



DISKRETE LOGARITHMEN
ÜBER PRIMEN RESTKLASSENGRUPPEN

KLAUS-GÜNTHER SCHMIDT

- Zulassungsarbeit -

Universität Passau
Fakultät für Informatik und Mathematik
Lehrstuhl für Mathematik mit Schwerpunkt Digitale Bildverarbeitung

eingereicht bei:

Prof. Dr. Tomas Sauer
24. Januar 2018

Inhaltsverzeichnis

1	Einleitung	1
2	Grundlagen	2
2.1	Modulo und Restklassen	2
2.2	Gruppen und ihre Ordnung	3
2.2.1	Zyklische Gruppen	3
2.2.2	Prime Restklassengruppen	5
2.2.3	Primitivwurzeln	8
2.3	Kryptologie	9
3	Das diskrete-Logarithmen-Problem	10
3.1	Der allgemeine Logarithmus	10
3.2	Der diskrete Logarithmus	11
3.3	Motivation für die Benutzung der diskreten Exponentiation	12
3.4	Der Diffie-Hellman-Schlüsselaustausch	17
3.4.1	Verfahren	17
3.4.2	Sicherheit	18
4	Algorithmen	19
4.1	Shanks Babystep-Giantstep-Algorithmus	19
4.1.1	Verfahren	19
4.1.2	Laufzeitkomplexität und Speicherbedarf	21
4.1.3	Zusammenfassung des Algorithmus	22
4.1.4	Beispiel	22
4.2	Pollard-Rho-Algorithmus	23
4.2.1	Verfahren	24
4.2.2	Laufzeitkomplexität und Speicherbedarf	27
4.2.3	Periodizität der Folge	28
4.2.4	Optimierung der Speicherkomplexität	29
4.2.5	Zusammenfassung des Algorithmus	30
4.2.6	Beispiel	31
4.3	Pohlig-Hellman-Algorithmus	34
4.3.1	Reduktion auf Gruppen mit Primzahlpotenzordnungen	35
4.3.2	Laufzeitkomplexität und Speicherbedarf	37
4.3.3	Reduktion auf Gruppen mit Primzahlzordnungen	38
4.3.4	Optimierte Komplexitätseigenschaften	41
4.3.5	Zusammenfassung des Algorithmus	42

4.3.6	Beispiel	43
5	Implementierung	45
5.1	Entwicklungsumgebung	45
5.1.1	Software	45
5.1.2	Hardware	45
5.2	Bericht zum Entwicklungsprozess	45
5.2.1	Shell	46
5.2.2	Zusätzliche Klassen	48
5.2.3	Algorithmen	48
5.3	Statistiken	51
5.4	Zusammenfassung	52
6	Analyse der Testdaten	54
6.1	Laufzeitanalyse	54
6.1.1	Längsschnittanalyse	54
6.1.2	Querschnittanalyse	56
6.2	Speicheranalyse	60
6.2.1	Management des JVM Heap	60
6.2.2	Speicherbedarf der Algorithmen	60
6.2.3	Zusammenfassung	64
7	Einbindung der Thematik in die Oberstufe des bayerischen Gymnasiums	65
8	Neue Standards für Quantenkryptografie	68
9	Literatur	V
10	Anhang: Schulmaterialien	VIII
11	Eigenständigkeitserklärung	XIII

Abbildungsverzeichnis

2.1	1 als erzeugendes Element	4
2.2	Beispielwerte der Eulerschen Phi-Funktion	7
2.3	2 und 3 als Erzeuger der Gruppe $(\mathbb{Z}/5\mathbb{Z})^*$	9
3.1	Graphen der Exponential- und Logarithmusfunktion zur Basis 2	10
3.2	Drei Funktionen zur O-Notation	13
3.3	Diffie-Hellman-Schlüsselaustausch nach [41, S. 355]	18
4.1	Ausgabe für das DLP $6^x \equiv 15 \pmod{41}$ mit Shanks BSGS Algorithmus .	23
4.2	Grafische Visualisierung der Rho-Periodizität	29
4.3	Ausgabe für das DLP $6^x \equiv 15 \pmod{41}$ mit dem Rho-Algorithmus	34
4.4	Der Ablauf des Pohlig-Hellman-Algorithmus	38
4.5	Ausgabe für das DLP $6^x \equiv 15 \pmod{41}$ mit dem Pohlig-Hellman-Algorithmus	44
5.1	Startbildschirm „Diskrete Logarithmen Berechner“	46
5.2	Überprüfung auf Parameterfehler	46
5.3	Zwei Varianten von Pollards Rho-Algorithmus	47
5.4	Benachrichtigung	47
5.5	Skript für automatisierte Berechnungen	48
5.6	Begrenzter Heapsize (Screenshot JProfiler)	49
5.7	Pollards Algorithmus findet keine Lösung für fehlerhaftes DLP	50
5.8	Optimierter Pohlig-Hellman-Algorithmus	51
5.9	statistische Daten - ausgewertet durch IntelliJ	52
6.1	Entwicklung der Laufzeit (in ms) bei ausgewählten Testfällen	55
6.2	Vergleich der Laufzeit (in ms) verschiedener Endsysteme bei ausgewählten Testfällen	57
6.3	Vergleich der Laufzeit (in ms) des obigen DLPs	58
6.4	Laufzeitvarianz (in ms) bei Pollards Rho-Algorithmus mit PC1	59
6.5	Laufzeitvarianz (in ms) bei Pollards Rho-Algorithmus mit PC4	59
6.6	#4mittel mit Pollard	60
6.8	Speicherbedarf für ein DLP mit Ordnung im Bereich 2^{200}	62
6.9	Speicherbedarf der vier mittleren DLPs mit Shanks Algorithmus	62
6.10	Speicherbedarf #2groß	63
6.11	Ausschnitt: Speicherbedarf #4groß	63

Tabellenverzeichnis

3.1	Diskrete Exponentiation zur Basis 3 in $(\mathbb{Z}/5\mathbb{Z})^*$	11
3.2	Diskreter Logarithmus zur Basis 3 in $(\mathbb{Z}/4\mathbb{Z})$	11
3.3	Die von 2 erzeugte Einheitengruppe (mod 13)	16
3.4	Die von 3 erzeugte Untergruppe (mod 13)	16
4.1	Berechnungen der Phase $q = 1$	32
4.2	Berechnungen der Phase $q = 2$	32
4.3	Berechnungen der Phase $q = 4$	32

1 Einleitung

Um digitale Kommunikation zwischen zwei oder mehreren Teilnehmern zu regeln, verwendet man Algorithmen. Sie legen u.a. fest, wie sich Teilnehmer in einem System authentifizieren oder wie gesendete Informationen ver- und entschlüsselt werden. Damit diese Kommunikation sicher und performant stattfindet, ist es sinnvoll, standardisierte bzw. empfohlene Algorithmen zu verwenden. Standardisierungsprozesse und Empfehlungen werden häufig von Behörden und Instituten vorgenommen, um sicherzustellen, dass ein Kryptosystem nicht angreifbar ist. Zu solchen Einrichtungen gehören z.B. das „Bundesamt für Sicherheit in der Informationstechnik“ (BSI) oder das „American National Standards Institute“ (ANSI). Eine Veröffentlichung des ANSI [2] informiert bspw. über Standards zur Vereinbarung von Schlüsseln in symmetrischen Kryptosystemen.

Das Management von Schlüsseln ist ein zentrales Thema der Kryptografie, da sie für die Sicherheit eines Verfahrens eine tragende Rolle spielen. Dies liegt an einem bedeutenden Grundsatz der Kryptografie, dem sogenannten „Kerckhoffs’schen Prinzip“. Es „besagt, dass die Sicherheit eines kryptografischen Verfahrens allein von den verwendeten Schlüsseln abhängen sollte“ [12, S. 296]. Das bedeutet, „dass ein Algorithmus nur dann als sicher gilt, wenn seine Bekanntmachung der Sicherheit der Kommunikation nicht schadet“ [36, S. 94]. Im Gegenzug erfordert dies aber für einen Schlüssel, dass er zum einen geheim gehalten werden muss und zum anderen bei der Kryptoanalyse durch Dritte (z.B. Hacker oder Geheimdienste) selbst mit effizientem Aufwand nicht herausgefunden werden darf.

In der oben erwähnten Veröffentlichung des ANSI zur Schlüsselvereinbarung wird als mathematische Grundlage auf die diskrete Exponentiation gesetzt. Mithilfe des diskreten Logarithmus, der zugehörigen Umkehrfunktion, könnte aus mitgelesenen Daten möglicherweise dennoch der Schlüssel herausgerechnet werden. Die Sicherheit von Kryptoverfahren, die diskrete Exponentiation verwenden, basiert somit auf der Frage, wie schwer es ist einen diskreten Logarithmus zu berechnen, womit sich die folgende Arbeit beschäftigt.

Dazu werden zunächst die für das Verständnis benötigten Grundlagen aus der Mathematik und Informatik erläutert. Im Anschluss wird die Motivation für die Nutzung der diskreten Exponentiation erklärt, wobei ich anhand des Diffie-Hellman-Schlüsselaustauschs eine Anwendung für die diskrete Exponentiation darlege. Im Kern der Arbeit schildere ich die Funktionsweise dreier Algorithmen, mit denen der diskrete Logarithmus berechnet werden kann. Dabei gehe ich auch jeweils auf deren Komplexität in Bezug auf Zeit und Speicherplatz ein. Danach stelle ich im Implementierungsbericht meine Umsetzung mittels Java vor und analysiere die mit dem Programm gewonnenen Testdaten. Abschließend erfolgt ein Vorschlag zur Einbindung des Themas in die Oberstufe des bayerischen Gymnasiums.

2 Grundlagen

Diskrete Logarithmen werden bei ihrer Anwendung in der Kryptologie häufig im Zusammenhang mit endlichen, zyklischen Gruppen verwendet. Im Folgenden sollen die Grundlagen hierzu geklärt werden, die der Darstellung von Buchmann [9, Kapitel 1, 2] folgen.

2.1 Modulo und Restklassen

Definition 1 (Kongruenzen)

Seien $a, b \in \mathbb{Z}$ und $m \in \mathbb{N}$. Zwei Zahlen sind genau dann zueinander kongruent, wenn sie bei einer Division mit dem gleichen Divisor m denselben (ganzzahligen) Rest lassen. Dies ist genau dann der Fall, wenn m die Differenz $b - a$ teilt. Wir schreiben hierfür:

$$a \equiv b \pmod{m}$$

Satz 2 (Modulo m als Äquivalenzrelation)

Auf \mathbb{Z} lässt sich mit der Kongruenz $(\text{mod } m)$ eine Äquivalenzrelation einführen:

1. Jede ganze Zahl ist bezüglich $(\text{mod } m)$ zu sich selbst kongruent. (Reflexivität)
2. Aus $a \equiv b \pmod{m}$ folgt, dass auch $b \equiv a \pmod{m}$ gilt. (Symmetrie)
3. Aus $a \equiv b \pmod{m}$ und $b \equiv c \pmod{m}$ folgt, dass auch $a \equiv c \pmod{m}$ gilt. (Transitivität)

Ein Beweis dazu findet sich unter [26, S. 1].

