$ | 3577, 5675, 18105, 31864, 66285, 141908 |
| $s+1$ | 2112, 3871, 17049, 20811, 55225, 79352 |
| $s+2$ | 3578, 7873, 18236, 18344, 56537, 168568 |

**Table 3.22**: Performance of Concurrent Pollard-$\lambda$ for different $s$

| $\lvert c\rvert$ | $\frac{average\ \#\ steps}{\sqrt{c}}$ **after** 50 **runs per** $c$ |
|---|---|
| 12 | 1.92 |
| 14 | 1.98 |
| 16 | 2.93 |
| 18 | 1.97 |
| 20 | 2.55 |
| 22 | 1.80 |
| 24 | 1.83 |
| 26 | 2.15 |
| 28 | 1.77 |
| 30 | 2.01 |
| 32 | 1.78 |

**Table 3.23**: Example of Pollard-$\lambda$ for different $c$

In our Java package, see Appendix A, we only implemented the step sizes $2^i,\ i = 0, 1, 2, ..., s_1$. Table 3.23 shows that for these exponential step sizes as supposed by Pollard, we need $\Theta(\sqrt{c})$ steps. These results confirm that our model for joining probability $\Theta(\frac{1}{\mu})$ per step, is correct. We have a significant difference in the average number of steps for different values of $c$. This verifies that the number of steps depends on $c$, as we expected from the rough approximations $s \approx \frac{\lvert c\rvert}{2} + \lVert c\rVert - 1$ and $\mu = \frac{1}{s}\sum_{i=0}^{s-1} 2^i$.

### 3.5.3 Concurrent Pollard-$\lambda$

Given that $h \in G_q$ as in Section 3.5.2, we have to find $v \in [0, c)$ such that $h = g^v$. As we saw in 3.5.2, the classical method needs on average $3.28\sqrt{c}$ steps, if the known sequence runs once while the unknown sequence runs repeatedly until it successfully joins the known. In [5, §5.1], a parallel algorithm is presented that has average runtime $\frac{2}{m}\sqrt{c}$ for $m$ processors. There, known and unknown sequences run concurrently, until one sequence collides with a *distinguished point* of the other. In this section, we restrict to $m = 1$ and on average $2\sqrt{c}$ steps are needed. Independently, by using a different parallelization technique for $m$ processors, in Section 6.3.2, we found the same result for $m = 1$.

**Heuristic 3.24** *Concurrent Pollard-$\lambda$ needs on average $2\sqrt{c}$ multiplications.*

**Argument** We start the known sequence on average at $E(v) = \frac{c}{2}$. For $u \in \mathbb{N}$, we store equidistant points every $\frac{k\mu}{u}$ steps[9], and we let this known sequence run. At the same time the unknown sequence starts from unknown integer $v = \log_g(h)$, and it also stores equidistant points. Both sequences continue until at a certain point, the unknown sequence steps on a equidistant point of the known sequence, or vice versa.

In this setting, one sequence starts on average $\frac{c}{4\mu}$ steps behind the other sequence. Since both sequences run at the same time, after $2\frac{c}{4\mu} = \frac{c}{2\mu}$ steps one sequence can find an equidistant point of the other sequence. To detect such a collision, we need on average $\frac{2k\gamma}{2u}$ more steps. For constant $\gamma = \frac{\mu}{\sqrt{c}}$, we minimize $\sqrt{c}(\frac{1}{2\gamma} + (1 + \frac{1}{2u})2k\gamma)$. As $u \to \infty$, we have $\sqrt{c}(\frac{1}{2\gamma} + (1 + \frac{1}{2u})2k\gamma) =$

[9] We can also use *distinguished point*, but *equidistant points* are more precise for analysis.

| # checkpoints per $\sqrt{c}$ steps | average # steps after 50 runs for $\lvert c\rvert = 20,\ 24,\ 28$ |
|---|---|
| 1 | 4308, 14260, 66854 |
| 2 | 3572, 10902, 47193 |
| 4 | 3034, 9473, 40480 |
| 8 | 2845, 8579, 37129 |
| 16 | 2651, 8051, 35426 |
| 32 | 2619, 7721, 34557 |
| 64 | 2584, 7646, 33915 |
| 256 | 2560, 7551, 33622 |
| 1024 | 2554, 7533, 33519 |
| 4096 | 2552, 7528, 33492 |

**Table 3.25**: Concurrent Pollard-$\lambda$ for different numbers of checkpoints

$\sqrt{c}(\frac{1}{2\gamma}+2k\gamma)$, which is minimum for $\gamma = \frac{1}{2}k^{-1/2}$. If random variable $T$ is the number of steps until collision, then $T$ has the geometric distribution $P(T = t) = \frac{1}{\mu}(1-\frac{1}{\mu})^{t-1}$, and $P(T > t) = (1-\frac{1}{\mu})^t$. Since on average $E(T) = \mu$, for $t = k\mu$ steps until collision we have that on average $k = 1$ and $\sqrt{c}(\frac{1}{2\gamma} + 2k\gamma) = 2\sqrt{c}$. Optimally, each step is equivalent to one multiplication. ♣

Suppose that we can make a precise average step size $\mu = \sqrt{c}$. Then optimally both sequences do on average $\frac{1}{2}\sqrt{c}$ steps before they can join, and they continue for another $\frac{1}{2}\sqrt{c}$ steps on average until a collision has occurred. Since we do not know which of the two sequences is ahead at the start, it appears that when sequences are to join, running them simultaneously is best.

In table 3.25, we show the results if we run sequences concurrently, while storing equidistant points. For the algorithm called Concurrent Pollard-$\lambda$, see Appendix A. Ideally, if we double the number of equidistant points per $\sqrt{c}$ steps, the average distance from the optimum number of steps is cut by half. Again we see that the algorithm needs approximately $2\sqrt{c}$ steps.

We finish with the case, in which we have to find $n \in [1, c]$ discrete logarithms for a setting with only one generator. We use Concurrent Pollard-$\lambda$ to find $n$ different discrete logarithms $v_1,\ v_2, ..., v_n$ given $h_1 = g^{v_1},\ h_2 = g^{v_2},\ ...,\ h_n = g^{v_n}$.

**Heuristic 3.26** *Using Concurrent Pollard-$\lambda$, the average number of multiplications per discrete logarithm for $n \in [1, c]$ discrete logarithms in one setting is at most $\frac{2}{n}\sqrt{(n+1)c}$ multiplications.*

**Argument** See Heuristic 3.24 for the case $n = 1$. For $n > 1$, we run one known sequence starting at 0 over the whole interval $[0, c)$ until it passes $c$. After that, we concurrently run $n$ unknown sequences starting at unknown positions $\log_g(h_1), \log_g(h_2), ..., \log_g(h_n)$ that eventually join this known sequence such that we find the $v_1, v_2, ..., v_n$. At the same time, we continue the known sequence because unknown sequences can start nearby $c$ and they have to join the known sequence to find the discrete logarithm. So in the second stage, we run $n+1$ sequences instead of $n$. For average $k = 1$ and $u \to \infty$ checkpoints and $\gamma = \frac{\mu}{\sqrt{c}}$, we have to minimize $\frac{c}{\mu} + (n+1)k\mu = \sqrt{c}(\frac{1}{\gamma}+(n+1)\gamma)$. This minimum is at $\gamma = \frac{1}{\sqrt{n+1}}$ and by definition $\mu = \gamma\sqrt{c} = \sqrt{\frac{c}{n+1}}$. On average, the known sequence starting at 0 does $\frac{c}{\mu} = \sqrt{(n+1)c}$ steps until it passes $c$, and it is used to find all discrete logarithms.

Since we now have smaller step sizes, it takes on average $\sqrt{\frac{c}{n+1}}$ steps per unknown sequence for finding the discrete logarithm. If we also divide the steps of known sequences over all $n$ discrete logarithms, then we have

$$\frac{1}{n}(\sqrt{(n+1)c} + (n+1)\sqrt{\frac{c}{n+1}}) = \frac{2}{n}\sqrt{(n+1)c}$$

multiplications per discrete logarithm. ♣

**A Time-space Tradeoff** We finish by a tradeoff between the expected number of checkpoints and the performance in number of steps, if for $u \in \mathbb{N}$, we store an equidistant point[10] every $\lceil \frac{c}{u\mu} \rceil$ steps. By Section 3.5.2, we have that $\frac{\mu}{\sqrt{c}} = \frac{1}{2}$ and the average starting distance is $\frac{\sqrt{c}}{2}$ steps for both sequences, until they can join[11]. After this distance is stepped, both make on average $\frac{c}{u\mu}$ more steps each and the probability of joining is by approximation $(1-\frac{1}{\mu})^{\frac{\mu}{u}} \approx 1 - e^{-\frac{1}{u}}$. Now on average after a total of $\frac{\sqrt{c}}{2} + \frac{c}{u\mu}$ steps for each sequence, one sequence collides with an equidistant point of the other. With probability $1 - e^{-\frac{1}{u}}$ we have a collision, and with probability $e^{-\frac{1}{u}}$, we continue with both sequences. Now, after a total of $\sqrt{c} + 2\frac{c}{u\mu}$ steps for both sequences together, we continue with probability $e^{-\frac{2}{u}}$, and so on. As a result, we expect to use approximately

$$\sqrt{c} + 2\sum_{i=0}^{\infty} \frac{c}{u\mu} e^{-\frac{i}{u}} = \sqrt{c} + \frac{2c}{u\mu} \sum_{i=0}^{\infty} (e^{-\frac{1}{u}})^i = \sqrt{c} + \frac{\sqrt{c}}{u} \frac{e^{1/u}}{e^{1/u} - 1}$$

$$= \sqrt{c} + \frac{\sqrt{c}}{u} \frac{1 + \frac{1}{u} + O(\frac{1}{u^2})}{1 + \frac{1}{u} - 1 + O(\frac{1}{u^2})} = 2\sqrt{c}(1 + \frac{1}{2u} + O(\frac{1}{u^2}))$$

steps. For $u$ equidistant points per $\lceil \frac{c}{\mu} \rceil$ steps, the expected number of equidistant points is

$$2\sum_{i=0}^{\infty} e^{-\frac{i}{u}} = 2\frac{e^{1/u}}{e^{1/u} - 1} = 2(\frac{1 + \frac{1}{u} + O(\frac{1}{u^2})}{1 + \frac{1}{u} - 1 + O(\frac{1}{u^2})}) = 2(u+1) + O(\frac{1}{u})$$

For $u \gg 1$, the expected number of equidistant points and the expected number of steps are approximately $2(u+1)$ and $(2+\frac{1}{u})\sqrt{c}$. Compared to Section 3.5.2, for $u = 10$ equidistant points, the first Pollard-$\lambda$ variant needs on average $2.93\sqrt{c}$ steps. If we would also use $2(u+1) = 10$ equidistant points for Concurrent Pollard-$\lambda$ such that $u = 4$, then we need on average $2\sqrt{c}(1 + \frac{1}{2u}) = 2.25\sqrt{c}$ steps. So Concurrent Pollard-$\lambda$ is superior to the first Pollard-$\lambda$ variant with checkpoints.

[10] For distinguished points the result will be almost the same, except that we need twice as many of them.

[11] This holds for both sequences since we do not know which sequence is behind, such that both the known and the unknown sequence start stepping simultaneously.

# Chapter 4

# Data Structure for Collision Search

In this Section we introduce a dedicated hash table for collision search. It is called ReStore[1]. For an introduction to hash tables, see Section 2.4. In Section 4.1, we first describe, how it works. Then in Section 4.2, we give the abstract data type and an implementation. In Section B we have an example of how ReStore can be used with Baby-step Giant-step. Finally, in Section 4.4, we present test results to show the performance.

## 4.1 Description

Compared to a hash table, ReStore is in fact simpler. However, these simplifications will result in a time-space tradeoff for performance analysis. For collision search we want to store encodings $\mathbf{g^v}$ of vectors $\mathbf{v}$. More generally, if we have the vectors -that really are tuples- ranked by $n(\mathbf{v}),\ n \in [0, c)$ then we have encoding function $\mathcal{E}(n) = \mathbf{g^v}$.

Let $s$ be the size of the reference table $R[\,]$, $m$ the maximal number of entries that have to be stored in $R[\,]$ and $r_h$ the number of tries for hashing. Instead of using an external hash function $h(\,)$, we simply truncate $\mathcal{E}(n)$ for rehashing up to $r_h$ times, defined by $h(n,r) = f(n, r_h)\ div\ 2^r\ mod\ s$ with $f(n, r_h) = N[\mathcal{E}(n)]_{s\ 2^{r_h}}$. We call $h(n,r)$ a rehashing function. In the implementation we will see that truncation is a fast operation made of one $AND$ and one bit shift operation.

To avoid hash collisions, we cannot use the vector information that is stored, since if we search for a collision to get a discrete logarithm vector, one of the vectors is a relative vector, while the other is absolute. One way to handle this, is to verify if a collision is not a hash collision. This means, that we evaluate the supposed discrete logarithm vector and verify it. The only way to reduce the probability of a hash collision without evaluation, is to put $b \in \mathbb{Z}$ bits of extra information from $N[\mathcal{E}(n)]$ in the hash table. We do this by splitting up $\mathcal{E}(n)$ into the parts

$$N[\mathcal{E}(n, r_h, b)] = (f(n, r_h),\ e(n, r_h, b),\ y(n, r_h, b))$$

$$= (\ N[\mathcal{E}(n)]_{s\ 2^{r_h}},\ N[\mathcal{E}(n)]\ div\ (s2^{r_h})\ mod\ 2^b,\ N[\mathcal{E}(n)]\ div\ (s2^{b+r_h})\ )$$

Because $r_h$ and $b$ are fixed, we write $(f(n),\ e(n),\ y(n))$ instead of $f(n, r_h),\ e(n, r_h, b),\ y(n, r_h, b))$. Now $e(n)$ contains $b$ extra bits of an encoding that can be considered pseudo random, such that the probability of a hash collision is reduced. The rest of the encoding, $y(n)$, is not used.

**Definition 4.1** *Let constant $b \in \mathbb{Z}$. A* pseudo collision *is a hash collision where also the $b$ extra bits of $e(..)$ for the encoding match.*

[1] Named after Rick, Sven and Sam, three of my pupils.

So, for every encoding that is stored, only with a probability of $O(2^{-b})$ we find a pseudo collision instead of a collision.

ReStore uses *open adressing*, which means that the reference table $R[\ ]$ is used directly for storing the indices $n$, instead of using a reference to another computer address. It is done by setting $R[h(n,r)] = n+1$, where the addition of 1 is to avoid the *null* (empty) value if $R[\ ] = 0$. We need at most $|m|$ bits for identifying $n$. The bits needed for $r$ limits the number of rehashes, and the bits for $e(n)$ limits the number of pseudo collisions. See Illustration 4.2.