Wir erhalten die zu einer Äquivalenzklasse von a gehörenden Zahlen, indem man zu a ganzzahlige Vielfache von m addiert:

$$\{b : b \equiv a \pmod{m}\} = a + m\mathbb{Z}$$

Diese Menge wird auch als Restklasse von $a \pmod{m}$ bezeichnet. Betrachtet man die Menge aller Restklassen $(\text{mod } m)$ so schreibt man $\mathbb{Z}/m\mathbb{Z}$. Die Standardvertreter von $\mathbb{Z}/m\mathbb{Z}$ sind $\{0, 1, \dots, m-1\}$.

Definition 3 (Verknüpfungen auf Restklassen)

Auf $\mathbb{Z}/m\mathbb{Z}$ lassen sich Addition und Multiplikation als Verknüpfungen definieren. Dabei gilt:

$$\begin{aligned}(a + m\mathbb{Z}) + (b + m\mathbb{Z}) &= (a + b) + m\mathbb{Z} \\ (a + m\mathbb{Z}) \cdot (b + m\mathbb{Z}) &= (a \cdot b) + m\mathbb{Z}\end{aligned}$$

Diese Verknüpfungen sind somit kommutativ und assoziativ.

2.2 Gruppen und ihre Ordnung

Definition 4 (Gruppe)

Gegeben sei eine nicht-leere Menge M , auf der eine Verknüpfung $\circ : M \times M \rightarrow M$ mit $(a, b) \mapsto a \circ b$ existiert. Sind die folgenden Eigenschaften erfüllt, so nennen wir (M, \circ) eine Gruppe.

- Assoziativität:

$$\forall a, b, c \in M : a \circ (b \circ c) = (a \circ b) \circ c$$

- Existenz des neutralen Elements:

$$\exists \varepsilon \in M : \varepsilon \circ a = a = a \circ \varepsilon, \forall a \in M$$

- Existenz der inversen Elemente:

$$\forall a \in M : \exists a^{-1} \in M : a \circ a^{-1} = a^{-1} \circ a = \varepsilon$$

Definition 5 (Ordnung einer Menge)

Die Anzahl der Elemente einer beliebigen Menge M bezeichnen wir als ihre Ordnung. Sie wird mit der Funktion $\text{ord} : M \rightarrow \mathbb{N} \cup \{\infty\}$, $\text{ord}(M) \mapsto n$ angegeben.

Definition 6 (Ordnung einer Gruppe)

Sei M eine Menge, für die die Gruppenaxiome gelten. Dann ist M eine Gruppe. Für $\text{ord}(M)$ gilt:

Null befindet sich nicht im Bildbereich, da Gruppen mindestens ein Element - das Neutrale - enthalten müssen. Unendlich ist enthalten, da es auch unendliche Gruppen gibt. Diese sind für die hier benötigte Anwendung aber nicht relevant.

Eine Gruppe wird „endliche Gruppe“ genannt, wenn ihre Grundmenge M aus endlich vielen Elementen besteht, d.h. wenn $\text{ord}(M) = n$ mit $n \in \mathbb{N}$ gilt.

Definition 7 (Untergruppe)

Sei (M, \circ) eine Gruppe und sei $U \subseteq M$. Die Teilmenge U heißt Untergruppe bzgl. (M, \circ) , wenn sie mit der Verknüpfung \circ wiederum eine Gruppe bildet. Nach dem Satz von Lagrange gilt, dass die Ordnung einer Untergruppe ein Teiler der Gruppenordnung ist [15, vgl. S. 29].

2.2.1 Zyklische Gruppen

Definition 8 (Zyklische Gruppe)

Sei (M, \circ) eine endliche Gruppe. M ist genau dann zyklisch, wenn es ein Element $y \in M$ gibt, für das $\langle y \rangle = M$ gilt. Das bedeutet, dass alle Elemente der Menge M durch die wiederholte Verknüpfung eines Elements y mit sich selbst erzeugt werden können. y heißt dann „erzeugendes Element von M “ und M „die von y erzeugte Gruppe“ oder „Spann von y “.

Die k -fache Verknüpfung des Elements y mit sich selbst notieren wir mit $y^k = \underbrace{y \circ \dots \circ y}_{k\text{-mal}}$.

Nachdem M eine Gruppe ist und in jeder Gruppe auch das neutrale Element enthalten ist, existiert ein $x \in \mathbb{N}$, sodass $y^x = \varepsilon$ gilt. Wir bezeichnen das kleinste x , für das die Aussage $y^x = \varepsilon$ gilt, als Ordnung n des Elements y .

$$o(y) := n = \min(x \in \mathbb{N} : y^x = \varepsilon).$$

Da alle Gruppenelemente als Potenz des erzeugenden Elements geschrieben werden können, ergibt sich auch, dass jede zyklische Gruppe kommutativ ist [20, vgl. S. 156].

Zwar gibt es auch unendliche zyklische Gruppen. Jedoch wird in dieser Arbeit immer von endlichen zyklischen Gruppen ausgegangen.

Beispiel 9 (zyklische Untergruppe)

Sei (M, \circ) eine zyklische Gruppe. Für jedes $g \in M$ bildet die Menge $U = \{g^k : k \in \mathbb{Z}\} = \langle g \rangle$ die (von g erzeugte) Untergruppe bzgl. M .

Satz 10 (Die Ordnung des Erzeugers als Ordnung der erzeugten Gruppe)

Die Ordnung n des erzeugenden Elements y einer zyklischen Gruppe M entspricht der Ordnung von M :

$$o(y) = n = \text{ord}(M)$$

Beweis. Wähle beliebiges $n' = q \cdot n + r$ mit $0 \leq r < n$ und $r, q \in \mathbb{N}_0$:

Nun gilt:

$$\begin{aligned} y^{n'} &= y^{q \cdot n + r} \\ &= \underbrace{y^n \circ \dots \circ y^n}_{q\text{-mal}} \circ y^r \quad | y^n = \varepsilon \\ &= \varepsilon^q \circ y^r \\ &= y^r \end{aligned}$$

Man sieht, dass sich jeder Exponent $n' = q \cdot n + r$ auf $r \equiv n' \pmod{n}$ reduzieren lässt. Da r zwischen 0 und $n - 1$ liegt, können also genau n Elemente erzeugt werden.

n als kleinste Potenz von y , die das neutrale Element erzeugt, entspricht also auch der Anzahl der Elemente, die im Spann von y liegen. \square

Beispiel 11

Die zyklische Gruppe $(\mathbb{Z}/4\mathbb{Z}, +)$ besitzt das erzeugende Element 1 und das neutrale Element 0. Die Menge $\mathbb{Z}/4\mathbb{Z}$ beinhaltet die Elemente $\{0, 1, 2, 3\}$ (siehe Abbildung 2.1).

In der nebenstehenden Abbildung sieht man die k -fache Verknüpfung des erzeugenden Elements 1 mit sich selbst:

k	1	2	3	4	5	6	...
$1^k \bmod 4$	1	2	3	0	1	2	...

Abbildung 2.1: 1 als erzeugendes Element

Dort erkennt man auch, dass 4 der erste Exponent ist, bei dem das neutrale Element 0 erzeugt wird.

$$1^4 \equiv 1 + 1 + 1 + 1 \equiv 0 \pmod{4}$$

Alle Elemente, auf die durch nachfolgende Exponenten abgebildet wird, sind Wiederholungen bereits erzeugter Elemente. Daher kommt auch der Name „zyklische Gruppe“, denn mit einem Zyklus von vier Schritten wiederholen sich alle Elemente aus dem Spann des Erzeugers. Dies wird durch die blauen Pfeile in Abbildung 2.1 verdeutlicht.

2.2.2 Prime Restklassengruppen

Definition 12 (Prime Restklassen)

Sei $m \in \mathbb{N}$. Schränkt man die Menge aller Restklassen $\{0, 1, \dots, m-1\}$ auf diejenigen Vertreter ein, die relativ prim zu m sind, erhalten wir die Menge aller primen Restklassen. Relativ prim bedeutet, dass der größte gemeinsame Teiler (ggT) der jeweiligen Restklasse und m gleich 1 ist.

$$(\mathbb{Z}/m\mathbb{Z})^* = \{a \in \mathbb{Z}/m\mathbb{Z} : \text{ggT}(a, m) = 1\}$$

Beispiel 13

Sei $m = 6$. Die Menge der Restklassen ist $\mathbb{Z}/6\mathbb{Z} = \{0, 1, 2, 3, 4, 5\}$. Die Elemente 2, 3, 4 und 0 haben als ggT zu 6 einen nichttrivialen Teiler. Deswegen sind die primen Restklassen $(\mathbb{Z}/6\mathbb{Z})^*$ die Restklassen zu den Vertretern $\{1, 5\}$.

Beispiel 14

Sei $m = 7$. Die Menge der Restklassen ist $\mathbb{Z}/7\mathbb{Z} = \{0, 1, 2, 3, 4, 5, 6\}$. Da die Zahl 7 eine Primzahl ist, hat sie nur mit der Zahl 0, die durch jede Zahl teilbar ist, den ggT 7. Alle anderen Restklassen besitzen mit 7 den ggT 1. Deswegen sind die primen Restklassen bezüglich einer Primzahl als Modulwert stets alle Restklassen bis auf die 0: $(\mathbb{Z}/7\mathbb{Z})^* = \{1, 2, 3, 4, 5, 6\}$.

Satz 15 (Prime Restklassengruppen)

Die Menge der primen Restklassen bezüglich m bildet mit der Multiplikation eine endliche Gruppe $((\mathbb{Z}/m\mathbb{Z})^, \cdot)$ und wird prime Restklassengruppe genannt.*

Beweis.

- Assoziativität: Diese Eigenschaft wird über die Assoziativität von $\mathbb{Z}/m\mathbb{Z}$ vererbt.
- Neutrales Element: 1 ist das neutrale Element zur Multiplikation. Da für beliebige Elemente $m \in \mathbb{N}$ immer $\text{ggT}(1, m) = 1$ gilt, ist 1 stets in der Menge der primen Restklassen enthalten. Daraus folgt auch, dass die Menge der primen Restklassen niemals leer ist.

- Invertierbarkeit: Das Lemma von Bézout [20, vgl. S. 58] besagt:

$$\forall a, b \in \mathbb{Z} \exists x, y \in \mathbb{Z} : \text{ggT}(a, b) = ax + by$$

In den primen Restklassen sind nur die Elemente r enthalten, die mit m den ggT 1 besitzen. Unter Verwendung des Lemmas von Bézout folgt, dass es $x, y \in \mathbb{Z}$ geben muss, die $1 = \text{ggT}(r, m) = rx + my$ erfüllen. Es gilt $1 \equiv rx + my \equiv rx \pmod{m}$. Das bedeutet, dass in \mathbb{Z} (bzw. in den zugehörigen Vertretern in $\mathbb{Z}/m\mathbb{Z}$) ein Inverses für r existiert. Somit folgt, dass alle primen Restklassen bezüglich m invertierbar sind.

- **Abgeschlossenheit:** Seien $a, b \in (\mathbb{Z}/m\mathbb{Z})^*$. Das bedeutet, dass weder a noch b einen gemeinsamen Faktor ($\neq 1$) mit m besitzen. Auch das Produkt $a \cdot b$ besitzt keinen gemeinsamen Teiler ($\neq 1$) mit m , da es genau das Produkt der Faktoren von a und b ist. Somit ist die Verknüpfung abgeschlossen.

□

Bemerkung 16 (Einheitengruppe)

Elemente, die ein Inverses (bezüglich einer Struktur) besitzen, werden auch Einheiten genannt. Wie oben gezeigt, enthält eine prime Restklassengruppe genau die Restklassen bezüglich eines Moduls, die invertierbar sind. Aus diesem Grund wird sie auch Einheitengruppe genannt.

Bemerkung 17 (Bezug zu diskreten Logarithmen)

Wie später beschrieben (siehe Kapitel 4), werden in dieser Arbeit diskrete Logarithmen über primen Restklassengruppen berechnet. Das bedeutet, dass die Urbildmenge bei der Bestimmung eines diskreten Logarithmus immer eine prime Restklassengruppe ist. Wie aufwendig dies ist, hängt stark von der Größe der primen Restklassengruppe ab. Statt die konkreten Elemente zu kennen, ist also vielmehr interessant, welche Ordnung eine prime Restklassengruppe hat. Dazu kann die Eulersche Phi-Funktion genutzt werden.

Bemerkung 18 (Fundamentalsatz der Arithmetik)

Die Menge der Primzahlen wird folgend mit \mathbb{P} beschrieben.

Nach dem Fundamentalsatz der Arithmetik gilt: „Jede natürliche Zahl $m > 1$ kann als Produkt von Primzahlen geschrieben werden. Bis auf die Reihenfolge sind die Faktoren in diesem Produkt eindeutig bestimmt“ [9, S. 11].

Davon ausgehend kann man eine Zahl $m \in \mathbb{N}$ in ihre Primfaktoren zerlegen. Für jeden Primfaktor $p \in \mathbb{P}$, der m teilt, geben wir den zugehörigen Exponenten mit $e(p) \in \mathbb{N}$ an [9, vgl. S. 11]. Somit lautet die (bis auf Reihenfolge) eindeutige Darstellung von m in Primfaktoren:

$$m = \prod_{p \in \mathbb{P}, p|m} p^{e(p)}$$

Definition 19 (Die Eulersche Phi-Funktion)

Mit der Eulerschen Phi-Funktion ist es möglich, ausgehend vom Modulwert m die

Anzahl der zugehörigen primen Restklassen zu berechnen.
Mithilfe der eindeutigen Primfaktorzerlegung von m (siehe Bemerkung 18) können wir die Eulersche Phi-Funktion wie folgt definieren [14, vgl. S. 29]:

$$\begin{aligned}\varphi : \mathbb{N} &\rightarrow \mathbb{N}, m \mapsto \varphi(m) \\ \varphi(m) &= \prod_{p \in \mathbb{P}, p|m} (p^{e(p)} - p^{e(p)-1}) \\ &= m \cdot \prod_{p \in \mathbb{P}, p|m} \left(1 - \frac{1}{p}\right)\end{aligned}$$

Im Sonderfall $m \in \mathbb{P}$ folgt: $\varphi(m) = m - 1$

Da die Eulersche Phi-Funktion die Anzahl der primen Restklassen zum Modulwert m bestimmt, gibt sie somit auch die Ordnung der primen Restklassengruppe bezüglich m an. $\varphi(m) = \text{ord}((\mathbb{Z}/m\mathbb{Z})^*)$.

Beispielhaft finden sich in der nächsten Abbildung die ersten 50 Werte der Eulerschen Phi-Funktion.

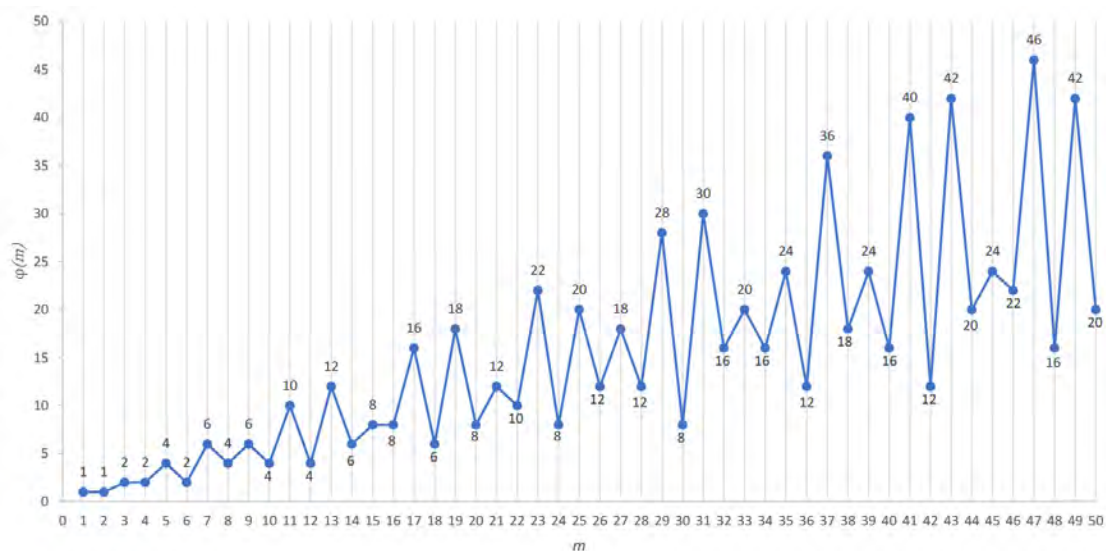


Abbildung 2.2: Beispielwerte der Eulerschen Phi-Funktion

Beispiel 20

Mit der Eulerschen Phi-Funktion kann nun überprüft werden, ob in den vorangegangenen beiden Beispielen (13, 14) die Anzahl der gefundenen primitiven Restklassen korrekt ist.

- zu Beispiel 13: $\varphi(6) = 6 \cdot \frac{1}{2} \cdot \frac{2}{3} = 2 = |\{1, 5\}|$
- zu Beispiel 14: $\varphi(7) = 7 - 1 = 6 = |\{1, 2, 3, 4, 5, 6\}|$

Nun folgt mit dem „Satz von Euler“ ein bedeutender Satz der Zahlentheorie, der auch für die Algorithmen in Kapitel 4 benötigt wird:

Satz 21 (Satz von Euler)

Sei $1 < m \in \mathbb{N}$ und sei $a \in \mathbb{N}$ relativ prim zu m , also $\text{ggT}(a, m) = 1$. Dann gilt

$$a^{\varphi(m)} \equiv 1 \pmod{m}$$

Beweis. Sei $G = ((\mathbb{Z}/m\mathbb{Z})^*, \cdot)$ eine prime Restklassengruppe. Da a relativ prim zu m ist, gilt $a \pmod{m} \in G$. a ist somit auch Erzeuger einer Untergruppe von G (siehe Beispiel 9). Es ist nach Satz 19 bekannt, dass $\varphi(m)$ die Ordnung der primen Restklassengruppe $((\mathbb{Z}/m\mathbb{Z})^*, \cdot)$ angibt. Außerdem wissen wir (siehe Definition 7), dass die Ordnung einer Untergruppe ein Teiler der Gruppenordnung ist. Damit gilt, dass die Ordnung von a , die Gruppenordnung von G , $\text{ord}(G) = \varphi(m)$, teilt. Das bedeutet $\exists k \in \mathbb{N} : o(a) \cdot k = \text{ord}(G)$. Nun lässt sich schreiben:

$$a^{\varphi(m)} \equiv a^{o(a) \cdot k} \equiv (a^{o(a)})^k \equiv 1^k \equiv 1 \pmod{m}$$

□

2.2.3 Primitivwurzeln**Definition 22** (Primitivwurzeln)

Damit eine prime Restklassengruppe $(\mathbb{Z}/m\mathbb{Z})^*$ zyklisch ist, muss nach Definition 8 ein erzeugendes Element existieren. Ein solches Element, für das $\langle y \rangle = (\mathbb{Z}/m\mathbb{Z})^*$ gilt, wird Primitivwurzel genannt.

Bemerkung 23 (Existenz von Primitivwurzeln)

Nicht alle primen Restklassengruppen sind zyklisch. Nach Burton [10, vgl. S. 162] existiert eine Primitivwurzel genau dann, wenn $m \in \{2, 4, p^k, 2p^k\}$, $p \in \mathbb{P} \setminus \{2\}$ und $k \in \mathbb{N}$ ist. Sofern primitive Wurzeln zu einem Modulwert m existieren, gibt es nach [10, vgl. S. 151] genau $\varphi(\varphi(m))$ viele. Durch die Existenz mindestens einer Primitivwurzel ist die zugehörige prime Restklassengruppe $(\mathbb{Z}/m\mathbb{Z})^*$ somit zyklisch.

Bemerkung 24 (Finden von Primitivwurzeln)

Für die Suche nach Primitivwurzeln ist die Kenntnis der Primfaktorzerlegung der Ordnung der Einheitengruppe von Vorteil:

Da nach dem Satz von Euler (21) für alle Elemente aus der Einheitengruppe $a^{\varphi(m)} = 1$ gilt, kann damit nicht bestimmt werden, ob a eine Primitivwurzel ist. Das Problem ist, dass es auch Untergruppen existieren können. Nach Definition 7 ist bekannt, dass die Ordnung einer Untergruppe immer ein Teiler der Gruppenordnung ist.

Deswegen werden bei der Suche nach Primitivwurzeln Schritt für Schritt alle Erzeuger von echten Untergruppen ausgeschlossen, sodass nur noch die Erzeuger der Einheitengruppe und somit die Primitivwurzeln übrig bleiben.

Dafür bestimmen wir die Menge T aller nichttrivialen Teiler von $\varphi(m)$, also ohne 1 oder $\varphi(m)$. Sofern für eine prime Restklasse $a \in (\mathbb{Z}/m\mathbb{Z})^*$ und einen Teiler $t \in T$ gilt, dass $a^t \equiv 1 \pmod{m}$ ist, hat a als Erzeuger die Ordnung t . Da t aber echt kleiner ist als $\varphi(m)$, kann a nicht die gesamte Menge aller primen Restklassen erzeugen. a fällt somit als Primitivwurzel heraus.

Zusammengefasst sind primitive Wurzeln also genau die primen Restklassen, die ausschließlich mit dem Exponent $\varphi(m)$ äquivalent zu 1 sind.

Beispiel 25

Sei der Modulwert $m = 5$. Da 5 eine Primzahl ist, gilt $\varphi(5) = 4 = \text{ord}((\mathbb{Z}/5\mathbb{Z})^*)$ und $(\mathbb{Z}/5\mathbb{Z})^* = \{1, 2, 3, 4\}$. Um die erzeugenden Elemente der Einheitengruppe zu finden, betrachten wir alle echten Teiler von $\varphi(5) = 4$. Dies ist lediglich die Zahl 2. Anschließend wird überprüft, für welche der Elemente $a \in (\mathbb{Z}/5\mathbb{Z})^*$ die Äquivalenz $a^2 \equiv 1 \pmod{5}$ gilt. Wir finden die Elemente 1 und 4 (in Abbildung 2.3 rot), die nun gestrichen werden können, da sie die Ordnung 2 haben. Als Primitivwurzeln bleiben somit die primen Restklassen 2 und 3 (grün) übrig.

Ob die Anzahl der gefundenen Primitivwurzeln stimmt, kann kontrolliert werden, indem wir $\varphi(\varphi(5)) = \varphi(4) = 2$ berechnen.