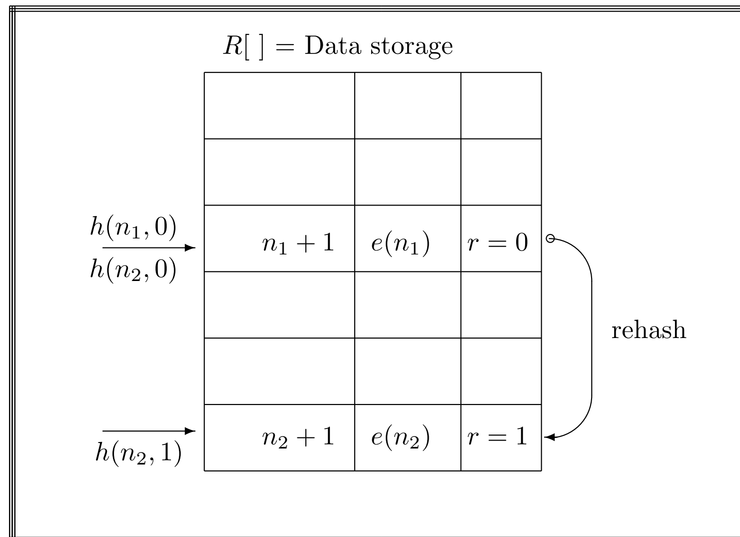


**Illustration 4.2** : ReStore

In Section 2.4.1 we defined *double hashing* by using a rehashing function $h(n,r)$ until either $r \geq r_h$ or we find that $R[h(n,r)] = 0$ with $r < r_h$. It is fast and it avoids *clustering* in $R[\ ]$. ReStore uses *double hashing*. After each rehash, numbered $r = 0,1,2,...,r_h - 1$, ReStore does $r_p$ probes using a secondary hash function $rh_p(n),\ r = 0,1,2,...,r_p - 1$. Only if there is still a storage collision after $r_{max} = r_h r_p$ tries, then data is lost. The complete double hash function is

$$h_d(n,r) = h(n, r\ div\ r_h) + (r\ mod\ r_p)h_p(n)\ mod\ s$$

For storage and lookup, we search the indices $r = 0,1,...,r_{max} - 1$ until either $r \geq r_{max}$ or there exists an $r < r_{max}$ such that $R[h_d(n,r)] = 0$. In ReStore, we only implemented $h_p(n) = 1$ so far, which makes the second stage a simple *linear probing*.

## 4.2 Abstract Data Type and Implementation

For encoding index $n \in [0,c)$, let $\mathcal{E}(n) = (f(n), e(n), y(n))$ where $h(n) \in \mathbb{Z}_s$ is a hash function and $e(n) \in \{0,1,2,...,2^b - 1\},\ b \leq |c|$ to reduce pseudo collisions. For rehashing, we use the abstract method $hash()$. See list 4.3 for the abstract methods. $R[\ ]$ is a table of entries $R[0], R[1], ..., R[s-1]$ that are addressed by different hash values. Clearly, if $s$ items have been stored, then set $S$ is full and $reserved(z)$ will return 0. For the representation invariant of $S$, we use table $R[\ ]$, in which $R[\ ] = (n+1,\ e(n),\ r)$. Now

$$S = \{\ j \mid \exists_{i \in \{1,...,s-1\}\ ;\ r \in \{0,1,2,...,r_{max}-1\}}\ (\ R[\ h_d(i,r)\ ] = (j+1,\ e(j),\ r)\ )\ \}$$

Now we discuss the public methods that are used in the Abstract Data Type. For further details, see the Java class ReStore in Appendix A. In list 4.4 , we recapitulate the functions, variables and constants that we use. At any time, the number of entries stored is less than

**List 4.3**: Abstract data type of ReStore

`new(` $s, \ r_{max}$ `) :` $S \leftarrow \emptyset$

`store(` $f(n),\ e(n),\ y(n),\ n$ `) : if storage(`$f(n), y(n)$`) then` $S \leftarrow S \ \cup \ \{(e(n), n)\}$

`similar(`$f(n), e(n)$`) : output` $\leftarrow \{ \ j \mid (e(j),\ j+1) \in S \ \wedge \ \exists_{r \in [0,1,\ldots,r_{max})} h_d(j,r) = h_d(n,r) \ \}$

`- private - storage(`$f(n), y(n)$`) :` $\exists_{r \in [0,1,\ldots,r_{max})}$ `reserved(hash(`$f(n), y(n), r$`))` $= 0$

`- private - reserved(`$z$`)` $:= \begin{cases} z > 0: & \texttt{max(reserved(}z-1\texttt{)} + |\texttt{similar(}z-1\texttt{)}| - 1,\ 0\texttt{)} \\ z = 0: & \texttt{max(}|\texttt{similar(}0\texttt{)}| - 1,\ 0\texttt{)} \end{cases}$

$s$. For $n \in \mathbb{Z}_m$ we have that $N[\mathcal{E}(n)] = (f(n), e(n), y(n))$. Array $R[\ ]$ contains entries as tuples $(n+1, e(n), r)$. For fixed $w$, we have $b = w - |r_{max}| - |m+1|$ bits left for pseudo collision reduction.

**List 4.4**: Variables and constants for ReStore

$w := 32$ : *word length in bits for hash table entries*
$r_p \in \{0, 1, ..., r_{max} - 1\}$ : *number of linear probes per rehash*
$r_h := r_{max}$ *div* $r_p$ : *number for rehashing*
$r \in \{0, 1, 2, ..., r_{max}\}$ : *number for rehashing and probing*
$R[0], R[1], ..., R[s-1]$ : *table with entries* $(n+1, e(n), r)$
$b := w - |r| - |m+1|$ : *number of bits free for reducing pseudo collisions*
$n \in \{1, 2, ..., m\}$ : *hash keys for given hash function* $h(n)$
$\mathcal{E}(n) = (f(n), e(n), y(n))$ : *representation of* $\mathcal{E}(n)$ *by a pseudo random triple*
$h(n, r) \in \mathbb{Z}_s$ : *hash value for table* $R[\ ]$ *index*
$e(n) \in \{0, 1, 2, ..., 2^b - 1\}$ : *bits to reduce pseudo collisions*
$y(n) \in \{0, 1, 2, ...\}$ : *rest of* $\mathcal{E}(n)$

**new**$(m,\ \lambda,\ r_{max})$ For $\lambda > 0$ a table $R[0], R[1], ..., R[s-1]$ of $s = \lceil \frac{m}{\lambda} \rceil$ words is reserved. Also, we set $r_h = \lfloor \sqrt{r} \rfloor$ and $r_p = r_{max}$ *div* $r_h$ and *double hash* function $h_d(n, r)$. Note, that for $\lambda > 1$, part of the input is lost. For our purpose, in Appendix A.3 the word length of ReStore is fixed to $w = 32$, since Baby-step Giant-step can already handle $(2^{32})^2 = 2^{64}$ elements with this setting.

**store**$(\mathbf{f}(\mathbf{n}), \mathbf{e}(\mathbf{n}),\ \mathbf{n}))$ For $r = 0, 1, 2, ..., r_{max} - 1$, the first empty table entry $R[h_d(n, r)]$ is searched. If no empty entry is found then input is lost. If for some $r \in \{0, 1, 2, ..., r_{max}\}$, we have $R[h_d(n, r)] = 0$ then the input is stored by $R[h_d(n, r)] = (n+1, e(n), r)$. Else the input it is lost.

**similar**$(\mathbf{f}(\mathbf{n}), \mathbf{e}(\mathbf{n}))$ For all $r = 0, 1, 2, ..., r_{max}$ look if $R[h_d(n, r)] = (n+1, e(n), r)$. Suppose that this is so for hash keys $n_1, n_2, ..., n_k$. Now we have that $h_d(n_1, r_1) = h_d(n_2, r_2) = ... = h_d(n_k, r_k)$ and $e(n_1) = e(n_2) = ... = e(n_k)$. We say that the elements $\mathcal{E}(n_1), \mathcal{E}(n_2), ..., \mathcal{E}(n_k)$ are *similar* to

**Algorithm 4.5**: Baby-step Giant-step using ReStore

1. ***input*** $g,\ h,\ c,\ \lambda,\ r_{max}$
2. $\mu := \lceil\sqrt{c}\rceil\ ;\ g_G := g^{\mu}\ ;\ v_G := 0\ ;\ t := 0\ ;\ S_B :=$***ReStore***$.new(\mu, \lambda, r_{max})$
3. ***for*** $v_B = 0, 1, ..., \mu$ ***do*** $S_B.store(N[hg^{-v_B}]_{\mu}, N[hg^{-v_B}]\ div\ \mu, v_B)$
4. ***do*** {
5. ***while*** $\forall_{v_B \in S_B.similar(N[g^t g_G^{v_G}])}(hg^{-v_B} \neq g^t g_G^{v_G})\ \wedge\ (v_G + \mu < c)$ ***do*** {
6. $v_G := v_G + \mu$ }
7. $t := t + 1$
8. } ***while*** $hg^{-v_B} \neq g^t g_G^{v_G}$
9. ***output*** $v := v_G + t + v_B$

$\mathcal{E}(n)$. The hash keys $n_1, n_2, ..., n_k$ are returned as an array. Note that for $b > 0$, then similar hash keys are only found once per $\Theta(2^b)$ requests, so this array is generally empty. The user has to verify whether $\mathcal{E}(n) = \mathcal{E}(n_i),\ i = 1, 2, ..., k$ to claim a true collision.

## 4.3 Example

We use ReStore for the basic version of Baby-step Giant-step for one coordinate, as given in Section 3.3. See algorithm 4.5. For encoding we have $\mathcal{E}(v) = g^v,\ v \in [0, c)$.

In line 2, the call $new(\mu, \lambda, r_{max})$ defines a data structure of type ReStore. In line 3, $S_B.store(N[hg^{-v_B}]_{\mu},\ N[hg^{-v_B}]\ div\ \mu,\ v_B)$ is used to store all indices $v_B \in \{0, 1, 2, ..., \mu\}$. Here we have $h(n) = N[hg^{-v_B}]_{\mu}$ and $(e(n), y(n)) = N[hg^{-v_B}]\ div\ \mu$. Given that $w = 32$, ReStore determines how many bits are left for $e(n)$.

In contrast to traditional hash tables, ReStore $S_B$ drops data if the hash value $N[g^{v_B}]_{\mu}$ of $v_B$ has a *storage collision* for more than $r_{max}$ hash values of previously stored indices. In fact, $r_{max}$ is an upper bound to the number of collisions that can be resolved. As a result Baby-step Giant-step fails sometimes, and we need to restart a loop for finding the discrete logarithm for $g^{-t}h,\ t = 0, 1, 2, ...$ successively, until a collision is found.

## 4.4 Efficiency

In table 4.6, some settings are shown together with the performance of ReStore. The first setting shows that ReStore can be used as a table that is lossless *in probability*. The last setting is also remarkable. It says that we have 25.8% loss if we use $\lambda = 1.25$, which means that we try to store 20% *more input* than the amount of available space.

For contrast, standard hash tables would become slow for $\lambda \in [0.8, 1)$, and break down for $\lambda \geq 1$. Moreover, they use a reference table of at least $|m|$ bits per entry on top of the $|m|$ bits for the hash keys. It is fair to compare ReStore to dedicated hash tables with *open addressing*, since we also directly store the index without using an extra reference table. At a rate of $\lambda = 0.8$ the space efficiency of a hash table with *open addressing* is 80%. However, the loss rate of inputs -which can be seen as a failure rate- makes ReStore useful for time-space tradeoffs.

**Example 4.7** ReStore can be used for $\lambda = 1.0$, with a loss of 5.9% if $r_{max} = 16$. We do not count the influence of different $r_{max}$ since all tables that use rehashing efficiently need at least the same overhead for this[2]. Therefore, we consider ReStore at least $100\% - 5.9\% = 94.1\%$ space efficient compared to other hash tables. At a failure rate of 5.9%, we reduce space by 20%. Since

[2]Hash tables for full storage need either more rehashes or an extra (linear) bin, to prevent loss of data.

| Setting for $\lambda$ and $r_{max}$ | average loss in % |
|---|---|
| $\lambda = 0.75,\ r_{max} = 256$ | 0.0000 |
| $\lambda = 0.8,\ r_{max} = 64$ | 0.004 |
| $\lambda = 0.8,\ r_{max} = 16$ | 0.7 |
| $\lambda = 0.8,\ r_{max} = 4$ | 8.9 |
| $\lambda = 1.0,\ r_{max} = 64$ | 2.1 |
| $\lambda = 1.0,\ r_{max} = 16$ | 5.9 |
| $\lambda = 1.0,\ r_{max} = 4$ | 15.9 |
| $\lambda = 1.25,\ r_{max} = 64$ | 20 |
| $\lambda = 1.25,\ r_{max} = 16$ | 20.2 |
| $\lambda = 1.25,\ r_{max} = 4$ | 25.8 |

**Table 4.6**: Efficiency of ReStore for $m = 2^{17}$ entries

for 20% space reduction we have to rerun a collision search with 5.9% probability, we now have a time-space tradeoff.

In the case of Baby-step Giant-step we only have to rerun the Giant steps at a shifted position, if the first run did not result in a collision. Since in the optimized case, the number of Giant steps is twice the number of Baby steps, we have that the average extra effort until collision is approximately $\frac{2}{3}\frac{1}{1-5.9\%} \approx 4\%$ in terms of multiplications, while space is reduced by 20%.

# Chapter 5

# Advanced Algorithms

By heuristic 3.3 in Section 3.1, we use the Birthday Paradox to explain why the bounded discrete logarithm vector is found in $\Omega(\sqrt{c})$ multiplications. For Baby-step Giant-step, we already have that the bounded discrete logarithm vector is $O(\sqrt{c})$. Until now we have one probabilistic algorithm, namely Pollard-$\lambda$, but it works only for the bounded discrete logarithm in one coordinate.

The first new development in this chapter is that we can guarantee that Pollard-$\lambda$ sequences join, which can be very useful in practice. This is done in Section 5.1. The next new development is a *probabilistic* bounded discrete logarithm vector algorithm that needs $\Theta(\sqrt{c})$ multiplications but less than $\Theta(\sqrt{c})$ space for elements from $G_q$. It is a so-called *hybrid* algorithm that combines Baby-step Giant-step with Pollard-$\lambda$, and it is explained in Section 5.2.

The last new development in this chapter is the use of restrictions on coordinates, to find the discrete logarithm vector faster. This is done in Section 5.3 and by a heuristic argument we think that in general, if the probability distribution on the coordinates is given, there is a lower bound for the number of multiplications.

## 5.1 Precomputation for Pollard-$\lambda$

For an introduction to Pollard-$\lambda$ , see Section 3.5. Not all runs for Pollard-$\lambda$ terminate after a fixed number of steps by finding the bounded discrete logarithm. In [2] Schoenmakers notes, that it is useful to know whether all of the unknown sequences starting in $[0, c)$ will join a known sequence starting at $c$ within a fixed number of steps. This is so if we need a time critical decoding. We will see that the precomputation time of such an algorithm given generator $g$ and integer $v \in [0, c)$ is significantly more than the time for doing one decoding. However, for repetitive decoding in the same setting, such an algorithm can be faster.

Since $G_q$ has finite order, it is even possible that two sequences starting differently but making steps by the same rule, will eventually cycle over $G_q$ without ever joining. If the function $N[g^v]_s$ for $g^v \in G_q$ is really pseudo random and $c \ll q$, then before the sequences are wrapped around $q$, a solution is almost surely found. However, to guarantee that we find a solution in a fixed number of steps, we present another approach.

### 5.1.1 A simple algorithm

We first start with a very simple but inefficient algorithm. If we run $\lambda$-sequences from different starting positions, they will eventually join in probability. Algorithm 5.1 finds a minimum set of ending positions for all sequences starting in $[0, c)$. We run $\lambda$-sequences with maximum step size $M$ and optimum average step size $\mu$. They successively start in $0, 1, 2, ..., c-1$ and we stop as soon as they arrive in the range $[2c, 2c+1, 2c+2, ..., 2c+M)$. From Section 3.5.2 it follows that most of the sequences will join one or more other sequences. Hence the number of endpoints in the range $[2c, 2c+1, 2c+2, ..., 2c+M)$ is smaller than $M$.

**Algorithm 5.1**: Minimum set of termination positions for $\lambda$ interval

1. ***input*** $s$
2. $\mathcal{S}(i) := 2^i, \; i = 0, 1, 2, ..., s-1$ ; $A := \emptyset$
3. $M := 2^{s-1}$ ; $b[0] := b[1] := ... := b[M] := 1$
4. ***for*** $j := 0$ ***to*** $c - 1$ ***do*** {
5. $\quad n := j$ ; $e = g^j$
6. $\quad$ ***while*** $n < 2c$ ***do*** {
7. $\quad\quad e = eg^{\mathcal{S}(N[e]_s)}$ ; $n = n + \mathcal{S}(N[e]_s)$
8. $\quad$ } $A := A \cup \{n\}$
9. } ***output*** $A$

**Heuristic 5.2** *On average $O(|M|)$ of all sequences starting in $[0, c)$ have not joined if they each do $\sqrt{c}$ more steps after passing $c$.*

**Argument** We determine the expected number of sequences that do not join any other sequence in $\Theta(\sqrt{c})$ steps. After a certain number of sequences have not joined, we expect that less than $\frac{1}{2}$ a sequence will not join any of the previous sequences. First note, that all sequences starting in $[0, c)$ step at least once in $[c, c+M)$. Let $\gamma$ be a constant. Let $n \leq M$ be the number of sequences that start in $[c, c+M)$ and that each does not join any of the other $n-1$ sequences before doing $\sqrt{c}$ steps of average size $\gamma\sqrt{c}$. As a result, we have $n$ different endpoints. Now run the other $M-n$ sequences starting in $[c, c+M)$. The probability that each of these $M-n$ sequences does not join any of the $n$ sequences is $(1 - \frac{n}{\gamma\sqrt{c}})^{\sqrt{c}} \approx e^{-n/\gamma}$. We expect that all of these $M-n$ sequences join the first $n$ sequences, if $(M-n)e^{-n/\gamma} \leq Me^{-n/\gamma} < \frac{1}{2}$, such that $n = O(|M|)$. ♣

This is a very good performance in terms of space. Unfortunately, all sequences first have to pass position $c$, which takes on average $\frac{1}{2}\sqrt{c}$ steps per sequence. After that they do another $\sqrt{c}$ steps each. Now the total number of multiplications for all sequences is $\Theta(c\sqrt{c})$. So in terms of multiplications, this algorithm is very inefficient.

### 5.1.2 An advanced algorithm

If a sequence $\{y\}_t$ has maximum step size $M$ then we know that given $y_0 \in [0, c)$, there exists $t$ such that $y_t \in [c, c+M)$. We need a definition for working with this concept.

**Definition 5.3** Attractor set. *A set $X$ is called an attractor set for $[0, c)$, if all sequences $\{y\}_t$ with $y_0 \in [0, c)$, there exists $t$ such that $y_t \in X$.*

Let $X$ be an attractor set. Trivially, $X = [0, c)$ is an attractor set for $[0, c)$. More interestingly, $X = [c, c+M)$ is an attractor set for $[0, c)$. From Section 3.5.2 we have that Pollard-$\lambda$ is fastest if $M = \Theta(\sqrt{c}|c|)$. Now we have that the attractor set for $[0, c)$ is $[c, c+M)$ which is of size $O(\sqrt{c}|c|)$.

**Heuristic 5.4** *Let $\mu$ be the average step size and $N \in \mathbb{N}$. If we have a set $X$ of attractors for $[0, c)$ that all have indices greater than $c$ with a maximum $N \geq c$, then we find the bounded discrete logarithm on average in at most $\frac{c}{2\mu} + \frac{N-c}{\mu}$ steps.*

**Argument** A sequence $\{y\}_t$ in the average case starts at position $y_0 = \lceil \frac{c}{2} \rceil$ with $h = g^{\lceil c/2 \rceil}$. Now we need on average $\frac{c}{2\mu}$ steps for a sequence to step in $[c, c+M)$. Next to that we need on average at most $\frac{N-c}{\mu}$ steps to collide with an attractor. ♣

**Definition 5.5** Survivor. *Let $N \geq c$. Let different sequences $\{y\}_t, \; t \in \mathbb{N}$ start subsequently from indices $y_0 = 0, 1, 2, ..., c-1$ and step until for some $t$ we have that $y_t \in [N-M, N)$. Let $X$ be*

**Algorithm 5.6**: Attractors

---

1. ***input*** $M$ ; $k \geq 1$ ; $s$
2. $\mathcal{S}(i) := 2^i, \ i = 0, 1, 2, ..., s-1$
3. $b[0] := b[1] := ... := b[M-1] := 1$
4. $base = c$; $n := M$ ; $last := 0$ ; $e = 1$
5. ***while*** $(n > k)$ ***do*** {
6.     ***for*** $i := 0$ ***to*** $M-1$ ***do*** {
7.         ***if*** $(b[i] := 1)$
8.             $e = eg^{base+i-last}$ ; $last = base + i$
9.             $b[i] := 0$ ; $j := i + \mathcal{S}(N[e]_s) \ mod \ M$
10.             ***if*** $(b[j] = 1)$ ***then*** $n := n - 1$ ;
11.                 ***else*** $(b[j] := 1)$
12.     } $base := base + M$
13. } $A := \emptyset$
14. ***for*** $i := 0$ ***to*** $M-1$ ***do***
15. ***if*** $(b[i] = 1)$ ***then*** $A := A \cup \{(g^{base+i}, \ base + i)\}$
16. ***output*** $A$

---

*the attractor set that contains the steps that these sequences made in $[N-M, N)$. Consider the sequence $\{x\}_t, \ t \in \mathbb{N}$ starting in $x_0 \in [0, c)$. Assume that after $t$ steps, $x_t > N$ without previously stepping on an attractor in $X$. Then $x_t$ is called a* survivor.

By Heuristic 5.2, the number of survivors has an upper bound of $O(1)$ if each sequence makes $\Theta(\sqrt{c})$ steps. To complete the subject of attractors, we describe an algorithm that finds $\Theta(1)$ attractors for $[c, c+M]$ in $O(\sqrt{c}|c|^3)$ steps, if $M = \Theta(\sqrt{c}|c|)$ as found in Section 3.5.2. See Algorithm 5.6. This algorithm starts at a window $[c + base, c + M + base)$. It shifts from $c + base$ to $c + base + 1$ after one step from the lowest position $c + base$, is done. Each time a step from $c + base$ ends on another position in the window that contains a survivor, the number of survivors is reduced by one. The window is of size $M$ by indexing $mod \ M$.

In line 2 we define the step sizes as proposed by Pollard. In line 3 we define a bit set that identifies the survivors, starting at integers $base$ to $base + M$. In line 3, we set the number of survivors to $M$. By line 5, the algorithm continues until there are $n \leq k$ survivors left. In the lines 6 to 11, each survivor makes one step and it is checked if it collides with another survivor. Note, that by setting bits for survivors $mod \ M$ in line 9 and by adding $M$ to the base in line 12, we continue with the survivors from positions $c + M$ on, until there are at most $k$ left.

In this algorithm, we did not consider the time to search the bit set for each next integer that has to be evaluated. For finding the lowest integer to make the next step with, we can use a Priority Queue. Then for $L \leq M$ survivors it takes $O(|L|)$ time to delete or store a node or to look for the next priority.

**Proposition 5.7** *Let $M = \Theta(\sqrt{c}|c|)$ be the maximum step size. Algorithm* Attractors *finds $\Theta(1)$ attractors for $[c, c+M)$ within $\Omega(\sqrt{c}|c|^2)$ and $O(\sqrt{c}|c|^3)$ multiplications, with $M = \Theta(\sqrt{c}|c|)$ bits space.*

**Proof** We start with $M$ attractors, that are indicated by a bit set with a base. For each survivor, we evaluate the integer by $O(|c|)$ multiplications and then we make one step. Since there are $M$ attractors over an interval of size $M$, the first step leads to joining with probability 1 . For the second, joining has probability $(1 - \frac{1}{M})$. For the third, the joining probability is $(1 - \frac{1}{M})^2$. As a

| interval length $c$ | $\frac{\textit{average \# steps}}{\sqrt{c}\lvert c\rvert}$ |
|---|---|
| $2^{12}$ | 0.34 |
| $2^{14}$ | 0.38 |
| $2^{16}$ | 0.60 |
| $2^{18}$ | 0.53 |
| $2^{20}$ | 0.47 |
| $2^{22}$ | 0.47 |
| $2^{24}$ | 0.45 |
| $2^{26}$ | 0.42 |
| $2^{28}$ | 0.39 |
| $2^{30}$ | 0.39 |
| $2^{32}$ | 0.67 |

**Table 5.9**: Average performance for $|c|$ survivors over 20 runs per $c$

result, after $kM$ steps in an interval of size $M$, per sequence we have a joining probability of

$$(1 - \frac{1}{M})^{kM} \approx e^{-k}$$

Now for $k = \Theta(|M|) = O(|c| + \|c\|) = O(|c|)$, we have $Me^{-k} = \Theta(1)$ survivors. Per step we need at least $\Omega(1)$ and at most $O(|c|)$ multiplications for evaluation. Now at least, we need $\Omega(1)kM = \Omega(\sqrt{c}|c|^2)$ multiplications. At most, we need $O(|c|)kM = O(\sqrt{c}|c|^3)$ multiplications. $\diamondsuit$

If we stop reducing attractors after $|c|$ survivors are left, then we see in table 5.8 , that our algorithm is approximately $\Theta(\sqrt{c}|c|)$ multiplications, which is faster than the bounds of Proposition 5.7 for finding $\Theta(1)$ survivors. If we have to find $n = O(c)$ discrete logarithms in the same setting, then the average number of steps is $\frac{1}{n}\Theta(n\sqrt{c} + \sqrt{c}|c|) = \Theta((1 + \frac{|c|}{n})\sqrt{c})$. Using the table, we conclude this section with the following heuristic.

**Heuristic 5.8** *Finding $n$ discrete logarithms with one setting, costs on average less than $(1 + \frac{|c|}{2n})\sqrt{c}$ multiplications.*

Now after a sequence has stepped a fixed minimum distance, we surely find the discrete logarithm.

## 5.2 A Hybrid of Baby-step Giant-step and Pollard-$\lambda$

In [2] and by [9], Schoenmakers suggests to split up vector coordinates for running different algorithms on them. We use this suggestion to develop a completely new algorithm. By redefining the vector $\mathbf{v} = (\mathbf{v}_D; \mathbf{v}_P)$ and the generator vector $\mathbf{g} = (\mathbf{g}_D; \mathbf{g}_P)$, we can run different strategies on both parts. In our case, the subscripts $_D$ and $_P$ denote a part for a deterministic method and a part for a probabilistic method. We can write $\mathbf{g}^{\mathbf{v}} = \mathbf{g}^{\mathbf{v}_D + \mathbf{v}_P} = h$. The forms $\mathbf{g}_D^{\mathbf{v}_D} = h\mathbf{g}_P^{-\mathbf{v}_P}$ or $\mathbf{g}_P^{\mathbf{v}_P} = \mathbf{g}_D^{-\mathbf{v}_D}h$ can be used as well, but then we have to be careful because the number of coordinates per vector change.

### 5.2.1 Baby-Giant $\lambda$

To minimize space requirements, we want to distinguish deterministic coordinates from probabilistic coordinates in such at a way, that we need the least amount of space. let $\pi(i)$ be a permutation such that $c_{\pi(1)} \leq c_{\pi(1)} \leq ... \leq c_{\pi(d)}$ are increasingly ordered and $d_1 \in \{1, 2, ..., d\}$. Now we have a number of $c_D := c_{\pi(1)}c_{\pi(2)}...c_{\pi(d_1)}$ deterministic coordinates, while for the probabilistic part $c_P := c_{\pi(d_1+1)}c_{\pi(d_1+2)}...c_{\pi(d)}$. Since Pollard-$\lambda$ only works over one coordinate, we take $d_1 = d - 1$ such that $c_P = c_{\pi(d)}$. This way, we are most efficient in terms of space.

Recall that in Extended Baby-step Giant-step, Section 3.3.2, first a Baby does $\mu = O(\sqrt{c})$ steps of unit size, and after that a Giant does $O(\sqrt{c})$ steps of size $\mu$. In the hybrid case, we split

off one coordinate $v_d \in [0, c_{\pi(d)})$ for Pollard-$\lambda$, such that Extended Baby-step Giant-step has to do $v_i \in [0, c_{\pi(i)}),\ i = 1, 2, ..., d-1$.

Now for each Baby step we run an unknown sequence for $\Theta(\sqrt{c_P})$ steps, and we store the end points. Then after each Giant step, we run a known sequence that does $\Theta(\sqrt{c_P})$ steps, and we check for a collision each step. See illustration 5.10.

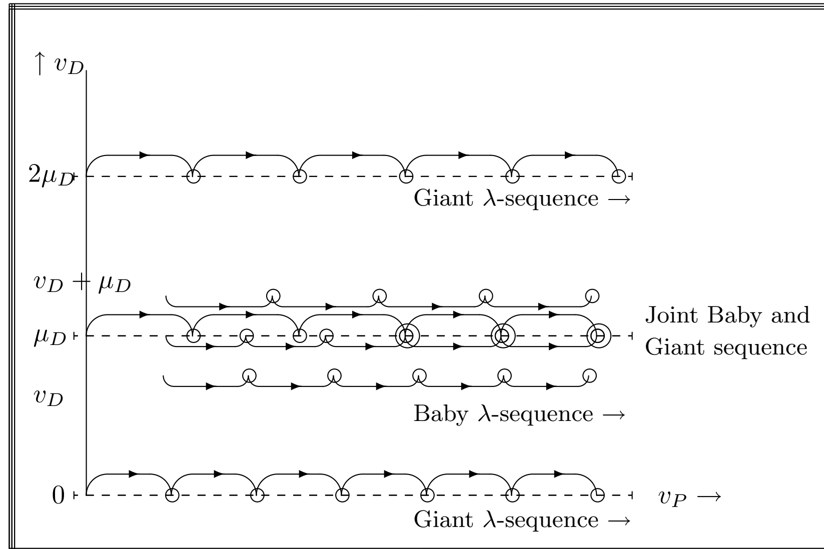


**Illustration 5.10** : Baby-Giant $\lambda$ for $d \geq 2$

We call this algorithm[1] Baby-Giant $\lambda$. For finding the bounded discrete logarithm vector in the optimal setting for Extended Baby-step Giant-step as described in the Sections 3.3.2 and 3.3.3, the $\lambda$-sequences have to run concurrently. This can be done, first by simulation of parallel processors for each Baby step and then by simulation of parallel processors for each Giant step. Then we need on average $\sqrt{2c_D}2\sqrt{c_P} = 2\sqrt{2c}$ multiplications, which is $\Theta(\sqrt{c})$. We implemented a simpler version of Baby-Giant $\lambda$ that runs $\lambda$-sequences for a limited number of steps. This version is a constant factor slower than optimum. With constant probability, one sequence starting at a Giant step joins a sequence starting at one of the Baby steps, such that a collision is detected. In algorithm 5.11 , we show the pseudo code.