$a \in (\mathbb{Z}/5\mathbb{Z})^*$	1	2	3	4
$a^2 \pmod{5}$	1	4	4	1

Abbildung 2.3: 2 und 3 als Erzeuger der Gruppe $(\mathbb{Z}/5\mathbb{Z})^*$

2.3 Kryptologie

In Bezug auf geheime Kommunikation gibt es drei wesentliche Begriffe, deren Bedeutungen nun geklärt werden:

Definition 26 (Kryptologie, Kryptografie, Kryptoanalyse)

Kryptologie ist der Überbegriff für die beiden Wissenschaften der Ver- und Entschlüsselung. Kryptografie bezeichnet dabei den Teil, der sich mit der Chiffrierung auseinandersetzt. Kryptoanalyse hingegen bezieht sich auf den Aspekt, der sich mit dem Brechen von verschlüsselten Texten und den zugehörigen Algorithmen beschäftigt. Da die Kryptografie stark mit der Kryptoanalyse zusammenhängt, wird wie bei Schmech [36, vgl. S. 11] im Folgenden zwischen Kryptologie und Kryptografie keine Unterscheidung vorgenommen.

3 Das diskrete-Logarithmen-Problem

Das Problem des diskreten Logarithmus (DLP) ist für einige kryptografische Verfahren von bedeutender Rolle. Für ein besseres Verständnis wird ausgehend vom allgemeinen Logarithmus der diskrete Logarithmus definiert. Im Anschluss werden die Grundlagen der Komplexitätstheorie erklärt und zugleich die Verwendung der diskreten Exponentiation motiviert. Danach wird das Zusammenspiel von Primitivwurzeln und diskreten Logarithmen erläutert. Zum Abschluss des Kapitels folgt mit dem Diffie-Hellman-Schlüsselaustausch ein konkretes Anwendungsbeispiel bei dem die Rolle des diskreten Logarithmus erklärt wird.

3.1 Der allgemeine Logarithmus

Logarithmen werden im aktuell achtjährigen bayrischen Gymnasium nach Lehrplan [38] in der 10. Klasse als Umkehrfunktion der Exponentialfunktion eingeführt.

Die Exponentialfunktion [16, vgl. S. 65] zu einer Basis $y \in \mathbb{R}^+$ definieren wir wie folgt:

$$\exp_y : \mathbb{R} \rightarrow \mathbb{R}^+, x \mapsto y^x = a$$

Mit Logarithmen hingegen ist es als Umkehrfunktion möglich zu einer gegebenen Basis $y \in \mathbb{R}^+$ und einer Potenz $a \in \mathbb{R}^+$ den Exponent $x \in \mathbb{R}$ zu bestimmen [16, vgl. S. 72].

$$\log_y : \mathbb{R}^+ \rightarrow \mathbb{R}, a \mapsto \log_y(a) = x$$

In folgender Abbildung sind beispielhaft Ausschnitte von \exp_2 und \log_2 zu sehen.

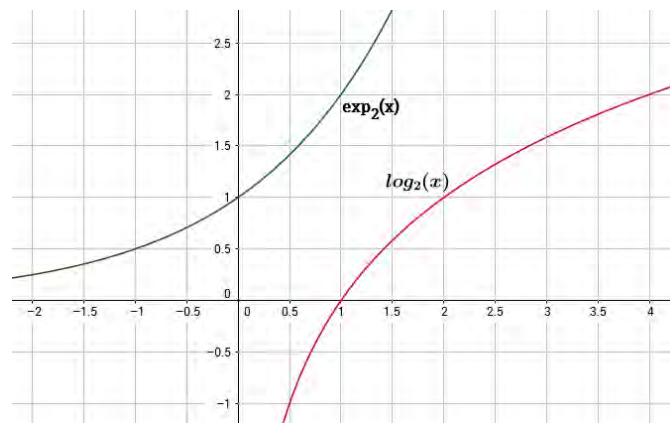


Abbildung 3.1: Graphen der Exponential- und Logarithmusfunktion zur Basis 2

3.2 Der diskrete Logarithmus

Im Gegensatz zum bisher bekannten Logarithmus haben die diskreten Exponential- und Logarithmusfunktionen andere Definitions- und Bildbereiche. Wir können sie wie folgt festlegen [17, vgl. S. 61], [9, vgl. S. 188]:

Definition 27 (diskrete Exponentiation, diskreter Logarithmus)

Sei G eine endliche zyklische Gruppe mit dem Erzeuger $y \in G$ und Ordnung n .

Die Abbildung

$$DExp_y : \mathbb{Z}/n\mathbb{Z} \rightarrow G, x \mapsto y^x = a$$

wird diskrete Exponentiation genannt. Die Abbildung

$$DLog_y : G \rightarrow \mathbb{Z}/n\mathbb{Z}, a \mapsto DLog_y = x$$

heißt diskreter Logarithmus.

Da $x \in \mathbb{Z}/n\mathbb{Z}$ ein ganzzahliges Element ist, erhält man die Begründung für den Bezeichner „diskret“, was im gegebenen Zusammenhang so viel wie ganzzahlig bedeutet.

In der Kryptografie werden gerne prime Restklassengruppen mit Primitivwurzeln als Grundlage für den diskreten Logarithmus verwendet. Diese Gruppen eignen sich, da sie wie gefordert zyklisch sind (siehe Definition 22).

Beispiel 28

Aus Beispiel 25 ist bekannt, dass $y = 3$ eine Primitivwurzel zur Einheitengruppe $(\mathbb{Z}/5\mathbb{Z})^*$ ist. Sie besitzt die Ordnung $n = 4$. Die Werte der diskreten Exponentialfunktion $DExp_3$ für $x \in \mathbb{Z}/4\mathbb{Z}$ sind:

x	0	1	2	3
$DExp_3(x)$	1	3	4	2

Tabelle 3.1: Diskrete Exponentiation zur Basis 3 in $(\mathbb{Z}/5\mathbb{Z})^*$

Damit lassen sich die Werte des diskreten Logarithmus für $a \in (\mathbb{Z}/5\mathbb{Z})^*$ angeben.

a	1	2	3	4
$DLog_3(a)$	0	3	1	2

Tabelle 3.2: Diskreter Logarithmus zur Basis 3 in $(\mathbb{Z}/4\mathbb{Z})$

Das Ausfüllen der Tabelle für den diskreten Logarithmus war nur möglich, da zuvor die Werte der zugehörigen diskreten Exponentiation berechnet worden waren. Dies führt nun zur Frage, warum die diskrete Exponentiation gerne in der Kryptologie verwendet wird.

3.3 Motivation für die Benutzung der diskreten Exponentiation

Ein Ziel der Kryptologie, der „Lehre der Verschlüsselung von Daten“ [36, S. 9], ist es, Informationen so zu verändern, dass Unbefugte aus verschlüsselten Daten den eigentlichen Inhalt nicht entnehmen können.

Theoretisch ist es möglich, sofern der Algorithmus bekannt ist, mit beliebig großer Zeit und Rechenleistung für ein gegebenes Problem eine oder mehrere Lösungen und somit die entschlüsselten Daten zu erhalten. Bedenklich ist dabei, dass bei mehreren gefundenen Ergebnissen die korrekte Dechiffrierung möglicherweise nicht von den anderen Lösungen unterschieden werden kann. Zudem besitzt man in der Realität keine beliebig große Rechenleistung und im Allgemeinen ist es auch nicht gewollt, zum Entschlüsseln eines Chiffrats mehrere Jahre (oder noch länger) zu warten. Daher müssen Aussagen getroffen werden, wie aufwendig es ist zu einem gegebenen kryptografischen (bzw. mathematischen) Problem eine Lösung zu finden.

Bemerkung 29

Der für einen Algorithmus nötige Aufwand bzw. sein Wachstum in Abhängigkeit von der Eingabegröße wird *Komplexität* genannt. Dabei wird die Komplexität bezüglich zweier Aspekte betrachtet:

- Zum einen interessiert die Laufzeitkomplexität. Hierbei wird die Anzahl an Operationen analysiert, die während des Algorithmus abgearbeitet werden; nicht die tatsächliche Laufzeit in Sekunden. Das liegt daran, dass die tatsächliche Zeit u.a. von der Implementierung, dem genutzten System und den sich fortwährend verbessernden Rechenkernen abhängt und somit keine repräsentative Information darstellt.
- Das andere Kriterium ist die Speicherkomplexität, bei der man die Menge der gespeicherten Daten analysiert. Dabei achten wir lediglich auf die Anzahl der zu speichernden Elemente und nicht auf die von der Implementierung abhängige Größe in Bytes.

Die Probleme bzw. die problemlösenden Algorithmen werden bzgl. ihres zeitlichen Aufwands oder Speicherbedarfs in verschiedene Kategorien eingeteilt. Dazu verwenden wir die Landau-Notation. Die Darstellung folgt hier Eckert [12, vgl. S. 314]:

Definition 30 (Landau-Notation)

Gegeben seien zwei Funktionen f und g mit Werten in \mathbb{R} . Wenn es eine Konstante $c > 0$ und ein $n_0 \in \mathbb{N}$ gibt, sodass

$$\forall n > n_0 : |f(n)| \leq c \cdot |g(n)|$$

gilt, schreiben wir

$$f(n) = \mathcal{O}(g(n)), \text{ für } n \in \mathbb{N} \text{ gegen Unendlich.}$$

Gilt $f(n) = \mathcal{O}(g(n))$ bedeutet dies, dass „ f bis auf einen konstanten Faktor c nicht schneller als g wächst“ [12, S. 314]. In der Kryptologie wird mit n oft die Länge eines

Eingabewertes, die Problemgröße, gemeint. Alternativ kann es z.B. auch die Ordnung der zu Grunde liegenden mathematischen Struktur (z.B. einer Gruppe) sein, in der die Berechnung stattfindet. Dies ist auch bei der Analyse von DLPs der Fall.

Häufig werden u.a. folgende Wachstumsarten verwendet, die hier nach aufsteigender Komplexität geordnet sind:

- $\mathcal{O}(1)$ - konstanter Aufwand
- $\mathcal{O}(\log(n))$ - logarithmischer Aufwand (Basis ist unerheblich)
- $\mathcal{O}(n)$ - linearer Aufwand
- $\mathcal{O}(n \cdot \log(n))$ - super-linearer Aufwand
- $\mathcal{O}(n^x)$, $x \in \mathbb{N} \setminus \{1\}$ - polynomieller Aufwand
- $\mathcal{O}(y^n)$, $y \in \mathbb{N} \setminus \{1\}$ - exponentieller Aufwand

Beispiel 31

Sei $f(n) = n^2 + 1$. Nun wählen wir z.B. $g(n) = n^2$, $c = 2$, $n_0 = 1$.

Wie wir in der folgenden Abbildung erkennen können, gilt nun $\forall n > n_0$, sodass $c \cdot g(n)$ (in rot) größergleich $f(n)$ (in grün) ist. Somit ist die obige Bedingung erfüllt und es gilt: $f(n) = \mathcal{O}(n^2)$

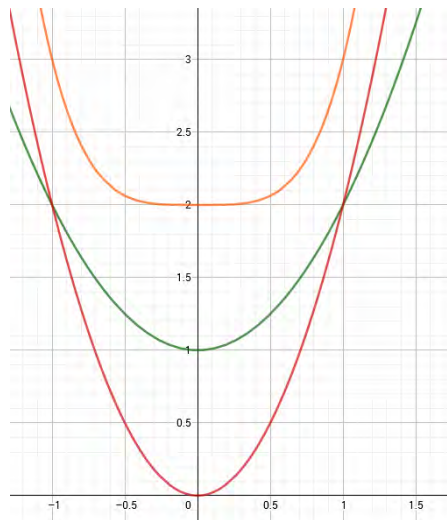


Abbildung 3.2: Drei Funktionen zur O-Notation

Bemerkung 32 (Uneindeutige Einteilung)

Die Einteilung in die Wachstumsarten ist dabei nicht eindeutig: Zum Beispiel könnten wir auch die Funktion $g'(n) = n^4 + 2$ (orange), $c = 1$ und $n_0 = 0$ wählen. $g'(n) = n^4 + 2$ ist (unabhängig von n') immer größer als $f(n) = n^2 + 1$. Es gilt also auch $f(n) = \mathcal{O}(n^4 + 2)$.