In line 6, the results of $\lambda$-sequences are stored in a hash table $S_B$ for each Baby step, that contains truncations of $h\mathbf{g}_{\mathbf{D}}^{-\mathbf{v_D}(n_B)} g_P^{v_P} \in S_B[1]$ as hash values, and the vector or rank as the hash key. In line 9 we run $\lambda$-sequences for the Giant steps. In line 11 and 12, if hash value $t[1] \in S_B[1]$, then the vector $\mathbf{v} = \log_{\mathbf{g}}(h)$ can be reconstructed with the hash key $(t[2]; t[3])$. This algorithm does not always terminate successfully. To keep it simple, we have not included retries, nor concurrent $\lambda$-sequences with checkpoints.

For $d = 2$, the space requirements are very low. If $c_1 = c_2 = \Theta(\sqrt{c})$ then we need $\Theta(\sqrt{c_1}) = \Theta(c^{1/4})$ space. Compared to Baby-step Giant-step , this is a reduction by a factor $\Theta(c^{1/4})$. If $d > 2$ and $c_i = \Theta(c^{1/d})$, the advantage is less impressive[2].

[1]Schoenmakers first proposed to combine Baby-step Giant-step with Pollard-$\lambda$. With these hints, the algorithm is developed independently by Rezaeian Farashahi and the author.

[2]For $d = \frac{|c|}{4}$ and $c_i = c^{1/d}$, we only have a space reduction of approximately $\sqrt{2^4} = 4$ compared to Baby-step Giant-step on $d$ coordinates.

**Algorithm 5.11**: Baby-Giant $\lambda$ for $d \geq 2$

1. ***input*** $h$ ; $c_D$ ; $c_P$ ; $k$ ; $\kappa$ ; $\mathbf{g} := (\mathbf{g}_D; g_P)$
2. $s := \frac{|c|}{2} + \|c\| - 1$ ; $n_G := 0$ ; $\mu_D := \lceil \sqrt{c_D} \rceil$ ; $\mu_P := \lceil \sqrt{c_P} \rceil$ ; $S_B := \emptyset$
3. $f(e) := x + 2^{N[e]_s}$ *mod* $q$
4. ***for*** $n_B \in \{0, 1, 2, ..., \mu_D\}$ ***do*** {
5. ***for*** $v_P = 0$ ***to*** $\lceil k\mu_P \rceil$ ***do*** $v_P := v_P + f(h\mathbf{g}_{\mathbf{D}}^{-\mathbf{v_D}(n_B)} g_P^{v_P})$
6. $S_B := S_B \cup \{(h\mathbf{g}_{\mathbf{D}}^{-\mathbf{v_D}(n_B)} g_P^{v_P},\ n_B,\ v_P))$ }
7. ***do*** { $v_P := 0$
8. ***do*** { $v_P := v_P + f(\mathbf{g}_{\mathbf{D}}^{\mathbf{v_D}(n_G)} g_P^{v_P})$
9. } ***until*** $(v_P > \lceil (k + \kappa)\mu_P \rceil) \vee (\mathbf{g}_{\mathbf{D}}^{\mathbf{v_D}(n_G)} g_P^{v_P} \in S_B[1])$ }
10. $n_G := n_G + \mu_D$
11. } ***while*** $(n_G < c) \wedge (\ \exists_{t \in S_B}\ \mathbf{g}_{\mathbf{D}}^{\mathbf{v_D}(n_G)} g_P^{v_P} = t[1]\ )$
12. ***output*** $\mathbf{v} := (\mathbf{v}(n_G + t[2]); v_P - t[3])$

## 5.3 Exploiting a priori Knowledge

For different reasons, the bounded discrete logarithm vectors may not be uniformly distributed. If we have information about the bounded discrete logarithm vector, then it is still the question whether we can make use of it efficiently. To answer this question, in Section 5.3.1 we suppose that additional information is given by a set of *constraints*. Also, for a given probability distribution over the bounded discrete logarithm vectors, we make a heuristic statement about the lower bound. We finish with a case in Section 5.3.2. This case is from the electronic voting system in [1], and we explain how it can be solved efficiently in different ways.

### 5.3.1 Constraints on coordinates

As described in Section 1.2 for the bounded discrete logarithm vector, we have to find $\mathbf{v}$ for $h \in G_q$ such that $h = \mathbf{g}^{\mathbf{v}}$. The first set of constraints is naturally $(v_i \geq 0)$ and $(c_i - 1 - v_i \geq 0)$ for $i = 1, 2, ..., d$. Since these constraints have no dependencies between the $v_i,\ i = 1, 2, ..., d$ the basic algorithms for finding the bounded discrete logarithm vector will do. Now suppose we have a finite set of constraints

$$r_j(\mathbf{v}) \geq 0,\ j \in J$$

where each constraint has at least two of the $v_i,\ i = 1, 2, ..., d$ depending on each other. These constraints may vary from linear programming constraints to non linear constraints. Now it is not certain that the basic algorithms for uniformly distributed vectors will do.

Working with constraints is rather explicit. In general, we can consider the probability density function for $\mathbf{v}$. By the next definition, we have a uniform probability density function that fits to the case of constraints on the $v_i$.

**Definition 5.12** *Let* $[true] = 1$ *and* $[false] = 0$. *Let* $c_r = \sum_{\mathbf{v}} \prod_{j \in J} [r_j(\mathbf{v}) \geq 0]$ *be the number of vectors that fit to the constraints. Now* $\mathcal{P}_r(\mathbf{v}) = \frac{1}{c_r} \prod_{j \in J} [r_j(\mathbf{v}) \geq 0]$.

In other words, this function is a uniform distribution on the vectors that fit the constraints. In heuristic 3.3 of Section 3.1, we supposed a lower bound of $\Omega(\sqrt{c})$ for the uniform distribution with $H(\mathbf{v}) = \log_2(c)$ bits. Generally, a probability density function has probabilities ranging in $[0, 1]$ for each of the vectors. Therefore, we give a heuristic statement that includes the case of constraints.

**Heuristic 5.13** *Let $P(\mathbf{v})$ be a probability distribution for a finite set of vectors. Let $H(\mathbf{v}) = -\sum_{\mathbf{v}} P(\mathbf{v})\log_2 P(\mathbf{v})$ be the Shannon entropy in bits. Finding the bounded discrete logarithm vector costs $\Omega(2^{(H(\mathbf{v})/2)})$ multiplications.*

**Argument** Assume that we can indicate all $c$ possible vectors $\mathbf{v}(n)$ in decreasing probability by taking $n = 0, 1, 2, ..., c-1$. Now we can determine the encodings $\mathbf{g}^{\mathbf{v}(n)}$ in order of decreasing probability and run over the first $n \in [0, 2^{H(n)})$ vectors $\mathbf{v}(n)$, by each time making one multiplication to have another, less probable, encoding. As a result of this procedure, the efficiency for finding the most probable solution drops before a collision is found. This collision is found generally in $\Omega(\sqrt{c})$ steps, and $\Theta(\sqrt{c})$ steps only if the indices are equiprobable. Therefore, the least amount of multiplications is $\Omega(\sqrt{2^{H(n)}}) = \Omega(2^{H(\mathbf{v})/2})$. ♣

### 5.3.2 Case: sum of coordinates known

Suppose that we use the election scheme of [1]. If we know that the $v_i$ are *almost surely* within lower and upper bounds such that $v_i \in [l_i, u_i)$, then we use that $t = \sum_{i=1}^d l_i + \sum_{i=1}^d v_i'$, where $v_i' \in [0, u_i - l_i)$. So the problem is reduced to finding $v_i' \in [0, u_i - l_i)$ such that $t' = (t - \sum_{i=1}^d l_i) = \sum_{i=1}^d v_i'$. This is particularly useful, if we have statistics of previous elections, or if we have permission to do exit polls.

Now after reducing the intervals as far as possible, let the total amount of votes $t = \sum_{i=1}^d v_i$ be given. Naturally we have $v_i \in [0, t)$, and because $t = \sum_{i=1}^d v_i$ we have

$$h = \mathbf{g}^{\mathbf{v}} = g_1^{v_1} g_2^{v_2} ... g_d^{v_d} = (g_d^{-1} g_1)^{v_1} (g_d^{-1} g_2)^{v_2} ... (g_d^{-1} g_{d-1})^{v_{d-1}} g_d^t$$

which for $\gamma_i = g_d^{-1} g_i$ and $k = h g_d^{-t}$ can be written as

$$k = \gamma_1^{v_1} \gamma_2^{v_2} ... \gamma_{d-1}^{v_{d-1}} \quad \textit{subject to} \quad \sum_{i=1}^{d-1} v_i \leq t$$

In particular, for two coordinates with $t_1 + t_2 = t$, the problem reduces to a one dimensional problem to solve $k = h g_2^{-t} = (g_2^{-1} g_1)^{t_1}$. This is simply solved by running Pollard-$\lambda$. For more than two coordinates, it is already mentioned in [1, footnote 4] that we can use Baby-step Giant-step. To date, other algorithms may be useful as well.

For more than two coordinates, at least in theory, we can still do a lot better. Let $d$ be a fixed number of coordinates. Suppose that we distribute $t$ votes over $d \ll t$ options. This can be done in by lining up all $t$ votes, and separating them in $d$ parts, with help of $d-1$ markers. Now the number of possible vectors is

$$\binom{t+d-1}{d-1} = \frac{(t+d-1)!}{t!(d-1)!} \approx \frac{t^{d-1}}{(d-1)!}$$

In Section 5.3.1, we already have a $\Theta(t^{(d-1)/2})$ multiplications problem if we take $v_i \in [0, t),\ i = 1, 2, ..., d-1$. However, for $d > 2$ coordinates, the factor $\sqrt{(d-1)!}$ can be considerable. If we manage to run algorithms only over the vectors within the constraints, then we reduce runtime to

$$\Theta( \frac{t^{(d-1)/2}}{\sqrt{(d-1)!}} )$$

multiplications, which is significantly faster.

We show how to be as efficient as possible, by using only $\Theta( \frac{t^{(d-1)/2}}{\sqrt{(d-1)!}} )$ multiplications and space. For example we take $d = 3$, since it is easy to apply to $d > 3$ as well.

**Example 5.14** Let $d = 3$, such that we have to solve $k = \gamma_1^{v_1}\gamma_2^{v_2}$, subject to the restriction $v_1 + v_2 \leq t$. See illustration 5.15. By coordinate reduction the number of possibilities reduces to $c = t^2$, but using the restriction we have $c_r \approx \frac{1}{2}t^2$. Hence, if we take $\mu_r \approx \frac{t}{\sqrt{2}}$ for the Giant step sizes then we can approximate the area within constraints by squares, each containing a Giant step. One square, starting at an unknown position, contains all Baby steps. At certain point, a Giant step lands on one of the Baby steps and a collision is found. As a result, the number of steps is approximately reduced by a factor $\sqrt{2}$.

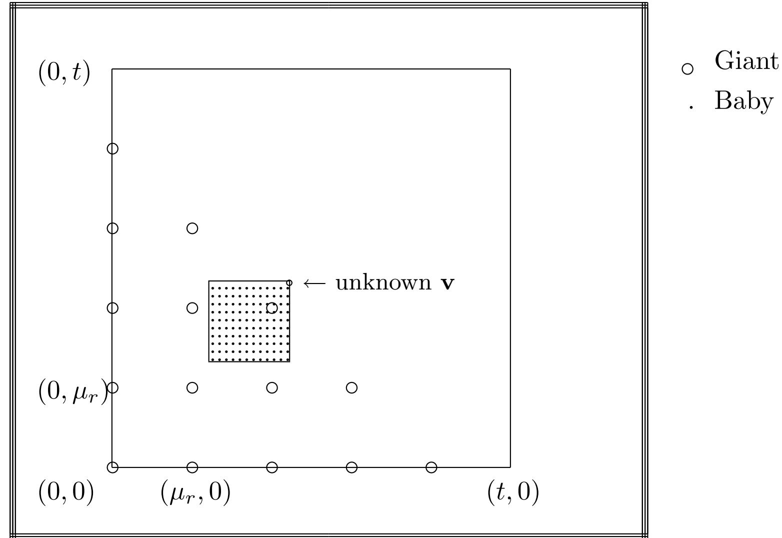


**Illustration 5.15** : Restricted Baby-step Giant-step

Note, that a combination of Baby-Giant $\lambda$ with restrictions on $d-1$ coordinates can be even better in terms of space. For example if $v_1 = t$ for Pollard-$\lambda$ and $t = \sum_{i=2}^{d} v_i$ for Baby-step Giant-step, then we need $\Theta(\frac{t^{(d-1)/2}}{\sqrt{(d-2)!}})$ steps, and only $\Theta(\frac{t^{(d-2)/2}}{\sqrt{(d-2)!}})$ space.

# Chapter 6

# Parallel Algorithms

Parallel algorithms are useful, when the computations that need to be done can be divided between computers that are interconnected. Suppose that the performance of a parallel algorithm is linear with the number of processors and each processor can do part of the computations on its own. Then even a modest internet connection is fast enough to distribute a problem between a multitude of processors for solving a computational problem in parts. Not all computational problems can be divided this efficiently, but in the case of a bounded discrete logarithm, it is possible.

We use some hardware structures for parallel computing. They are defined according to [18].

**Definition 6.1** *A* computer cluster *is a group of loosely coupled computers that work together closely so that in many respects they can be viewed as though they are a single computer.*

**Definition 6.2** Distributed computing *is decentralized and parallel computing, using two or more computers communicating over a network to accomplish a common objective or task.*

**Definition 6.3** *We speak of* shared memory *when processors in a multi-processor system have access to one large block of memory.*

In this chapter, we use the problem description of Section 1.2. First in Section 6.1 we explain how Baby-step Giant-step can be divided between interconnected computers. In Section 6.2 we explain how this is done for Pollard-$\rho$. Then in Section 6.3 Pollard-$\lambda$ follows. For Pollard-$\lambda$, we explain a new parallel algorithm that has -at least in theory- advantages over an earlier algorithm.