Wegen dieser Uneindeutigkeit werden im Allgemeinen für $g(n)$ meist einfache, bzw. bekannte Funktionen verwendet, die ein möglichst ähnliches Steigungsverhalten wie f besitzen.

Nun ist die Frage zu klären, was das Wachstumsverhalten über die Lösbarkeit der Probleme aussagt. Grundsätzlich teilt man in der Komplexitätstheorie Algorithmen in zwei Problemklassen ein: solche die „effizient“ gebrochen werden können oder eben nicht.

Definition 33 (effizient)

Nach Eckert [12, vgl. S. 314] werden mit „effizient“ alle Algorithmen bezeichnet, deren Komplexität höchstens polynomiellen Aufwand hat. Algorithmen mit dieser Eigenschaft sammelt man in der Klasse P.

Wie in der Auflistung oberhalb zu sehen ist, gehören also u.a. Algorithmen mit konstantem, logarithmischem, linearem, super-linearem und polynomiellen Aufwand in die Klasse P.

Bemerkung 34 (ineffizient)

Alle anderen Algorithmen, die bezüglich des gewählten Kategorisierungskriteriums (Zeit bzw. Speicherplatz), stärker als polynomiell wachsen, werden als „ineffizient“ bezeichnet. Die zugehörige Komplexitätsklasse heißt NP (nicht polynomiell). [41, vgl. S. 184]

Bemerkung 35 (Bezug zur Kryptologie)

Für eine gelungene Kommunikation ist es sinnvoll, dass die gewünschten Teilnehmer ihre Nachrichten schnell ver- und entschlüsseln können. Für ungewünschte Mithörer hingegen soll es nicht bzw. nur mit möglichst großem Aufwand machbar sein, mitgeschnittenen Daten Sinn entnehmen zu können. Mit anderen Worten soll die Ver- und Entschlüsselung zwischen den eigentlichen Teilnehmern „effizient“ geschehen, während der Aufwand für Eindringlinge „ineffizient“ sein soll.

Mit dem diskreten Logarithmus bzw. der diskreten Exponentiation kann ein genau solches Kommunikationssystem geschaffen werden (siehe Abschnitt 3.4). Während das Berechnen der diskreten Exponentiation „effizient“ möglich ist, fällt es hingegen sehr schwer DLPs zu lösen. Aus diesem Grund wird der diskrete Logarithmus als zahlentheoretisches Problem bezeichnet.

Bemerkung 36 (Spezialfälle der Einteilung)

Zwar gilt aufgrund der Kategorisierung, dass ineffiziente Algorithmen schneller wachsen als effiziente Algorithmen, aber dennoch sollte man zwei theoretische Eigenheiten der Begriffe kennen.

- Ineffiziente Algorithmen sind nicht immer „ineffizient“ zu lösen. Sei beispielsweise ein Algorithmus mit Zeitkomplexität $f(n) = 5^n + 1$ gegeben. Das bedeutet, dass zur Lösung dieses Algorithmus f mit einer Problemgröße n nun $5^n + 1$ Berechnungen durchgeführt werden müssen.

Es ist nach der Landau-Notation nun $f = \mathcal{O}(5^n)$, wodurch f in NP liegt (siehe Bemerkung 34). Beispielhaft sei ein konkretes Problem zu lösen, bei dem $n = 2$ gilt. Setzen wir dies in f ein, so müssen zur Lösung des Problems nur $5^2 + 1 = 26$

Berechnungen getätigt werden, was mit geringem Aufwand möglich ist. Die Bezeichnung eines Algorithmus mit „ineffizient“ bedeutet demnach nicht, dass er unlösbar ist.

- Aber auch die Benennung mit „effizient“ ist manchmal irreführend. Denn es kann sein, dass ein Problem, obwohl es in P liegend als „effizient“ klassifiziert wurde zu viel Rechenaufwand/Speicherplatz in Anspruch nimmt, um mit akzeptablen Kosten gelöst werden zu können.

Sei hierzu ein Algorithmus gegeben, der $g(n) = n^{1000} + 1$ viele Schritte zur Lösung benötigt. Es gilt nun $g = \mathcal{O}(n^{1000})$ und somit liegt g in P . Sei wieder ein konkretes Problem mit $n = 2$ gegeben. Jetzt sind $2^{1000} + 1$ Berechnungen nötig. Dies ergibt eine Zahl mit 302 Stellen, was im Vergleich zu dem obigen Beispiel f wesentlich mehr ist.

Dennoch macht die Kategorisierung mit den beiden Begriffen Sinn. Das liegt daran, dass in der Kryptografie die Entschlüsselung durch Unbefugte (ohne Kenntnis eines Schlüssels) nur mit „ineffizienten“ Algorithmen zu bewerkstelligen ist. Meistens werden auch Schlüssel mit einer Länge (n) aus hohen Zahlenbereichen verwendet, sodass „ineffizient“ zu entschlüsselnde Algorithmen auch tatsächlich „ineffizient“ sind. Problematisch wird es allerdings, wenn Wissenschaftler oder Hacker einen neuen Algorithmus finden, mit dem ein ursprünglich „ineffizientes“ Problem auf einmal mit einem „effizienten“ Algorithmus gelöst werden kann.

Der zweite Spezialfall kommt ebenfalls selten vor. In einem Kryptosystem ist die Verschlüsselung so konstruiert ist, dass dies von einem „effizienten“ Algorithmus erledigt wird. Zwar könnte es in einem Sonderfall sein, dass dafür überraschend viele Ressourcen verbraucht werden, wodurch die Gesamtsicherheit des Systems jedoch nicht geschwächt wird. Obwohl die Verschlüsselung länger dauern würde, verändert dies nichts an dem Aufwand, der für die unbefugte Entschlüsselung benötigt wird.

Zudem achten die Wissenschaftler, die Kryptosysteme entwerfen, auf derartige Spezialfälle und können somit zuverlässige und möglichst sichere Kommunikationsverfahren entwickeln.

Bemerkung 37 (Einsatz von Primitivwurzeln)

Aus diesem Grund wird für den Erzeuger y bei der diskreten Exponentiation auf Primitivwurzeln zurückgegriffen. Dadurch wird die Umkehrung, also die Berechnung des diskreten Logarithmus möglichst aufwendig. Primitivwurzeln spannen, wie bereits bekannt, die gesamte Gruppe G auf (siehe Definition 22). Würde man hingegen als Basis den Erzeuger einer Untergruppe $U \subsetneq G$ nutzen, so lägen in dessen Spann nicht alle Gruppenelemente von G . Dadurch wäre es bei der Berechnung des diskreten Logarithmus einfacher eine Lösung zu finden, da es weniger Möglichkeiten gäbe. Dies wird nun mit einem Beispiel verdeutlicht:

Beispiel 38

Sei G die Einheitengruppe $(\mathbb{Z}/13\mathbb{Z})^* = \{1, \dots, 12\}$ mit $\text{ord}(G) = 12$. Nach Bemerkung 23 gibt es $\varphi(\varphi(13)) = \varphi(12) = 4$ viele Primitivwurzeln. Somit ist G zyklisch.

2 ist eine der Primitivwurzeln von G , denn sie hat als erzeugendes Element von G die Ordnung 12:

i	1	2	3	4	5	6	7	8	9	10	11	12
$2^i \pmod{13}$	2	4	8	3	6	12	11	9	5	10	7	1

Tabelle 3.3: Die von 2 erzeugte Einheitengruppe (mod 13)

Wenn wir nun probieren, das DLP $2^x \equiv 9 \pmod{13}$ durch schrittweises Erhöhen des Exponenten zu lösen, benötigen wir 8 Versuche. Allgemein, ohne ein spezielles Lösungsverfahren anzugeben, müssen maximal $\text{ord}(G) = 12$ Berechnungen durchgeführt werden.

Nun sei die Basis des DLP die Zahl 3. Diese ist jedoch nur erzeugendes Element einer Untergruppe U mit Ordnung 3, da in ihrem Spann nur die Elemente 1, 3 und 9 liegen:

i	1	2	3
$3^i \pmod{13}$	3	9	1

Tabelle 3.4: Die von 3 erzeugte Untergruppe (mod 13)

Jetzt wird die vorherige Problemstellung mit der Basis 3 wiederholt und wir lösen das DLP $3^x \equiv 9 \pmod{13}$ durch schrittweises Inkrementieren des Exponenten. Dazu werden nun, wie in der Tabelle ersichtlich, nur noch zwei Versuche benötigt. In diesem Fall müssen generell maximal $\text{ord}(U) = 3$ Berechnungen durchgeführt werden.

Wie im Beispiel ersichtlich, lassen sich diskrete Logarithmen in primen Restklassengruppen normalerweise einfacher berechnen, wenn die gewählte Basis keine Primitivwurzel ist, sondern nur erzeugendes Element einer Untergruppe. Deswegen wird im Umgang mit diskreten Logarithmen bevorzugt eine Primitivwurzel als Basis verwendet, da diese die Berechnung des diskreten Logarithmus erschweren. Aus diesem Grund wird im Folgenden und auch bei der Implementierung davon ausgegangen, dass eine Primitivwurzel als Basis genutzt wird.

Bemerkung 39 (Benutzung des diskreten Logarithmus als Einwegfunktion)

Bisher wurde stets erwähnt, dass die Berechnung eines diskreten Logarithmus möglichst schwierig sein soll. Die Frage, die sich nun stellt, ist, in welchen Anwendungsbereichen ein solches Verfahren überhaupt sinnvoll ist.

In der Kryptologie werden oftmals Algorithmen verwendet, bei denen die Entschlüsselung einer Chiffre ohne einen geheimen Schlüssel nicht effizient möglich sein sollte. Sofern man jedoch dieses Geheimnis kennt, ist die Entschlüsselung traditionell mit angemessenem Aufwand machbar. Dadurch wird sichergestellt, dass nur eingeweihte Kommunikationspartner (mit Schlüssel) die Informationen aus den verschlüsselten Daten extrahieren können.

Bei diskreten Logarithmen gibt es jedoch keinen solchen Schlüssel und, wie oben erwähnt, ist eine „einfache“ Berechnung des diskreten Logarithmus auch nicht möglich, sofern man nicht für alle möglichen Werte die diskrete Exponentiation vornehmen möchte. Dies wäre in großen primen Restklassengruppen vom Aufwand her nicht machbar. Ein Algorithmus mit den Eigenschaften, dass er leicht zu berechnen, aber nur schwer umzukehren ist, wird Einwegfunktion genannt. Eine solche ist beispielsweise die diskrete Exponentiation (siehe Abschnitt 3.2).