## 6.1 Parallelization of Baby-step Giant-step

Again we use the setting of Section 1.2 here, and we have to find $\mathbf{v}$ such that $h = \mathbf{g}^{\mathbf{v}}$. We found a new variant that is faster than the optimized Baby-step Giant-step in Section 3.3.3. The new idea is to do Baby steps and Giant steps concurrently and store the group elements for both Babies and Giants, until a collision is detected. In the case of parallel processors, all processors need to have fast access to the storage such that we need a *computer cluster* for this.

The simplest version of this algorithm also runs on a single processor, if we simulate parallel computation by doing Baby steps and Giant steps alternately. See Algorithm 6.4 . For more processors, the loop between the lines 3 to 8 can be parallelized.

**Proposition 6.5** *If we do Baby steps and Giant steps concurrently and store them, then the number of steps until collision is on average* $\frac{4}{3}\sqrt{c} - 1$*. Also, we need an average storage for* $\frac{4}{3}\sqrt{c} - 1$ *evaluations.*

**Proof** We rank both the Baby steps and the Giant steps, such that each Baby or Giant step takes one multiplication. The Baby step size for this ranking is an increment by 1, and the Giant step size is $\mu = \lceil\sqrt{c}\rceil \in [\sqrt{c}, \sqrt{c} + 1)$. We count the number of steps done until collision. Suppose that

**Algorithm 6.4**: Concurrent Baby-step Giant-step

1. ***input*** $g,\ h$
2. $g_G := g^{\mu}$ ; $v_B := 0$ ; $v_G := 0$ ; $S_B = \emptyset$ ; $S_G = \emptyset$
3. ***do*** {
4. $\quad v_B := v_B + 1$
5. $\quad S_B := S_B \cup \{(hg^{-v_B}, v_B)\}$
6. $\quad v_G := v_G + 1$
7. $\quad S_G := S_G \cup \{(g_G^{v_G}, v_G)\}$
8. } ***until*** ( $\exists_{t \in S_B}(g_G^{v_G} = hg^{-t[2]}) \vee \exists_{t \in S_G}(g_G^{t[2]} = hg^{-v_B})$ )
9. ***if*** $(g^{\mu t[2]+v_B} = h)$ ***then*** $v = \mu t[2] + v_B$
10. ***else*** $v = \mu v_G + t[2]$
11. ***output*** $v$

we have done $j$ Baby steps and $j$ Giant steps, without finding a collision. Now if we do one more Giant step and after that one more Baby step, we have a probability of $\frac{j}{\mu^2} + \frac{j+1}{\mu^2} = \frac{2j+1}{\mu^2}$ to find a collision. Let $J$ be the total of either Giant or Baby steps until collision. Then the expected number of steps until a collision is approximately

$$E(J) = \sum_{j=0}^{\mu} j(\frac{2j+1}{\mu^2}) = \frac{1}{\mu^2}\sum_{j=1}^{\mu} j + \frac{2}{\mu^2}\sum_{j=1}^{\mu} j^2 =$$

$$\frac{1}{2\mu^2}(\mu^2+\mu) + \frac{2}{\mu^2}\frac{1}{6}(2\mu^3 - 3\mu^2 + \mu) \approx \frac{2}{3}\mu - \frac{1}{2} \quad (\mu \ large)$$

So altogether the Babies and Giants do on average $2(\frac{2}{3}\mu - \frac{1}{2}) = \frac{4}{3}\sqrt{c} - 1$ steps. Since we store all evaluations, on average we also need $\frac{4}{3}\sqrt{c} - 1$ space. $\diamondsuit$

This variant needs only $\frac{4/3}{\sqrt{2}} \approx 94\%$ of the optimized runtime found in Proposition 3.10 of Section 3.3.3. However, in the worst case we need $2\sqrt{c}$ space for this new variant, while the variant in Section 3.3.3 needs $\sqrt{\frac{c}{2}}$ space.

## 6.2 Parallelization of Pollard-$\rho$

Here we have to find an integer $v \in [0, q)$ such that $h = g^v$ with $m$ processors. For a description of Pollard-$\rho$, see Section 3.4. The idea for parallelization of Pollard-$\rho$ is found in [5, §4]. We will see that the joining of sequences -which has the shape of a $\lambda$- is more likely than a cycle -which has the shape of a $\rho$- in one of the sequences. To refer to the different problems that are implied by the names Pollard-$\rho$ and Pollard-$\lambda$, we still use the name Pollard-$\rho$ for this parallel version. From Section 3.4, we know the average number of steps that has to be done before a collision occurs. A collision means, that either one of the sequences starts to cycle or two of the sequences join. Given $m$ processors, we can choose for an average of $w$ distinguished points per processor by applying a suitable criterion for distinguished points.

Instead of one sequence of average length $\sqrt{\frac{\pi q}{2}}$ which has a cycle of average length $\sqrt{\frac{\pi q}{8}}$, we start $m$ different processors for an average of $\frac{1}{m}\sqrt{\frac{\pi q}{2}}$ steps each. We choose for an average total of $u = mw$ checkpoints until collision. Because we use on average $w$ *distinguished points* per processor, we restart sequences at a new point after $\frac{1}{u}\sqrt{\frac{\pi q}{2}}$ steps and we repeat this on average $w$ times until a collision is found. After a processor has arrived at a *distinguished point*, we store it and start a new sequence. This restarting stops if two of the sequences have joined or if one

sequence has made a cycle, and then we find the discrete logarithm in $\frac{q-1}{q}$ of the cases[1]. For certain probability $p$ of finding a *distinguished point* in one step, the probability of finding no distinguished points per processor during a sequence of average length, is $(1-p)^{w/p} \approx e^{-w}$.

Now we determine the probability that processors run idle. The probability of finding a distinguished point in one step is $p = u\sqrt{\frac{2}{\pi q}}$. Let $K$ be the random variable for the number of steps until a cycle starts, and let $k > 0$. In the following analysis, sequences are handled as if they have a real-valued length while in fact sequences only have integer lengths. From Fact 2.5, but now for sequences that are a factor $mw$ shorter, we have that $P(K > \frac{k}{mw}\sqrt{\frac{\pi q}{2}}) \approx 1 - e^{-\frac{\pi}{4}(\frac{k}{mw})^2}$. Therefore,

$$P(K = \frac{k}{mw}\sqrt{\frac{\pi q}{2}}) \approx \frac{\pi}{2(mw)^2} k e^{-\frac{\pi}{4}(\frac{k}{mw})^2}$$

Now the *failure probability* is the probability that a single sequence ends up in a cycle *without detection*. It is

$$\frac{\pi}{2}\int_0^\infty \frac{k}{(mw)^2} e^{-\frac{\pi}{4}(\frac{k}{mw})^2} e^{-kw} dk$$

We have, that each processor finds on average $w$ distinguished points. For small failure probabilities, the average number of processors that end up running idle before a collision, is approximately

$$mw\frac{\pi}{2}\int_0^\infty \frac{k}{(mw)^2} e^{-\frac{\pi}{4}(\frac{k}{mw})^2} e^{-kw} dk$$

Already for $m = 20,\ w = 10$ this average number is less than $7.8 \cdot 10^{-5}$, so for this parallel version of Pollard-$\rho$ we do not have to consider undetected cycles and processors running idle. Note that we still need $w \gg 1$ to make sure that all processors have stored a few distinguished points before the total of $\sqrt{\frac{\pi q}{2}}$ steps is done. We need on average $w$ distinguished points until collision and from 2.3 we known that we have to do the average number of steps in a sequence to detect a collision. Now the average number of multiplications per processor is

$$\frac{1}{m}\sqrt{\frac{\pi q}{2}}(1 + \frac{1}{w}) = \frac{1}{m}\sqrt{\frac{\pi q}{2}}(1 + \frac{m}{u})$$

Because in most cases, the average number of distinguished points $um$ is not so great, processors can use *distributed computing* to sent distinguished points to a *shared memory*, either for storage or for collision detection.

**Example 6.6** If we have $w = 5$ and $m = 10,000$ processors connected, then we have to store on average $50,000$ checkpoints. Say, that we need 160 bits to store an 80 bits truncated encoding together with an 80 bits integer. Now these checkpoints altogether need 1 Mega bytes, and this easily fits in a cache. The network capacity is no problem at all. Suppose that the computation of a 80 bits discrete logarithm by $10,000$ processors takes four minutes. Then the network needs a capacity of only $\frac{10^6}{240} \approx 280,000$ bits per second.

We finish this section with the special case in which we have to find $n \ll q$ different discrete logarithms $v_1,\ v_2, ..., v_n$ given $h_1 = g^{v_1},\ h_2 = g^{v_2},\ ...,\ h_n = g^{v_n}$ in one setting.

**Heuristic 6.7** *For finding $n \ll q$ discrete logarithms in one setting with parallel Pollard-$\rho$ and $m$ processors, the average number of steps per discrete logarithm is approximately $\frac{1}{m}\sqrt{\frac{\pi q}{2n}}$ multiplications.*

**Argument** This argument is very similar to the explanation of Baby-step Giant-step, where a rectangle with area $q$ has one side for Baby steps and the other side for Giant steps. The difference is, that we now need that a sequence collides with any of the steps that are done previously by

[1] In $\frac{1}{q}$th of the cases, we have to restart one of these sequences until we find another cycle or joining

any of the processors. Thus we have a triangular area that grows each step. As this triangular area grows, we find more collisions.

If we search for collisions, then we need to do $\sum_{t=0}^{t_1} t = \frac{1}{2}t_1(t_1-1) = \frac{1}{2}t_1^2 - \frac{1}{2}t_1$ pseudo random steps until the first collision. From Fact 2.6 we know that the first collision is on average when approximately $t_1^2 = \frac{\pi q}{2}$. For the next collision, we need that $\sum_{t=t_1+1}^{t_2}(t-1) = (\sum_{t=t_1}^{t_2} t) - (t_2 - t_1) = \frac{1}{2}(t_2^2 - t_1^2) - t_2 + t_1 = \frac{\pi q}{2}$, which means that in the worst case $2\sum_{t=0}^{t_2} t = t_2^2 - 2t_2 = 2t_1^2 = \pi q$. So $t_2 \approx \sqrt{2}t_1$. Next we have in the worst case, that $2\sum_{t=0}^{t_3} t = t_3^2 - 3t_3 = 3t_1^2 - 3t_3 = 3\frac{\pi q}{2}$, such that $t_3 \approx \sqrt{3}t_1$ and generally $t_n \approx \sqrt{n}t_1$. To have a good approximation, we need that $nt_n \approx n\sqrt{n}t_1 \ll nt_1^2$. Hence $n \ll q$.

If we divide $t_n = \sqrt{n}t_1$ by $n$ then we have $t_n = \frac{t_1}{\sqrt{n}}$. ♣

Note that in [5, §4.1], the steps for finding more discrete logarithms with Pollard-$\rho$ is mentioned in a different context. There, the time to find a useful collision depends on the probability that a collision is useful. Since this probability is very close to 1, we neglected it in our analysis.

## 6.3 Parallelization of Pollard-$\lambda$

Now we restrict to finding an integer $v \in [0, c)$ such that $h = g^v$. First in Section 6.3.1, we show a technique from [5, §5]. As we will see, this technique leads to a small but significant loss of efficiency. This loss of efficiency can be reduced by a new and faster technique, which we explain in Section 6.3.2.

### 6.3.1 A First Technique

In [5, §5], Wiener and Van Oorschot show an algorithm for Pollard-$\lambda$, that for $m$ processors reduces runtime by a factor $m$. In Section 3.5.3, the case of $m = 1$ is done. In this section, from $m$ processors[2], $\frac{m}{2}$ known sequences start around $\frac{c}{2}$. At the same time, $\frac{m}{2}$ sequences start around unknown discrete logarithm $\log_g(h)$.

Let $\gamma$ be a constant and $\mu = \gamma\sqrt{c}$ the average step size. The distance between the known and the unknown sequences is on average $\frac{c}{4}$, which boils down to an average of $\frac{c}{4\mu} = \frac{\sqrt{c}}{4\gamma}$ steps per sequence. Since we do not know whether the known or the unknown sequences start ahead, the known and the unknown sequences run concurrently.

Suppose that the distance between the known and the unknown sequences is done, that is, known sequences can join unknown sequences and vice versa. The probability of joining between one known and one unknown sequences is $\frac{(m/2)^2}{\mu} = \frac{m^2}{4\gamma\sqrt{c}}$ per step. Let $T$ be the random variable for the number of steps until collision. Then $T$ is distributed geometrically by $P(T = t) = \frac{m^2}{4\gamma\sqrt{c}}(1 - \frac{m^2}{4\gamma\sqrt{c}})^{t-1}$, with an average of $E(T) = \frac{4\gamma\sqrt{c}}{m^2}$ steps per sequence.

Together, we have that each sequence on average makes $\frac{\sqrt{c}}{4\gamma} + \frac{4\gamma\sqrt{c}}{m^2}$ steps until a collision between a known and an unknown sequence is found. For $\gamma = \frac{m}{4}$ we have an average step size of $\frac{m}{4}\sqrt{c}$ and a total minimum of $\frac{2}{m}\sqrt{c}$ steps.

Unfortunately, instead of collisions between known and unknown sequences, it is more probable that two known sequences or two unknown sequences join[3]. In such cases, one of the sequences has to restart nearby the last distinguished point, which causes loss of efficiency. Suppose that we store on average $u = mw$ checkpoints together for both sequences, which is equal to one distinguished point every $\frac{1}{w}\frac{2}{m}\sqrt{c}$ steps. Then for the probability that two known or two unknown sequences have joined before or at a distinguished point, is $1 - e^{-1/w} = \frac{1}{w} + O(\frac{1}{w^2})$. Since we drop one of the two sequences that joined, for $w \gg 1$, approximately $\frac{1}{w}$ of the efficiency is lost. Also, on average we have to do $\frac{1}{w}\frac{2}{m}\sqrt{c}$ more steps per sequence to arrive at a distinguished point after one

[2] In case $m$ is odd, we can *simulate* two processors on each processor

[3] This is detected by one bit per distinguished points that tells if it is from a known or an unknown sequence.

known and one unknown sequence have joined. Hence, the average number of multiplications per processor is approximately

$$\frac{2\sqrt{c}}{m}(1+\frac{2}{w}) = \frac{2\sqrt{c}}{m}(1+\frac{2m}{u})$$

This formula looks different from the result given at the end of [5, §5.1]. The first reason is, that we also take the number of steps that is lost by joining of two known or two unknown sequences into account. The second reason is, that we use an average amount of distinguished points *relative* to the expected runtime. Our formula is more practical for making a time-space tradeoff.

### 6.3.2 An Advanced Technique

We present a model for parallel computation, where collisions between two known or two unknown sequences are not a problem, but even favorable for a shorter average runtime. At least in theory[4], we obtain a performance that is significantly better than the first technique for the same amount of distinguished points.

**Description** We take $m = 2n$ processors, if necessary by simulating 2 processors on each single one. Again we let all sequences run concurrently and we store on average $u = mw$ distinguished points. Contrary to [5, §5], we start the known sequences at equal distance over $[0, c)$, at the positions

$$\lceil\frac{c}{n}\rceil,\ \lceil\frac{2c}{n}\rceil, ...,\ \lceil\frac{(n-1)c}{n}\rceil,\ c$$

By spreading the known starting positions this way, any unknown sequence starts at a distance of at most $\lceil\frac{c}{2n}\rceil$ from one of the known sequences. At the same time, we start $n$ unknown sequences spread out over an interval of size $\lceil\frac{c}{n}\rceil$, with a distance of size $\lceil\frac{c}{n^2}\rceil$ between them. The first unknown starts at $\log_g(h) + \lceil\frac{c}{2n^2}\rceil$, and the last at $\log_g(h) + \lceil\frac{c}{n} - \frac{c}{2n^2}\rceil$. Since this so called *window* (of unknown sequence starting positions) is shifted by unknown coordinate $\log_g(h) \in [0, c)$, we have an average minimum distance of $\frac{c}{4n^2} = \frac{c}{m^2}$ between one of the known starting positions and one of the unknown sequence starting positions. See illustration 6.8.

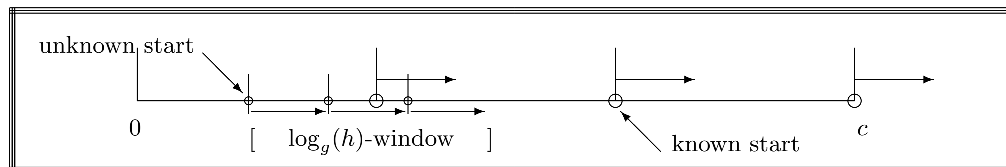


**Illustration 6.8** : Parallelization over one coordinate

The smallest distance between one known and one unknown is on average $c' = \lceil\frac{c}{m^2}\rceil$. From Section 3.5.2 it follows that on average a known and an unknown sequence need to step along for $\mu$ steps until they join. Because all sequences run at the same time, we can minimize the number of steps per sequence. For the total number of steps per sequence $\frac{c'}{\mu} + \mu$, we have a minimum for $\mu = \sqrt{c'} = \frac{1}{m}\sqrt{c}$ such that $\frac{c'}{\mu} + \mu = \frac{2}{m}\sqrt{c}$.

**Implementation** Algorithm 6.9 shows the pseudo code of algorithm Parallel Pollard-$\lambda$ in Appendix A. For simulation of parallel processors, each sequence makes one step on turn until a collision is found[5]. Here we use equidistant points instead of distinguished points, without a special reason. We choose a total of $u = mw$ distinguished points average, such that we have

[4]When finishing this thesis, we found that our test results of this algorithm are unreliable so we left them out.

[5]For analysis of averages this is not a problem, since in practice the processors are not much different. All sequences (read: processors) have to run until a collision is found. Sometimes, a faster processor will find a collision faster and sometimes, a slower processor will find a collision slower. So for the average runtime, only the average processor speed counts.

**Algorithm 6.9**: Parallel Pollard-$\lambda$

1. ***input*** $m$ ; $u$ ; $s$
2. $n := \frac{m}{2}$ ; $r := \frac{1}{n\ u}\sqrt{c}$ ; $t := 0$
3. ***for*** $i = 0$ *to* $n-1$ ***do*** {
4. $\quad x_{i,0} := \frac{ic}{n}$
5. $\quad y_{i,0} := \frac{(2i+1)c}{2n^2}$ }
6. ***do*** {
7. $\quad$ ***if*** $(t \bmod r = 0)$ ***then*** {
8. $\quad\quad S_k := S_k \cup (x_{i,t}, g^{x_{i,t}})$
9. $\quad\quad S_u := S_u \cup (y_{i,t}, hg^{y_{i,t}})$ }
10. $\quad$ ***if*** $\exists_{e \in S_u}(g^{x_{i,t}} = S_u[2])$ ***then return*** $(x_{i,t} - e[1])$
11. $\quad$ ***if*** $\exists_{e \in S_k}(hg^{y_{i,t}} = S_k[2])$ ***then return*** $(e[1] - y_{i,t})$
12. $\quad$ ***for*** $i = 0$ *to* $n-1$ ***do*** {
13. $\quad\quad x_{i,t+1} := x_{i,t} + \mathcal{S}(N[g^{x_{i,t}}]_s)$
14. $\quad\quad y_{i,t+1} := y_{i,t} + \mathcal{S}(N[hg^{y_{i,t}}]_s)$ }
15. $\quad t := t+1$
16. } ***while*** $(true)$

$r = \frac{1}{nw}\sqrt{c} = \frac{2}{u}\sqrt{c}$ steps per equidistant point. For the implementation of step sizes $\mathcal{S}(..)$, see Pollard-$\lambda$ in Section 3.5.2.

**Efficiency Analysis** In this algorithm, even if two unknown sequences join, we do not have any loss of efficiency. This is so, because our analysis is based on one known and one unknown sequence that start at nearest distance. On the other hand, if two unknown sequences join then we can stop running one of them. On average, this happens only if no known and unknown sequence have joined earlier. The probability for this is $(1 - \frac{m}{\sqrt{c}})^{\frac{2}{m}\sqrt{c}} \approx e^{-2}$. Then in $\frac{\frac{m}{2}-1}{\frac{m}{2}} = \frac{m-2}{m}$ of these cases we have that two unknown sequences join first, but the extra runtime is now half the a priori expected runtime. So we think that the efficiency compared to the first technique can be raised by

$$\xi_1 = \frac{1}{2}\frac{m-2}{m}e^{-2} \to \frac{1}{2}e^{-2} \approx 0.068 \qquad (m \to \infty)$$

In the algorithm Parallel Pollard-$\lambda$ that we show, only for simplicity we did not show the stopping of unknown sequences that join other unknown sequences. However, this stopping is implemented in the complete algorithm that is used for testing.

There is another reason why the advanced technique is more efficient than the first technique. We always have that either two unknown sequences start around a known sequence, or that there are two unknown sequences that both are close to a known sequence. Because the unknown sequences can join (one of) the (two) known sequence(s), we have a higher probability to find a collision each step. We want to know how much the efficiency is raised compared to the case of only one unknown sequence. Therefore, we have to express the probability that the shortest of the two distances did not result in the first collision, because the longest of the two distances results in an earlier collision. Also, we have to find how much longer the sequences at shortest distance would have run. Let $\delta \in [0, 1)$. We have that after $(1+\delta)\frac{\sqrt{c}}{m}$ steps, the nearest unknown sequence that starts approximately $(1-\delta)\frac{\sqrt{c}}{m}$ steps away from a known sequence, has not joined this known sequence with probability $e^{-2\delta}$. The probability that the unknown sequence that starts approximately at $(1+\delta)\frac{\sqrt{c}}{m}$ steps away from a known sequence has not joined earlier, is $1 - e^{-\delta}$.

We think that the gain in efficiency is

$$\xi_2 = \int_0^1 (1-\delta)e^{-2\delta}(1-e^{-\delta})d\delta \approx 0.056$$

Just as in the first technique, we also have that the efficiency drops because the sequences that join still have to be detected at a distinguished point. This decreases efficiency by $\frac{1}{w} + O(\frac{1}{w^2})$. Now for $m \gg 4$ and a total of $w \gg 1$ distinguished points, $\xi_1,\ \xi_2 \in [0,1]$, the number of multiplications per processor before collision is

$$\frac{2}{m}\sqrt{c}(1+\frac{1}{w})(1-\xi_1)(1-\xi_2)$$

Even if $\xi_1,\ \xi_2 = 0$ we need only half of the distinguished points for the same performance as the first technique. Because $\xi_1,\ \xi_2 > 0$, the advanced algorithm should be more efficient. Another practical advantage of this advanced algorithm is, that we do not have to restart sequences. The efficiency analysis is summarized by the following heuristic.

**Heuristic 6.10** *For* $w = \frac{u}{m} \gg 1,\ m \gg 2,\ \xi_1 \approx 0.068,\ \xi_2 \approx 0.056$, *the number of multiplications is on average*

$$\frac{1.76}{m}\sqrt{c}(1+\frac{m}{u})$$

We have not yet found reliable test results that verify Heuristic 6.10.

# Chapter 7

# Conclusions

## 7.1 Useful Results

We investigated ways to find the bounded discrete logarithm vector efficiently. An important step is the formulation of the heuristics 3.3 and 5.13, that implies lower bounds to the number of multiplications for finding bounded discrete logarithm vectors efficiently.

As a result of these heuristic bounds, we first concentrated on generalizations and optimizations for Baby-step Giant-step. Also by the name ReStore, we presented a very efficient type of hash table that can be used for tradeoffs between time and space.

For Pollard-$\rho$ with distinguished points, there is a risk that sequences end up in a cycle without any distinguished point. This can be avoided by using *equidistant points*, by starting a new sequence after each distinguished point or by increasing the probability to find distinguished points.

We gave optimizations for Pollard-$\lambda$. Also, we showed how we can guarantee a maximum runtime of Pollard-$\lambda$ by precomputation of *attractors* for each discrete logarithm in $[0, c)$.

In particular, we optimized Baby-step Giant-step and Pollard-$\lambda$ for finding more than one discrete logarithm in the same setting. Finding more discrete logarithms in a setting with a single generator and a fixed interval $[0, c)$, is impressively faster than finding more discrete logarithms for different generators.

A milestone is our first hybrid algorithm Baby-Giant $\lambda$. It is as fast as Pollard-$\lambda$, but can be used for more than one coordinate while the space requirements are significantly lower than for Baby-step Giant-step[1]. Also, the use of constraints on coordinates is helpful to reduce runtime for finding the discrete logarithm.

Finally, parallel algorithms are explained for Pollard-$\rho$, Baby-step Giant-step, and Pollard-$\lambda$. We considered the efficiency of these algorithms as a function of the number of checkpoints, such that a time-space tradeoff is easily made. For Parallel Pollard-$\lambda$, we suggested an algorithm that, at least in theory, is easier, faster and more efficient in terms of space than the algorithm explained in [5, §5]. This Parallel Pollard-$\lambda$ algorithm should be already faster than Concurrent Pollard-$\lambda$, if we only *simulate* parallel processors.

## 7.2 Further Research

During the project, we had a few ideas that can be useful for further research. First of all, it is interesting to find out whether the heuristics 3.3 and 5.13 for lower bounds on the bounded discrete logarithm vector problem, can be proved.

In Section 5.1, we guaranteed a maximum runtime to find a discrete logarithm by precomputation. For Section 5.1, precomputation took $O(\sqrt{c}|c|)$ bits space. This storage for precomputation

[1] The idea that a combination of Baby-step Giant-step and Pollard-$\lambda$ should work, was the starting point for this thesis. Initially, there was hope to find an algorithm that needs $\Theta(\sqrt{c})$ steps and $\Theta(1)$ space.

can be reduced impressively, if we determine attractor sets for each of the known sequences in the advanced Parallel Pollard-$\lambda$ variant of Section 6.3.2. We can do the precomputations for finding each of the $\frac{m}{2}$ attractor sets serially. We think that simulation of $m = \Theta(c^{1/4})$ processors can reduce the optimum average step size by a factor $\Theta(c^{1/4})$. As a result it will cost at most $\Theta(c^{1/4}|c|)$ bits space to find a small number of attractors for each of these simulated processors and to store them.

We think that it is possible to use the advanced Parallel Pollard-$\lambda$ variant to find $n$ discrete logarithms in a setting with one generator and a bounded interval. We expect that the average runtime per discrete logarithm can be reduced to approximately $\frac{1.76}{m}\sqrt{\frac{c}{n}}$ multiplications. However, even for $n = 1$, verification of this runtime has still to be done.

In general, it is even more interesting to develop algorithms that find $n$ bounded discrete logarithm vectors in $O(\sqrt{\frac{c}{n}})$ steps.

Finally, we have two completely different ideas for algorithms that find bounded discrete logarithm *vectors* in $\Theta(\sqrt{c})$ steps. They are followed by a spin off.

**A $\psi$ approach** Suppose that we have to solve $h = \mathbf{g}^{\mathbf{v}},\ v_i \in [0, c_i),\ i = 1, 2, ..., d$. We can make sequences do pseudo random walks by a predefined stepping rule in more than one coordinate. Define the steps of a *pseudo random walk* such that $\mathcal{S}(..) \sim N(0, \sigma_i^2),\ i = 1, 2, ..., d$. Now after $t$ steps, the sequence is stepping around 0 by the probability distribution $N(0, t\sigma_i^2),\ i = 1, 2, ..., d$. If we choose the right $\sigma_i,\ i = 1, 2, ..., d$ after $t = \sqrt{c}$ steps we can force a *density* of $\Theta(\frac{\sqrt{c}}{\prod_{i=1}^{d} c_i}) = \Theta(\frac{1}{\sqrt{c}})$ steps over the set $[0, c_1) \times [0, c_2) \times ... \times [0, c_d)$. As a result, a known sequence $\{x\}_t$ with evaluations $\mathbf{g}^{\mathbf{x}_t}$ and an unknown sequence $\{y\}_t$ with evaluations $h\mathbf{g}^{\mathbf{y}_t}$ will join with constant probability in $\Theta(\sqrt{c})$ steps.

This algorithm is called $\psi$[2], and it works fine for $d = 1$ with a constant number of $u \geq 5$ known and unknown sequences starting equidistantly in $[0, c)$. If we only count the number of steps, then it is faster than Pollard-$\lambda$. However, until now we have not tested ways to do each step by only one multiplication. At this moment we either need $\Theta(|c|)$ multiplications per step or, if we want one multiplication per step, we need $\Theta(c^{3/4})$ precomputed steps. By using so called *multiplication windows*, it is possible to do only a constant number of $z$ multiplications per step if we store $\Theta(c^{3/(4z)})$ precomputed steps. Another problem is, that for $d > 2$ we need too small $\sigma_i$ which easily leads to cycles, and this is *unwanted* in $\psi$. Therefore, we think that $\psi$ works differently (or maybe not at all) for $d \geq 3$. An advantage of $\psi$ is, that it can be used locally for using a priori knowledge. It can also be hybridized in a combination with Baby-step Giant-step and it can easily be parallelized.

**A $\beta$ approach** Suppose that we have to solve $h = \mathbf{g}^{\mathbf{v}},\ v_i \in [0, c_i),\ i = 1, 2, ..., d$. Define two sets of vectors $V = \{\mathbf{v} \mid v_i \in [0, \frac{4}{3}c_i),\ i = 1, 2, ...d\}$ and $W = \{\mathbf{w} \mid w_i \in [\frac{1}{3}c_i, \frac{5}{3}c_i),\ i = 1, 2, ...d\}$. A sequence is started at a random vector $\mathbf{v} \in V$ or in $\mathbf{w} \in W$. Now we evaluate either $h\mathbf{g}^{\mathbf{v}}$ or $\mathbf{g}^{\mathbf{w}}$ by the pseudo random decision rule $f : G_q \to \{V,\ W\} \times \{0,\ 1,\ 2, ..., \lceil c(\frac{4}{3})^d \rceil - 1\}$. Given integer $n = \lceil 2c(\frac{4}{3})^d \rceil$, we can use the modulo $n$ truncated elements $N[h\mathbf{g}^{\mathbf{v}}]_n$ or $N[\mathbf{g}^{\mathbf{w}}]_n$ for this rule. By $f(.)$, the next set and vector for the sequence only depend on the actual set and vector. Thus we have a sequence stepping in $V$ and $W$ in a pseudo random order, hence the name $\beta$.