Was hat es für einen Sinn, in der Kryptografie Daten zu verschlüsseln, die nur mit hohem Aufwand wieder entschlüsselt werden können? Die Antwort ist, dass die Kommunikationspartner die verschlüsselten Daten dennoch in irgendeiner Art und Weise nutzen können. Ein Beispiel dafür ist der Diffie-Hellman-Schlüsselaustausch (DHS):

3.4 Der Diffie-Hellman-Schlüsselaustausch

3.4.1 Verfahren

Die beiden an der Kommunikation beteiligten Personen heißen Alice und Bob. Sie wollen einen gemeinsamen Schlüssel erstellen, mit dem sie zukünftig ihre Kommunikation in einem symmetrischen Kryptosystem absichern. Das bedeutet, sie verwenden zur Ver- und Entschlüsselung jeweils den gleichen Schlüssel. Die Vereinbarung eines Schlüssels durch ein persönliches Treffen ist umständlich. Aber auch die Übermittlung des ganzen Schlüssels über digitale Wege ist unsicher, da ein Eindringling ihre Kommunikation belauschen könnte. Deswegen verwenden sie eine Methode, bei dem sie sich gegenseitig Informationen zum Schlüssel senden, sodass jeder für sich aus den erhaltenen Informationen den gemeinsamen Schlüssel berechnen kann:

1. Zunächst einigen sich Alice und Bob auf eine Primzahl p und eine Primitivwurzel g von $(\mathbb{Z}/p\mathbb{Z})^*$. Diese beiden Werte müssen nicht notwendigerweise geheimgehalten werden.
2. Anschließend wählt Alice eine zufällige Zahl $a \in \mathbb{Z}/(p-1)\mathbb{Z}$ aus, welche ihr „privater Schlüssel“ ist. Sie berechnet $g^a \pmod{p} \equiv x$ und sendet diesen Wert als „öffentlichen Schlüssel“ an Bob.
3. Auch Bob wählt eine zufällige Zahl $b \in \mathbb{Z}/(p-1)\mathbb{Z}$ als „privaten Schlüssel“ aus. Er ermittelt $g^b \pmod{p} \equiv y$ und übermittelt diesen „öffentlichen Schlüssel“ an Alice.
4. Alice nimmt Bobs „öffentlichen Schlüssel“ y und potenziert diesen mit ihrem eigenen „privaten Schlüssel“:

$$y^a \equiv (g^b)^a \equiv g^{ab} \pmod{p}$$

Das Gleiche macht auch Bob mit x und seinem „privaten Schlüssel“:

$$x^b \equiv (g^a)^b \equiv g^{ab} \pmod{p}$$

Auf diese Weise [41, vgl. S. 355] können Alice und Bob einen gemeinsamen Schlüssel g^{ab} erzeugen, den die beiden nun in einem symmetrischen Kryptosystem zur sicheren Kommunikation verwenden können.

Das Schema zum DHS findet sich auch in Abbildung 3.3.

3.4.2 Sicherheit

Alice und Bob können x und y theoretisch auf einem öffentlichen Kanal austauschen. Die Sicherheit des DHS beruht darauf, dass Eve - die Dritte, die den gemeinsamen Schlüssel herausfinden möchte - zwar p, g, x und y kennt, aber dennoch nicht den gemeinsamen Schlüssel berechnen kann. Dafür muss sie einen diskreten Logarithmus $DLog_p(x) = a$ oder $DLog_p(y) = b$ berechnen. Sofern dies nicht „effizient“ zu ermitteln ist, können Alice und Bob ohne Bedenken mit ihrem gemeinsamen Schlüssel g^{ab} kommunizieren.

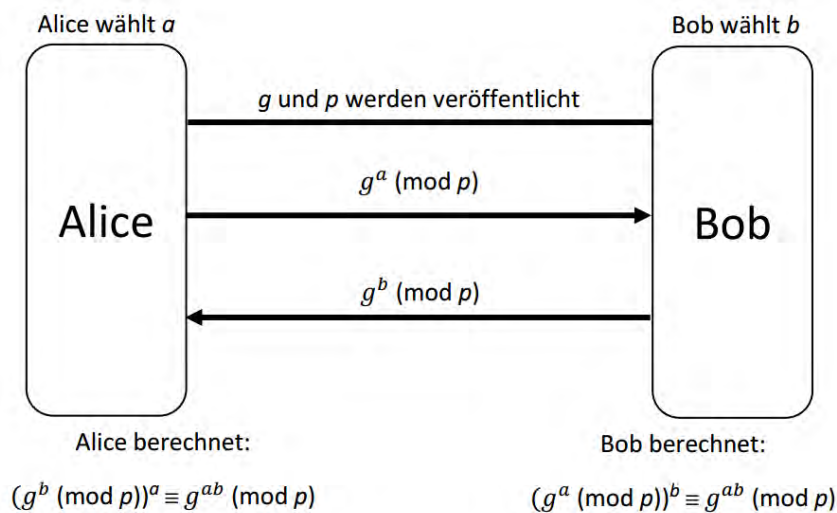


Abbildung 3.3: Diffie-Hellman-Schlüsselaustausch nach [41, S. 355]

Für Sicherheitsforscher, aber auch Eve, ist es also von großer Bedeutung, wie schnell ein diskreter Logarithmus berechnet werden kann. Dazu gibt es verschiedene Algorithmen, die im Folgenden vorgestellt werden.

4 Algorithmen

Es gibt mehrere Algorithmen zur Berechnung von diskreten Logarithmen. Sie unterscheiden sich in ihrer Laufzeit- und Speicherkomplexität sowie ihrer Anwendbarkeit. Yan [41, vgl. S. 254f] unterteilt dabei:

1. Algorithmen für beliebige endliche, zyklische Gruppen:
 - a) Shanks Babystep-Giantstep-Algorithmus
 - b) Pollard- ρ -Algorithmus
2. Algorithmen für beliebige endliche, zyklische Gruppen, die besonders effektiv sind, wenn die Gruppenordnung in viele kleine Primfaktoren zerlegt werden kann:
z.B.: Pohlig-Hellman-Algorithmus

Die Funktionsweise der drei Algorithmen wird im Folgenden beispielhaft in primen Restklassengruppen gezeigt. Dazu werden folgende Annahmen vereinbart: Gegeben sei eine prime Restklassengruppe $G = (\mathbb{Z}/p\mathbb{Z})^*$ mit $p \in \mathbb{P}$. Dadurch muss bei jeder Gruppenoperation $(\bmod p)$ angewendet werden. Das gegebene DLP lautet $y^x = a$, mit $a, y \in G$. Die Ordnung von y bzw. G wird mit $n = \varphi(p)$ bezeichnet.

4.1 Shanks Babystep-Giantstep-Algorithmus

Daniel Shanks veröffentlichte 1971 einen Algorithmus zur besseren Berechnung von diskreten Logarithmen [37]. Dieser nutzt zwei zentrale Schritte, die die hauptsächliche Kalkulation übernehmen und jeweils eine Tupel-Menge erzeugen. Shanks nannte diese die „Babystep“- bzw. „Giantstep“-Mengen. Hiervon kommt auch der Name Babystep-Giantstep-Algorithmus (BSGS). Nach Buchmann [9, S. 218f] kann er wie folgt beschrieben werden:

4.1.1 Verfahren

1) Initialisierung

Definiere zunächst die neue Variable $m = \lceil \sqrt{n} \rceil$.

Anschließend schreiben wir die Lösung x des diskreten Logarithmus um:

$$\begin{aligned} DLog_y(a) &= x \\ &= qm + r \end{aligned} \tag{4.1}$$

Dabei sind $r, q \in \{0, 1, \dots, m-1\}$ eindeutig bestimmt ([42, vgl. S. 26ff]), aber noch unbekannt.

Es gilt nun:

$$\begin{aligned}
a &= y^x \\
&= y^{qm+r} \\
&= y^{qm} \cdot y^r \\
\Leftrightarrow a \cdot y^{-r} &= y^{qm}
\end{aligned}$$

Im nächsten Schritt bestimmen wir j als das Inverse von y in G . Dies geschieht z.B. mit dem erweiterten euklidischen Algorithmus [5, vgl. S. 76] oder dem Satz von Euler 21. Wir formen um:

$$\begin{aligned}
y^{qm} &= a \cdot y^{-r} \\
&= a \cdot j^r
\end{aligned} \tag{4.2}$$

2) Berechnung der Babysteps B

Ein Babystep ist das Ergebnis einer Berechnung der Form $a \cdot j^r$, wobei a und j bekannt sind. Es gilt $0 \leq r < m$. Die Ergebnisse der Berechnungen werden nun in Tupeln mit dem zugehörigen Wert von r abgespeichert.

$$B = \{(a \cdot j^r, r) : 0 \leq r < m\} \tag{4.3}$$

3) Schrittweises Bestimmen der Giantsteps

Wir gehen nun wieder von Gleichung 4.2 aus: $y^{qm} = a \cdot j^r$ und führen wie folgt δ ein:

$$\begin{aligned}
a \cdot j^r &= y^{qm} \\
&= (y^m)^q \\
&= \delta^q
\end{aligned}$$

Mit der gegebenen Kenntnis von m und y können wir $\delta = y^m$ ermitteln. Jetzt berechnen wir iterativ in maximal m Schritten Giantsteps der Form δ^{q_i} mit $q_i = i$ und $i \in \{0, m-1\}$.

4) Auffinden einer Übereinstimmung

Im i -ten Schritt wird δ^{q_i} berechnet. Dabei wird sofort geprüft, ob es ein Tupel in B (siehe Gleichung 4.3) gibt, das in der ersten Komponente dem Wert von δ^{q_i} entspricht. Wird eine Übereinstimmung gefunden, gilt:

$$\begin{aligned}
&\delta^{q_i} = a \cdot j^r \\
\Leftrightarrow \delta^{q_i} &= a \cdot y^{-r} \\
\Leftrightarrow (y^m)^{q_i} &= a \cdot y^{-r} \\
\Leftrightarrow y^{q_i \cdot m} &= a \cdot y^{-r} \\
\Leftrightarrow y^{q_i \cdot m} \cdot y^r &= a \\
\Leftrightarrow y^{q_i \cdot m + r} &= a
\end{aligned}$$

$y^{q_i \cdot m + r} = a$ entspricht der gesuchten Darstellung und sowohl m, r als auch q_i sind bekannt. Nun lässt sich mit $x = q_i \cdot m + r$ (siehe Gleichung 4.1) der diskrete Logarithmus $DLog_y(a) = x$ berechnen.

5) Sukzessives Inkrementieren von q_i

Wird mit dem Wert von q_i keine Lösung gefunden, so wird Schritt 4 mit q_{i+1} wiederholt. Wenn der Algorithmus bis q_m läuft und keine Lösung gefunden wurde, so existiert diese nicht und das Verfahren stoppt. Dies ist nur der Fall, wenn die Ausgangswerte kein DLP darstellen.

4.1.2 Laufzeitkomplexität und Speicherbedarf

Nach Buchmann [9, vgl. S. 220] kann der diskrete Logarithmus durch Shanks Algorithmus bezüglich einer Gruppe G mit $\mathcal{O}(\sqrt{\text{ord}(G)})$ vielen Operationen und ebensoviel Speicherplatz gelöst werden.

Laufzeitkomplexität

- Die Bestimmungen von $m = \lceil \sqrt{n} \rceil$ und j als Inverses von y werden vernachlässigt, da sie für große Gruppenordnungen n irrelevant werden.
- Für die Berechnung der Babystep-Tupel werden $m = \lceil \sqrt{n} \rceil$ Berechnungen ausgeführt. Da in jedem Schritt das vorangehende Ergebnis mit j multipliziert werden muss, ist jeweils nur eine Gruppenoperation nötig [22, vgl. S. 308], wodurch sich eine Komplexität von $\mathcal{O}(\sqrt{n})$ ergibt.
- Am geschicktesten ist es, die Babystep-Tupel für die spätere Suche in einer Hash-tabelle abzuspeichern. Sofern es die Hash-tabelle schafft, mit konstantem Aufwand ein Element zu finden, besitzt dieser Vorgang eine konstante Zugriffskomplexität $\mathcal{O}(1)$.
- Im Anschluss werden die Giantsteps berechnet. Dazu gibt es maximal $m = \lceil \sqrt{n} \rceil$ Schritte, wobei in jedem Schritt eine Multiplikation und eine Suche (über die Hash-tabelle in $\mathcal{O}(1)$) durchgeführt werden. Folglich hat die Giantstep-Phase eine maximale Komplexität von $\mathcal{O}(2 \cdot \sqrt{n})$.

Insgesamt ergibt sich also eine Komplexität von

$$\mathcal{O}(\sqrt{n}) + \mathcal{O}(2 \cdot \sqrt{n}) = \mathcal{O}(\sqrt{n})$$

Obwohl dieser Algorithmus in seiner Laufzeit wesentlich besser ist als ein Brute-Force-Angriff (iteratives Ausrechnen aller diskreten Exponentationen), kommt er nach Buchmann [9, vgl. S. 220] bei Ordnungen $n > 2^{160}$ an seine Einsatzgrenze.

Speicherkomplexität

Die von n abhängig zu speichernden Elemente sind gerade die Elemente der Babystep Menge. Auch das sind $\mathcal{O}(\sqrt{n})$ viele. Der reale Speicherbedarf wird anhand von Testdaten, die mithilfe der eigens für diese Arbeit entwickelten Software gewonnen wurden, in Kapitel 6 analysiert.

4.1.3 Zusammenfassung des Algorithmus

```
Definiere  $m = \lceil \sqrt{n} \rceil$ 
Bestimme mit dem erweiterten euklidischen Algorithmus  $j$  als Inverses von  $y$  zu  $p$ 
Berechne die Menge der Babysteps:  $B = \{(a \cdot j^r, r) : 0 \leq r < m\}$ 
Bestimme  $\delta = y^m$ 
for ( $i = 0; i < m; i++$ )
    Berechne Giantstep  $\delta^i$ 
    if (Für einen Babystep  $(a \cdot j^r, r)$  gilt  $a \cdot j^r = \delta^i$ )
        Terminiere mit  $i \cdot m + r$  als Lösung des diskreten Logarithmus
Terminiere ohne Lösung
```

4.1.4 Beispiel

Das Problem sei $DL_6(15) = x$ in $(\mathbb{Z}/41\mathbb{Z})^*$. $p = 41, y = 6, a = 15$
Nach der Eulerschen φ -Funktion gilt: $n = \varphi(41) = 41 - 1 = 40$ $n = 40$

Initialisierung

$m = \lceil \sqrt{40} \rceil = 7$ $m = 7$

Nun berechnen wir mit dem erweiterten euklidischen Algorithmus
das Inverse von 6 in $(\mathbb{Z}/41\mathbb{Z})^*$. Dieses ist $j = 7$. $j = 7$

Jetzt ist die Gleichung $15 \cdot 7^r = 6^{q \cdot 7}$ zu lösen.

Babysteps

Berechne die Menge der Babysteps: $B = \{(15 \cdot 7^r, r) : 0 \leq r < 7\}$

r	0	1	2	3	4	5	6
$15 \cdot 7^r \pmod{41}$	15	23	38	20	17	37	13

Giantsteps

Anschließend bestimmen wir $\delta \equiv 6^7 \pmod{41} \equiv 29$. $\delta = 29$

Mit $29^q \pmod{41}$ und $q \in \{0, \dots, 6\}$ können jetzt die Giantsteps bestimmt werden:

q	0	1	2	3	4	5
$29^q \pmod{41}$	1	29	21	35	31	38

In der fünften Iteration wird eine Übereinstimmung gefunden, sodass wir die Lösung berechnen können:

$$x = m \cdot q + r = 7 \cdot 5 + 2 = 37$$

Der diskrete Logarithmus $DL_6(15)$ hat somit in $(\mathbb{Z}/41\mathbb{Z})^*$ die Lösung $x = 37$.

Das gleiche Ergebnis liefert auch die Implementierung, die in Java umgesetzt wurde.

```

----- Shanks Babystep-Giantstep Algorithmus -----

~~~~ Initialisierung ~~~~
Primzahl p:          41
Primitivwurzel y:    6
Potenz a:            15
Ordnung n:           40
Wurzel m:            7
Inverses j:          7
Delta d:             29

~~~~ Babysteps ~~~~
(15, 0)
(23, 1)
(38, 2)
(20, 3)
(17, 4)
(37, 5)
(13, 6)

~~~~ Giantsteps ~~~~
(0, 1)
(1, 29)
(2, 21)
(3, 35)
(4, 31)
(5, 38)