At the time this sequence starts to cycle, there is a constant probability of $\Theta((\frac{3}{4})^d)$ that the starting vector of a cycle, $\mathbf{a}_c$ say, has two *different* predecessors $\mathbf{v} \in V$ and $\mathbf{w} \in W$ such that $N[h\mathbf{g}^{\mathbf{v}}]_n = N[\mathbf{g}^{\mathbf{w}}]_n$ both lead[3] to $\mathbf{a}_c$. Now with constant probability, also $h\mathbf{g}^{\mathbf{v}} = \mathbf{g}^{\mathbf{w}}$.

Like Pollard-$\rho$, it can easily be parallelized.

**Collision check per multiplication** Considering the above ideas, where each step takes $O(|c|)$ multiplications, we think that it is interesting to find out what the effect is if we test for a collision

[2]When turned upside down, $\psi$ symbolizes a starting position and a probability density around it.

[3]We have that $f(h\mathbf{g}^{\mathbf{v}}) = f(\mathbf{g}^{\mathbf{w}})$, such that the same set and vector follows.

after each multiplication instead of only after a complete step. We think that splitting up a step in both $O(|c|)$ multiplications and collision verifications, is a factor $O(\sqrt{|c|})$ faster.

# Appendix A

# Java Package

## A.1 Discrete Logarithm Settings

To implement and test discrete logarithm vector algorithms on a computer, we use large integers as group elements modulo prime $p$. There is a much faster non-generic algorithm called *index calculus* if we take integers for group element. This is explained for example in [3, §8.3.4]. However, for testing generic algorithms this is not an issue. For an implementation of $\mathbf{g}^{\mathbf{v}} \equiv_p h$, we first fix the order by prime $q$. Then we find $w$ such that $p = wq + 1$ is prime and we search for an integer $b$ such that $b^w \not\equiv 1 \bmod p$. Because $g^q \equiv_p b^{wq} \equiv_p b^{p-1} \equiv_p 1$, we have a generator $g = b^w$ which is of order $q$. Subsequently we make generators $g_i = g^{m_i},\ i = 1, 2, ..., d$, for random $m_i \in [0, q)$ such that we have random $g_i \in G_q$.

We may take for example $|p| = 1024$ and $|q| = 160$. In Java, $BigInteger$ $\boldsymbol{BigInteger}(l, t)$ returns[1] a random number of $l$ bits which is probably prime with an error rate of $2^{-t}$. We also have the method $boolean$ $\boldsymbol{isProbablyPrime}(t)$ that tests if a number is probably prime with an error rate of $2^{-t}$. Another method $BigInteger$ $\boldsymbol{BigInteger}(l, rnd)$ generates an integer of $l$ bits, randomly picked by pseudo random generator $rnd$.

The method $BigInteger$ $\boldsymbol{random}(|p|)$ returns a pseudo random integer value with bit $|p| - 1$ set, such that $random(|p|) \in [2^{|p|-1} - 1,\ 2^{|p|-2}]$. The method $BigInteger$ $\boldsymbol{clearBit}(i)$ clears bit number $i$. Since it is not so easy to get variables from a uniform distribution over integers $\{0, 1, 2, ..., p-1\}$, in line 5 and 7 we chose to approximate a uniform distribution by first taking a very large random number $N$ with $|N| = 2|p|$ bits. In line 7, for $N = pn + r$, on average we have $n = \Theta(p)$ and rest $r \in [0, p)$ is almost uniformly random[2].

## A.2 Class Hierarchy

Arrows with dashed lines imply *calls*. For example Demonstration *calls* Algorithms. Note that the classes Evaluator and Setting are used in this package as if they are one class.

---

[1] For $\pi(p)$ the number of primes smaller than $p$, $\lim_{p\to\infty}\ \pi(p)\frac{\log(p)}{p} = 1$. So for large $p \approx 2^{|p|}$ it takes about $\log(p) = \Theta(|p|)$ tries to find a prime of $|p|$ bits.

[2] If we pick $k = 0,\ 1,\ ...,\ N = \Theta(p^2)$ successively, then $k$ runs $n$ times over the full interval $[0, p)$, while it only once stops before it reaches $p$, namely when $k = np + r$. Since $n = \Theta(p)$, only $\Theta(\frac{1}{p})$ of the values for $r$ come from an interval that is smaller than $[0, p)$. Hence only $\Theta(\frac{1}{p})$ of the draws is from a non-uniform distribution over $[0, p)$.

**Algorithm A.1**: Integer group setting for $d$ pseudo randomly picked generators

1. ***input*** $d$, $|p|$, $|q|$, $t$, $rnd$
2. $q :=$ *BigInteger*$(|q|, t)$
3. ***do*** $\{$ $w := random(|p|).divide(q).clearBit(0)$ ; $p := wq + 1$
4. $\}$ ***until*** $p.isProbablyPrime(t)$
5. ***do*** $\{$ $b :=$ *BigInteger*$(2|p|, rnd).mod(p)$ ; $g := b^w$
6. $\}$ ***while*** $g = 1$ *mod* $p$
7. ***for*** $(i = 1$ *to* $d)$ ***do*** $\{$ $r :=$ *BigInteger*$(2|q|, rnd).mod(q)$
8. $g_i := g^r$ $\}$
9. ***output*** $p$, $g_1, g_2, ..., g_d$

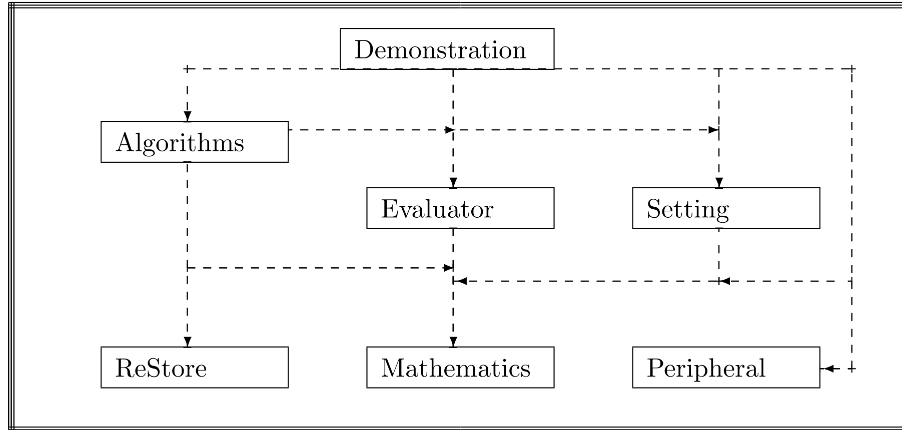


**Illustration A.2** : Class hierarchy of package DLV

## A.3 Class Description

For a good understanding of the classes and their methods, we start with a description of the most elementary classes. This bottom up description is continued until we arrive at the main class.

### A.3.1 Peripheral

Peripheral contains very basic methods to present data on screen. For example, we have the method *void* **out**(*int*[ ]), that prints a vector of integers on screen.

Peripheral is only used by the class **Demonstration**.

### A.3.2 Mathematics

Mathematics contains many methods that are necessary for our algorithms, but that are not available in Java. For example, the method *long* ***trunc***(*BigInteger* $b$, *int* $m$) returns $b$ *mod* $2^m$ as a long provided that $m \in \{0, 1, 2, ..., 63\}$. Also, Mathematics contains a method for ranking vectors *long* ***rank***(*long*[ ] $v$, *long*[ ] $c$) and for generation of vectors by their rank *long*[ ] ***vector***(*long* $r$, *long*[ ] $c$). Also simple statistics over a series of observations are available by a method *long*[ ] ***vector***(*long*[ ] $o$). The returned array contains average, standard deviation, maximum and minimum of the values in $o$. Surprisingly, Java does not support an integer square-root approximation of *long*, so we added the method *int* ***sqrt***(*long* $l$) to Mathematics. There are also a number of methods for bit manipulations in *long* that are used for fast vector evaluations.

Mathematics is used by the classes **Setting**, **Evaluator**, **Algorithms** and **Demonstration**.

### A.3.3 ReStore

ReStore implements a storage device with a method *void* ***store***(*BigInteger b*, *long i*) to index $i$ such that it can be retrieved by checking the indices that are returned after calling the method *long*[ ] ***similar***(*BigInteger b*).

ReStore can be used for time-space tradeoffs when hash tables need too much storage. At the cost of a very small number of losses of inputs, this alternative is even up to 20% more efficient than hash tables. An typical disadvantage is that the loss of storage sometimes causes a missed collision. In those cases, the algorithm has to restart (or to continue) until there is a collision with any of the stored data.

ReStore is only used by the class **Algorithms**.

### A.3.4 Setting

For a given number of coordinates $d$ and $|c|$ bits for the discrete logarithm vectors, this class sets generators $g_i$ and the coordinate bounds $[0, c_i)$ for $i = 1, 2, 3, ..., d$. For a mathematical background of the implementation, see Section A.1.

Setting is used by the classes **Algorithms** and **Demonstration**.

### A.3.5 Evaluator

Evaluator does precomputations. For example, the $g_i^{2^j}$ are stored for $i = 1, 2, ..., d$ and $j = 0, 1, 2, ...|c_i| - 1$, such that steps of size $2^j$ take only one multiplication. By Section 2.5.3, we have an average of 2 multiplications to evaluate a step of size $g_i^{2^j}$. Evaluator contains methods to evaluate a single bit change in one coordinate, by doing one multiplication in the multiplicative group. For example, we have *BigInteger* ***value***(*BigInteger e*, *long* $r_o$, *long*[ ] $r_n$) that returns a new element, given element $e$ with its rank $r_o$ and the new rank. This is particularly useful, if only one bit changes in one of the vector coordinates. As a counterpart, the method *BigInteger* ***invValue***(*BigInteger e*, *long* $r_o$, *long*[ ] $r_n$) evaluates inverse elements, by using the factors $g_i^{-2^j}$ for $i = 1, 2, ..., d$ and $j = 0, 1, 2, ...|c_i| - 1$.

Evaluator is used by the classes **Algorithms** and **Demonstration**.

### A.3.6 Algorithms

This class contains not only the algorithms in the form of methods that are each described and tested in the thesis. For example, there is also a $\psi$ algorithm based as described in Section 7.2, and a hybrid of Baby-step Giant-step and bruteForce. For running these algorithms, Setting and Evaluator have to be initialized.

Some algorithms only run for $d = 1$, but many algorithms are hybrid such that they can handle $d \geq 1$. All algorithms are called with input parameter $h$ and they return scalar $v$ or vector $\mathbf{v}$ such that $h = g^v$ or $h = \mathbf{g}^{\mathbf{v}}$.

- *long*[ ] ***bruteForce***(*BigInteger h*)
- *long*[ ] ***bruteStore***(*BigInteger h*)
- *long* ***bsGs***(*BigInteger h*)
- *long*[ ] ***extendedBsGs***(*BigInteger h*)
- *long*[ ] ***bsGsBruteForce***(*BigInteger h*)
  This hybrid algorithm precedes ***bGLambda***. It does Brute Force in one coordinate, such that it is $\Theta(\sqrt{\frac{c}{c_d} c_d}) = \Theta(\sqrt{c_d c})$ multiplications.

- *long*[ ] ***bGLambda***(*BigInteger h*)
  This hybrid algorithm combines Baby-step Giant-step with Pollard-$\lambda$. It takes only $\Theta(\sqrt{c})$ multiplications.

- *long* ***ConcurrentLambda****(BigInteger h, int u, int stepSizeOffset)*
  Input parameter $u$ is the average number of checkpoints until collision. The *stepSizeOffset* changes the maximum step size. For example, if normally the maximum step size is $2^{s-1}$, it becomes $2^{s-1+stepSizeOffset}$.

- *long* ***ParallelLambda***(*BigInteger h*, *int u*, *int m*)
  The fastest parallel algorithm for Pollard-$\lambda$, based on our new parallelization technique. Input parameter $u$ is the average number of checkpoints per processor until collision. Parameter $m$ tells the number known and unknown processors that are simulated.

- *long* ***lambdaGuaranteed***(*BigInteger h*, *int checkpoints*)
  This algorithm first finds *checkpoints* attractors as described in Section 5.1. After that, it runs an unknown Pollard-$\lambda$-sequence to find the discrete logarithm.

- *long* ***singlePsi***(*BigInteger h*)
  This algorithm runs a $\psi$ algorithm for $d = 1$. It takes $\Theta(\sqrt{c})$ steps.

Algorithms is only used by the class **Demonstration**.

### A.3.7 Demonstration

This is the main class. It first initializes the classes Setting and Evaluator by defining the number of coordinates $d$ and the number of bits $|c|$ that are needed to define a bounded discrete logarithm vector $\mathbf{v}$ with rank $n(\mathbf{v}) \in [0, c)$. It contains methods for demonstrating different algorithms in groups. For example *void* ***bruteAlgorithms***, *void* ***singleAlgorithms*** and *void* ***vectorAlgorithms***. For the results in this thesis, most measurements are done in a loop by means of repetitive testing. The method *void* ***loopingAlgorithms*** obtains results and returns average test results.

# Appendix B

# Description and Validation of Experiments

## B.1 Introduction

In this appendix we explain, validate and discuss the experiments that we use in the thesis for verification of what -mostly- are heuristics. First some general remarks that hold for all experiments.

- In our Java package, we use *long integers* for coordinates and vector ranks, such that $c \in [0, 2^{64})$. This is no limitation to us, since the computer configuration[1] we used is too slow for cases where $c > 2^{40}$.
- We test algorithms by running them many times for finding different *random* bounded discrete logarithm (vectors). As a result we obtain an indication of how the algorithm behaves, but the randomness in the bounded discrete logarithm (vector) causes variation in the average runtime that is measured.
- For testing algorithms with fixed step sizes, we use the amortized coordinate increment of Section 2.5.3. For Pollard-$\lambda$, we have the average binary coordinate addition of Section 2.5.3. Table B.1 compares the average number of bit changes that cause multiplications, to the number of steps done. Since every change is on average 2 multiplications, we consider the optimum number of multiplications half the number of multiplications that have been counted during runtime.

| **# steps $\approx 2\sqrt{c}$** | **Stable decimal # bit changes per step in 50 runs** |
|---|---|
| $2^6$ | 2 |
| $2^7$ | 2 |
| $2^8$ | 2.0 |
| $2^9$ | 2.0 |
| $2^{11}$ | 2.0 |
| $2^{13}$ | 2.00 |
| $2^{15}$ | 2.00 |
| $2^{17}$ | 2.000 |
| $2^{19}$ | 2.0000 |
| $2^{21}$ | 2.0000 |

**Table B.1**: Average # multiplications per step in Pollard-$\lambda$

[1] We implemented our package in Java, on a pentium IV microprocessor with a 1.6 GHz internal clock and 1 Gb RAM.

- We verified that the experiments yielded the same results for different $|p|$, with constant $|q| = 160$. See table B.2 . Theoretically we would have $2\sqrt{c} = 8192$ multiplications for Concurrent Pollard-$\lambda$ as described in Section 3.5.3, if exactly $\mu = \sqrt{c}$. For acceptably fast calculations, we decided to do the rest of the experiments with $|p| = 320$, $|q| = 160$ bits multiplicative elements in $G_q$.

| $\|p\|$ | **average # steps after 50 runs** |
|---|---|
| 200 | 8732 |
| 320 | 7420 |
| 512 | 8864 |
| 640 | 7485 |
| 1024 | 9265 |
| 1280 | 9032 |

**Table B.2**: Concurrent Pollard-$\lambda$ for $|p|$ if $|q| = 160$, $c = 2^{24}$

## B.2 Pollard-$\lambda$

To examine Pollard-$\lambda$, there are three important aspects. The first is the choice of the optimum average step size. As we restricted our setting to a small set of step sizes $2^i$, $i = 0, 1, 2, ..., s-1$, we have an average step size $\mu = \frac{1}{s}\sum_{i=0}^{s-1} 2^i$ that usually can't be exactly equal to $\sqrt{c}$. Table B.3 shows the optimum step sizes for different values of parameter $c$, after we do 50 runs with Concurrent Pollard-$\lambda$. It appears, that the approximation $s = \frac{|c|}{2} + \|c\| - 1$ is the best number of steps for this set of step sizes.

| **maximum step size** $2^{s-1}$ | **average # steps in 50 runs,** $\|c\| = 20,\ 22,\ 24,\ 26,\ 28,\ 30$ |
|---|---|
| $s = \frac{\|c\|}{2} + \\|c\\| - 1$ | 2113, 3248, 6743, 14703, 32475, 59121 |
| $s-1$ | 2232, 5228, 10518, 21821, 65419, 86332 |
| $s-2$ | 3577, 5675, 18105, 31864, 66285, 141908 |
| $s+1$ | 2112, 3871, 17049, 20811, 55225, 79352 |
| $s+2$ | 3578, 7873, 18236, 18344, 56537, 168568 |

**Table B.3**: Performance of Concurrent Pollard-$\lambda$ for different $s$

In table B.4, we show the number of steps for Concurrent Pollard-$\lambda$ from Section 3.5.3, if we have different values for parameter $c$ and $u = 100$ checkpoints per $\sqrt{c}$ steps. We can see that it is impossible to stay always close to the theoretical optimum runtime when only using step sizes $2^i$, $i = 0, 1, 2, ..., s-1$. We see that the average number of multiplications is close to the theoretical $2\sqrt{c}$.

In table B.5 we show that the difference from the minimum runtime drops on average by a factor $\frac{1}{u}$ if we leave checkpoints every $\frac{1}{u}\sqrt{c}$ steps.

| $c$ | $\frac{average\ \#\ steps}{\sqrt{c}}$ **after** 50 **runs per** $c$ |
|---|---|
| $\|c\| = 12$ | 1.92 |
| $\|c\| = 14$ | 1.98 |
| $\|c\| = 16$ | 2.93 |
| $\|c\| = 18$ | 1.97 |
| $\|c\| = 20$ | 2.55 |
| $\|c\| = 22$ | 1.80 |
| $\|c\| = 24$ | 1.83 |
| $\|c\| = 26$ | 2.15 |
| $\|c\| = 28$ | 1.77 |
| $\|c\| = 30$ | 2.01 |
| $\|c\| = 32$ | 1.78 |

**Table B.4**: Example of Pollard-$\lambda$ for different $c$

| **# checkpoints per** $\sqrt{c}$ **steps** | **average # steps after** 50 **runs for** $\|c\| = 20,\ 24,\ 28$ |
|---|---|
| 1 | 4308, 14260, 66854 |
| 2 | 3572, 10902, 47193 |
| 4 | 3034, 9473, 40480 |
| 8 | 2845, 8579, 37129 |
| 16 | 2651, 8051, 35426 |
| 32 | 2619, 7721, 34557 |
| 64 | 2584, 7646, 33915 |
| 256 | 2560, 7551, 33622 |
| 1024 | 2554, 7533, 33519 |
| 4096 | 2552, 7528, 33492 |

**Table B.5**: Concurrent Pollard-$\lambda$ for different numbers of checkpoints

## B.3 ReStore

For ReStore, we only tested the basic version, with $r_p = r_h = \sqrt{r}$, for $r$ storage retries. We used different $r$ and load factors $\lambda$, and we took truncated multiplicative elements from the cyclic group $G_q$, with $|p| = 2|q| = 320$ bits. For every setting, we stored $2^{17}$ truncated multiplicative elements, counted the number that could not be stored after $r$ tries. We put the average loss over 10 runs per setting in table B.6, showing the loss rate for different settings.

| **Setting for** $\lambda$ **and** $r_{max}$ | **average loss in** % |
|---|---|
| $\lambda = 0.75,\ r_{max} = 256$ | 0.0000 |
| $\lambda = 0.8,\ r_{max} = 64$ | 0.004 |
| $\lambda = 0.8,\ r_{max} = 16$ | 0.7 |
| $\lambda = 0.8,\ r_{max} = 4$ | 8.9 |
| $\lambda = 1.0,\ r_{max} = 64$ | 2.1 |
| $\lambda = 1.0,\ r_{max} = 16$ | 5.9 |
| $\lambda = 1.0,\ r_{max} = 4$ | 15.9 |
| $\lambda = 1.25,\ r_{max} = 64$ | 20 |
| $\lambda = 1.25,\ r_{max} = 16$ | 20.2 |
| $\lambda = 1.25,\ r_{max} = 4$ | 25.8 |

**Table B.6**: Efficiency of ReStore for $m = 2^{17}$ entries

## B.4 Pollard-$\lambda$ Attractors

to find attractor sets we need more steps than doing a single Pollard-$\lambda$ algorithm for the same $c$. In table B.7 we show averages for finding 20 attractor sets per $c$.

| **interval length** $c$ | $\frac{\textit{average \# steps}}{\sqrt{c}\lvert c\rvert}$ |
|---|---|
| $2^{12}$ | 0.34 |
| $2^{14}$ | 0.38 |
| $2^{16}$ | 0.60 |
| $2^{18}$ | 0.53 |
| $2^{20}$ | 0.47 |
| $2^{22}$ | 0.47 |
| $2^{24}$ | 0.45 |
| $2^{26}$ | 0.42 |
| $2^{28}$ | 0.39 |
| $2^{30}$ | 0.39 |
| $2^{32}$ | 0.67 |

**Table B.7**: Average performance for $|c|$ survivors over 20 runs per $c$

## B.5 Advanced Parallel Pollard-$\lambda$ (not confirmed)

*For unknown reasons, the test results that we show here could not be confirmed by tests with the newest version of our package. Therefore, we can't guarantee that these results reflect the relative runtime of Advanced Parallel Pollard-$\lambda$ when compared to the Parallel Pollard-$\lambda$ of [5, §5].*

For starting the advanced parallel algorithm, we need at most $m + 2|c|$ multiplications for finding equidistant starting positions for the $\frac{m}{2}$ known and the $\frac{m}{2}$ unknown sequences. Since in practice $\sqrt{c} \gg m$, these starting multiplications are insignificant but we counted them for determination of the performance. As a result, for small $c$, we expect that the number of processors influence the performance. Therefore, we decided to only use $m = 4, 6, 8, ..., 102$ for every value of $c$. After that we take the total number of multiplications for Parallel Pollard-$\lambda$.

With the same setting and for the same discrete logarithms, we also run Concurrent Pollard-$\lambda$ and take the total number of multiplications. Finally, the relative performance is the number of multiplications for Parallel Pollard-$\lambda$, divided by the number of multiplications for Concurrent Pollard-$\lambda$.

Generally, the average step sizes differ from optimum $\sqrt{c}$ in the case of Concurrent Pollard-$\lambda$, and $\frac{1}{m}\sqrt{c}$ in the case of Parallel Pollard-$\lambda$. Therefore, for some values of $c$, there is a significant mismatch in relative performance, compared to the average. On average, we used a total of $w = 50$ checkpoints average per processor per $\frac{1}{m}\sqrt{c}$ steps for both Parallel Pollard-$\lambda$ and Concurrent Pollard-$\lambda$.

See Table B.9 for the results. Now the average relative runtime of Advanced Parallel Pollard-$\lambda$ compared to Concurrent Pollard-$\lambda$ is 0.894. From this relative runtime, we would expect that the number of multiplications for Parallel Pollard-$\lambda$ is approximately

$$0.894\frac{2}{m}\sqrt{c} \approx \frac{1.79}{m}\sqrt{c}, \quad (u \to \infty)$$

| **maximum** $c$ | **relative average runtime for** $m = 4, 6, ..., 102$ |
| --- | --- |
| $2^{20}$ | 0.85 |
| $2^{21}$ | 0.88 |
| $2^{22}$ | 0.88 |
| $2^{23}$ | 0.88 |
| $2^{24}$ | 0.87 |
| $2^{25}$ | 0.89 |
| $2^{26}$ | 0.85 |
| $2^{27}$ | 0.87 |
| $2^{28}$ | 0.98 |
| $2^{29}$ | 1.02 |
| $2^{30}$ | 0.91 |
| $2^{31}$ | 0.89 |
| $2^{32}$ | 0.80 |
| $2^{33}$ | 0.90 |
| $2^{33}$ | 0.89 |
| $2^{34}$ | 0.89 |
| $2^{35}$ | 0.95 |

**Table B.8**: Parallel Pollard-$\lambda$ compared to Concurrent Pollard-$\lambda$

# Bibliography


[1] R. Cramer, R. Gennaro and B. Schoenmakers. *A secure and optimally efficient multi-authority election scheme*. In *Advances in Cryptology - EUROCRYPT '87*, volume 304 of *Lecture Notes in Computer Science*, pages 127-141. Springer-Verlag, Berlin 1988.

[2] B. Schoenmakers, *Fast computation of Relaxed Discrete Logarithms (draft)*. Handed out in Eindhoven, fall 2003 (without date).

[3] H.C.A. van Tilborg. *Fundamentals of cryptology*. Springer-Verlag, December 1999. ISBN 0792386752.

[4] J.M.Pollard. *Monte Carlo methods for index computation (mod p)*. In *Mathematics of computation*, 32(143):918-924, 1978.

[5] Paul C. van Oorschot and Michael J. Wiener. *Parallel collision search with cryptanalytic applications*. In *Journal of Cryptology*, 12(1):1-28, 1999.

[6] V. Shoup. *Lower bounds for discrete logarithms and related problems*. In *Advances in cryptology - EUROCRYPT '97, Lectures Notes in computer Science*, volume 1233, pages 256–266. Springer-Verlag, Berlin 1997.

[7] D. Chaum, J.-H. Evertse and J. van de Graaf. *An improved protocol for demonstrating possession of discrete logarithms and some generalizations*. In *Advances in cryptology - Eurocrypt'87, Lectures Notes in computer Science*, volume 304, pages 127-341. Springer-Verlag, Berlin, 1988.

[8] S. Brands. *An efficient off-line electronic cash system based on the representation problem*. In *Report CS-R9323, Centrum voor Wiskunde en Informatica*, Amsterdam, March 1993.

[9] B. Schoenmakers. *Personal communication*, 2003.

[10] Menenzes, Van Oorschot and Vanstone, *Handbook of applied cryptography*, CRC Press, Fifth Edition, August 2001. ISBN: 0849385237.

[11] R. Sedgewick, T.G. Symansky and A.C. Tao, *The complexity of finding cycles in periodic functions*, SIAM Journal on Computing, volume 11, no. 2 (1982), pages 376–390.

[12] J.J. Quisquater and J.P. Delescaille, *How easy is collision search. New results and applications to DES*, Advances in Cryptology - Crypto '89 proceedings, Lecture Notes in Computer Science, volume 435, Springer-Verlag, Berlin, pages 408-413.

[13] D. Knuth, *The Art of Computer Programming*, Volume 3: Sorting and Searching, Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Section 6.4: Hashing, pages 513-558.

[14] Wikipedia, *Hash table*, December 2006

[15] D. Knuth, *The Art of Computer Programming* (Pre-fascicle 2A), Volume 7, *Section 7.2.1.1* (draft): *Generating all n-tuples*, Addison-Wesley, p.1–3.

[16] D. Knuth, *The Art of Computer Programming* (Pre-fascicle 2C), Volume 7, *Section 7.2.1.3* (draft): *Generating all combinations*, Addison-Wesley, p.20–23.

[17] J.A. Rice, *Mathematical Statistics and Data Analysis*, second edition: Duxbury Press, 1995. ISBN 0534209343. p.41.

[18] D. Howe, *Free On-line Dictionary of Computing (FOLDOC)*, founded in 1985. Reedited by Liao, 13 april 2006 for Wikipedia.

# Index

$N[..]_s$, 17

Attractor set, 34
Average step size, 23

Binary representation, 13
Binary representation shorthand, 13
Bit length, 13
Bounded discrete logarithm problem, 4
Bounded discrete logarithm vector problem, 4
breaking of logarithms, 4

Checkpoint, 9
clustering, 11
Collision, 8
Computer cluster, 41
constraints, 38

Discrete logarithm problem, 4
Distinguished Point, 9
Distributed computing, 41
Double hashing, 11

Equidistant Point, 9
Evaluation, 13

Generation, 7
Generic algorithm, 16

hash collision, 11
hash function, 10
hash key, 10
Hash pseudo randomness, 11
hash value, 10
hashing, 10

integer intervals, 4

Known sequence, 20

Linear probing, 11
Load factor, 11

null, 29

Open addressing, 11

Pseudo Collision, 28

Rank, 7
Relative discrete logarithm, 5
running idle, 9

Sequence, 20
Shared memory, 41
Signed binary representation, 13
Survivor, 34

traversal, 7

unbounded, 4
Unknown sequence, 20

Vector encoding, 4
visiting, 7

window, 45

# Acknowledgments

First of all, I would like to thank Berry Schoenmakers for supporting me during this final project. Although I have been very confused and stressed from time to time, Berry stayed patient with me. He also tried to keep me focussed on more practical goals, when I was spending too much time on new ideas. During this proces I learned to keep more focus on less ideas, and to verify ideas by tests before continuing with other ideas. Berry, thank you for helping me with all that.

Also, I would like to thank Jan Willem Knopper. His arsenal of different tea tastes and in particular the fruitful discussions that developed over tasting them, inspired me.

In many ways, the pastors of the ESK have been very important to me, and this is not only during the final project. In particular Elisabeth Fricker, Wim de Leeuw and Martin van Moorsel. They helped me to get focussed on my own life in the many difficult times that I had to go through. They taught me how important it is to reach inner peace and to believe in love as the seed of all creative power.

Special thanks go to Jan Draisma, who took a lot of time to review the thesis in different stages. His detailed analysis and advise helped me to write more clearly about the mathematical problems that I encountered.