Ein passender Babystep wurde gefunden:  38

Das Ergebnis ist:      37

```

Abbildung 4.1: Ausgabe für das DLP $6^x \equiv 15 \pmod{41}$ mit Shanks BSGS Algorithmus

4.2 Pollard-Rho-Algorithmus

Ein weiterer Algorithmus zur Bestimmung von diskreten Logarithmen wurde 1978 von John M. Pollard entwickelt [33].

Der Name des Verfahrens kommt daher, dass im Zuge der Berechnung eine Folge verwendet wird, die sich in der Form des griechischen Buchstabens Rho ρ schreiben lässt (siehe Abschnitt 4.2.3).

Der Vorteil gegenüber dem BSGS-Algorithmus ist, dass bei gleichbleibender Laufzeitkomplexität der Speicherbedarf letztendlich auf eine konstante Komplexität von $\mathcal{O}(1)$ reduziert werden kann (siehe Abschnitt 4.2.4).

Für die anschließende algorithmische Erarbeitung werden zwei Aussagen benötigt:

Satz 40

Seien $\gamma, \eta \in \mathbb{N}$ und $\text{ggT}(\gamma, \eta) = 1$. Sei e die Ordnung von γ in der primen Restklassengruppe $((\mathbb{Z}/\eta\mathbb{Z})^*, \cdot)$. Seien $i, j \in \mathbb{N}$ beliebig.

Dann gilt: $\gamma^i \equiv \gamma^j \pmod{\eta} \Leftrightarrow i \equiv j \pmod{e}$.

Dieser Satz findet sich inklusive Beweis bei Wagstaff [40, S. 88].

Weiterhin benötigen wir folgendes Lemma [5, vgl. S. 104], [41, vgl. S. 123f], zu dessen Beweis auf Wagstaff [40, S. 65f] verwiesen wird:

Lemma 41 (Lösbarkeit von linearen Kongruenzen)

Sei $\mu \in \mathbb{N}$ und $\nu, \varrho \in \mathbb{Z}$ beliebige Werte. Die Kongruenz $\nu \cdot x \equiv \varrho \pmod{\mu}$ mit $d := \text{ggT}(\nu, \mu)$ hat eine Lösung, genau dann, wenn $d \mid \varrho$ gilt.

In diesem Fall gibt es bezüglich $\pmod{\mu}$ genau d viele Lösungen, die wie folgt gefunden werden können:

$$x \equiv \hat{x} + \frac{\kappa\mu}{d} \pmod{\mu}, \quad \kappa = 0, 1, \dots, d-1$$

wobei \hat{x} eine Lösung der Kongruenz

$$\frac{\nu}{d} \cdot \hat{x} \equiv \frac{\varrho}{d} \pmod{\frac{\mu}{d}}$$

ist.

4.2.1 Verfahren

Die Beschreibung des Algorithmus orientiert sich an Buchmann [9, vgl. S. 220] sowie Baumslag et al. [5, vgl. S. 104f]:

1. Schritt: Aufteilung der Gruppenelemente

Zunächst müssen die Elemente von G drei paarweise disjunkten Mengen G_1, G_2, G_3 zugewiesen werden, sodass $G = G_1 \cup G_2 \cup G_3$ gilt. Damit der Algorithmus möglichst gut funktioniert, sollten die drei Mengen annähernd gleich groß sein. Ohne Einschränkung der Allgemeinheit gehen wir davon aus, dass die Teilmengen $\{G_i\}_{i=1,2,3}$ so nummeriert sind, dass $1 \notin G_2$ ist, wobei 1 das neutrale Element von G bezeichnet. Andernfalls werden die Teilmengen entsprechend vertauscht.

Eine Einordnung in die drei Mengen kann z.B. durch die Aufteilung nach den Äquivalenzklassen $\pmod{3}$ erreicht werden [11, vgl. S. 100].

2. Schritt: Bestimmung zweier identischer Folgenglieder

Zentraler Bestandteil dieses Algorithmus ist eine unendliche Folge an Gruppenelementen. Die Idee dazu ist, dass es aufgrund der endlichen Gruppe ($\text{ord}(G) = n \neq \infty$) in einer unendlichen Folge von Gruppenelementen irgendwann zu einer Wiederholung kommen muss. Mit diesem Wissen lassen sich später weitere Schlüsse ziehen.

Die unendliche Folge wird durch eine von Pollard verwendete Funktion gebildet:

$$f : G \rightarrow G, \quad \beta_i \mapsto f(\beta_i) = \beta_{i+1} = \begin{cases} y \cdot \beta_i \pmod{p} & \text{für } \beta_i \in G_1 \\ (\beta_i)^2 \pmod{p} & \text{für } \beta_i \in G_2 \\ a \cdot \beta_i \pmod{p} & \text{für } \beta_i \in G_3 \end{cases}$$

mit $i \in \mathbb{N}_0$. Durch iterative Anwendung der Funktion f erhalten wir also eine Folge $\{\beta_i\}_{i \in \mathbb{N}_0}$. Dazu wird zu einem fest gewählten $w_0 \in \{0, \dots, n-1\}$ der Startwert auf $\beta_0 = y^{w_0}$ gesetzt.

Baumslag et al. [5, vgl. S. 104] schlagen vor, $\beta_0 \equiv 1 \equiv y^0 \pmod{p}$ zu wählen.

Ausgehend von der obigen Definition der Funktion f lassen sich die Folgenglieder β_i als Produkt der Potenzen von a und y mit den zugehörigen Exponenten $w_i, v_i \in \mathbb{N}_0$ schreiben.

$$\beta_i = y^{w_i} \cdot a^{v_i}, \forall i \geq 0$$

Im weiteren Verlauf des Algorithmus ist es notwendig, sich zu einem Schritt i den Wert des Folgenglieds β_i sowie die Werte der zugehörigen Exponenten w_i, v_i zu merken.

Für die Veränderung der Werte w_i und v_i gilt durch die einmalige Anwendung der Funktion f :

$$w_{i+1} = \begin{cases} w_i + 1 \pmod{n} & \text{für } \beta_i \in G_1 \\ 2 \cdot w_i \pmod{n} & \text{für } \beta_i \in G_2 \\ w_i \pmod{n} & \text{für } \beta_i \in G_3 \end{cases}$$

sowie

$$v_{i+1} = \begin{cases} v_i \pmod{n} & \text{für } \beta_i \in G_1 \\ 2 \cdot v_i \pmod{n} & \text{für } \beta_i \in G_2 \\ v_i + 1 \pmod{n} & \text{für } \beta_i \in G_3 \end{cases}$$

Werden die Anfangswerte wie oben gewählt, entspricht das erste 3-Tupel mit den Einträgen für (β_0, w_0, v_0) dem Tupel $(1, 0, 0)$.

In Schritt 1 wurde darauf hingewiesen, dass das neutrale Element nicht in der Menge G_2 enthalten sein darf. Würde dies der Fall sein, also $\beta_i = 1 \in G_2$, so würden alle weiteren Folgenglieder auch dem neutralen Element entsprechen, da wegen $\beta_i \in G_2$ nun $(\beta_i)^2 = 1^2 = 1 = \beta_{i+1}$ gelten würde.

Wie bereits erwähnt, kann mit der Funktion f und den gegebenen Startwerten eine unendliche Folge an Gruppenelementen erzeugt werden. Da es allerdings nur endlich bzw. n viele Elemente in G gibt, kommt es spätestens im $i = n + 1$ -ten Schritt zu einer Kollision. Das bedeutet, dass das neue Folgeelement bereits in der Folge enthalten ist. Es gibt nach Buchmann [9, vgl. S. 221] also zwei Werte $k, l \geq 0$ und $k \neq l$ mit

$$\begin{aligned}
& \beta_k \equiv \beta_l \pmod{p} \\
\Rightarrow & y^{w_k} \cdot a^{v_k} \equiv y^{w_l} \cdot a^{v_l} \pmod{p} \\
\Rightarrow & y^{w_k - w_l} \equiv a^{v_l - v_k} \pmod{p} \\
\Rightarrow & y^t \equiv a^s \pmod{p}, \text{ mit } t = w_k - w_l \text{ und } s = v_l - v_k
\end{aligned} \tag{4.4}$$

3. Schritt: Bestimmen des diskreten Logarithmus

Aus dem letzten Schritt ist bekannt, dass $y^t \equiv a^s \pmod{p}$ gilt. Zusätzlich wissen wir, dass das initiale Problem der Form $a \equiv y^x \pmod{p}$ lautet.

$$\begin{aligned}
& y^x \equiv a \pmod{p} \\
\Rightarrow & y^{xs} \equiv a^s \pmod{p} \\
\Rightarrow & y^{xs} \equiv y^t \pmod{p}
\end{aligned}$$

Nun wird Satz 40 mit $\gamma = y, \eta = p, e = n, i = xs$ und $j = t$ angewendet. Mit dem Wissen, dass $ggT(y, p) = 1$ und die Ordnung von y als erzeugendes Element $o(y) = n$ ist, gilt:

$$\begin{aligned}
& y^{xs} \equiv y^t \pmod{p} \\
\Leftrightarrow & xs \equiv t \pmod{n} \\
\Leftrightarrow & x \cdot (v_l - v_k) \equiv w_k - w_l \pmod{n}
\end{aligned} \tag{4.5}$$

Falls $(v_l - v_k) = 0$ gilt, was allerdings sehr unwahrscheinlich ist [11, vgl. S. 101], muss der Algorithmus an dieser Stelle abgebrochen und mit einem neuen beliebigen Startwert $\beta_0 \in G$ begonnen werden. Ansonsten fahren wir wie folgt fort:

Sofern x eine existierende Lösung des DLP ist, folgt, dass auch die gegebene Kongruenz (mindestens) eine Lösung besitzt.

Nun wird Lemma 41 verwendet. Wir setzen $n = \mu, s = \nu$ und $t = \varrho$. Daraus erhalten wir, dass mit $d := ggT(s, n)$ auch $d|t$ gilt.

Außerdem wird die Kongruenz

$$\frac{s}{d} \cdot \hat{x} \equiv \frac{t}{d} \pmod{\frac{n}{d}}$$

aufgestellt und gelöst, um \hat{x} zu erhalten. Dies geschieht, indem die Kongruenz mit dem Inversen von $\frac{s}{d}$, also $(\frac{s}{d})^{-1}$, multipliziert wird. Abschließend werden die d vielen möglichen Lösungen für x in Abhängigkeit von κ berechnet.

$$\hat{x}_\kappa = \hat{x} + \frac{\kappa n}{d} \pmod{n}, \quad \kappa = 0, 1, \dots, d - 1$$

Von den d vielen Lösungen \hat{x}_κ gibt es nur eines, das auch die Lösung des diskreten Logarithmus ist. Deswegen müssen, falls die Lösung nicht eindeutig ist ($d > 1$), die gefundenen \hat{x}_κ durch Einsetzen in $y^{\hat{x}_\kappa} = a$ überprüft werden, um das $\hat{x}_\kappa \equiv x$ zu finden, welches das DLP löst.

4.2.2 Laufzeitkomplexität und Speicherbedarf

Laufzeitkomplexität

Der erste Schritt des Algorithmus, die Aufteilung der Elemente von G in die drei Mengen, wird nicht für alle Elemente durchgeführt. Das würde bei großen Primzahlen (und somit großen Ordnungen) einen großen Aufwand verursachen.

Um diesen zu vermeiden, genügt es im Schritt 1 eine Methode festzulegen, mit der beim Berechnen der Folgenglieder (Schritt 2) mit möglichst wenig Aufwand überprüft werden kann, zu welcher Menge das jeweilig erzeugte Element zugeordnet ist. Dies kann durch die erwähnte Modulo-3-Rechnung geschehen. Diese Berechnung wird in den nächsten Schritt eingebettet.

Schritt 2 beschäftigt sich mit der Erzeugung und Suche von wertgleichen Elementen in der Folge und ist das (rechenaufwendige) Herzstück des Algorithmus. Hierbei ist von besonderer Bedeutung, dass das Auftreten jedes Elements in der Folge gleich wahrscheinlich ist. Obwohl die Einteilung in die drei Mengen nicht zufällig ist, sondern auf einem Zuordnungsschema (z.B. der erwähnten Modulo-3-Rechnung) beruht, wird durch diese Konstruktion dennoch die Entstehung einer von y , w_0 und a unabhängigen „Zufallsfolge“ gewährleistet. Somit kann nun ein bekanntes Prinzip der Stochastik angewandt werden [9, vgl. S. 221]:

Bemerkung 42 (Das Geburtstagsparadoxon)

Nach Wagstaff [40, vgl. S. 19] sei dazu eine n -elementige Menge $\{1, \dots, n\}$ gegeben. Es werden k Elemente aus dieser Menge gewählt, wobei die Wahrscheinlichkeit für alle Elemente $\{1, \dots, n\}$ unabhängig sowie gleichverteilt ist und doppelte Elemente erlaubt sind. Das Geburtstagsparadoxon besagt nun, dass $\approx 1,18 \cdot \sqrt{n} = k$ viele Elemente ausgewählt werden müssen, damit die Wahrscheinlichkeit für ein Duplikat 50% übersteigt.

Das Geburtstagsparadoxon kann auch für die Auftrittswahrscheinlichkeit zweier Duplikate in der bei Pollard generierten Folge angewandt werden. Das bedeutet nun, dass $1,18 \cdot \sqrt{n}$ (mit n als Gruppenordnung) viele Folgenglieder erzeugt werden müssen, damit die Wahrscheinlichkeit für ein beliebiges Duplikat 50% beträgt.

Ergänzen wir hier die aus Schritt 1 übernommene Bestimmung der zugehörigen Menge, erhalten wir $2 \cdot 1,18 \cdot \sqrt{n}$ Algorithmusoperationen für eine 50% Kollisionswahrscheinlichkeit. Die Suche in der Folge aller Elemente kann mit einer Datenstruktur wie einer Hashtabelle wieder mit konstantem Aufwand ablaufen.

Wenn eine Kollision gefunden wurde, müssen nur noch wenige Berechnungen stattfinden: Zwei Subtraktionen $t = w_k - w_l$ sowie $s = v_l - v_k$, die Berechnung von d als größtem gemeinsamen Teiler und die zugehörigen Divisionen. Anschließend erfolgt die Inversion von $\frac{s}{d}$ und die Multiplikation mit $\frac{c}{d}$. Zuletzt werden die d -vielen Lösungen für x berechnet und durch Einsetzen überprüft. Der Aufwand für diese konstant vielen Rechnungen

kann vernachlässigt werden.

Insgesamt ergibt sich also ein Aufwand von $\mathcal{O}(2 \cdot 1,18 \cdot \sqrt{n}) = \mathcal{O}(\sqrt{n})$. Allerdings ist dies keine Worst-Case-Laufzeit, denn mit dem Geburtstagsparadoxon kann lediglich der Erwartungswert angegeben werden [11, vgl. S. 100]. Somit wird auch mit diesem Algorithmus immer eine Lösung gefunden (sofern sie existiert). Der dafür benötigte Aufwand kann jedoch stark vom Erwartungswert abweichen. Diese Schwankungen werden später auch in Kapitel 6 genauer betrachtet.

Speicherkomplexität

Der Speicheraufwand verhält sich bei der aktuellen Variante des Algorithmus wie folgt: Zunächst werden die Primzahl, in deren multiplikativer Gruppe gerechnet wird, deren Ordnung, die Primitivwurzel und die Potenz gespeichert. Dies sind also unabhängig von n immer vier Elemente.

Im Schritt 2 wird die Folge der β_i aufgebaut. Zusätzlich wird zu jedem Folgeelement auch noch v_i und w_i abgespeichert. Verwenden wir wieder das Geburtstagsparadoxon als Grundlage, so erhalten wir mit den erwarteten Folgengliedern für eine Kollision $3 \cdot 1,18 \cdot \sqrt{n}$ zu speichernde Elemente.

Die im letzten Schritt benötigten Elemente sind s, t, d, n, \hat{x} sowie das Element x als mögliche Lösung. Somit ist auch hier die Anzahl der zu speichernden Elemente von n unabhängig und somit konstant.

Zusammengefasst spielt also nur der im zweiten Schritt benötigte Speicher eine Rolle, womit eine Speicherkomplexität von $\mathcal{O}(3 \cdot 1,18 \cdot \sqrt{n}) = \mathcal{O}(\sqrt{n})$ zu erwarten ist.

4.2.3 Periodizität der Folge

Dass es aufgrund der endlichen Gruppe und der unendlichen generierten Folge β_i zwei gleiche Folgenglieder β_l und β_k gibt, wurde bereits erläutert. Seien β_l und β_k nun das erste wertgleiche Folgenpaar. Buchmann [9, vgl. S. 222] schreibt die beiden Elemente wie folgt um:

O.E. sei $l < k$, dann gibt es ein $z \in \mathbb{N} : k = l + z$. Somit gilt $\beta_k = \beta_{l+z}$ und z ist der Abstand der beiden gleichen Folgeelemente.

Da $\beta_{l+z} = \beta_k = \beta_l$ gilt, sind diese Elemente auch in der gleichen Menge G_1, G_2 oder G_3 zu finden. Dies bedeutet, dass aufgrund der Fallunterscheidung in der Funktion f die Berechnung für β_{l+z+1} und β_{l+1} gleich abläuft und somit $\beta_{l+z+1} = \beta_{l+1}$ gilt. Dieses Erkenntnis lässt sich nun auf alle weiteren Folgenglieder fortführen [11, vgl. S. 101]:

$$\beta_i = \beta_{i+z} \quad \forall i \in \mathbb{N} \text{ mit } i \geq l$$

Aus dieser Schreibweise lässt sich erkennen, dass sich die Folgeelemente ab dem Wert β_l fortwährend periodisch wiederholen. Deswegen werden die Folgenglieder $\beta_l, \dots, \beta_{l+z-1}$ auch „Periode“ der Länge z genannt. Die anfänglich erzeugten Elemente von β_0 bis β_{l-1} werden hingegen als Vorperiode bezeichnet [9, vgl. S. 222]. Pollard hat diese Eigenschaften grafisch mit dem griechischen Buchstaben Rho visualisiert:

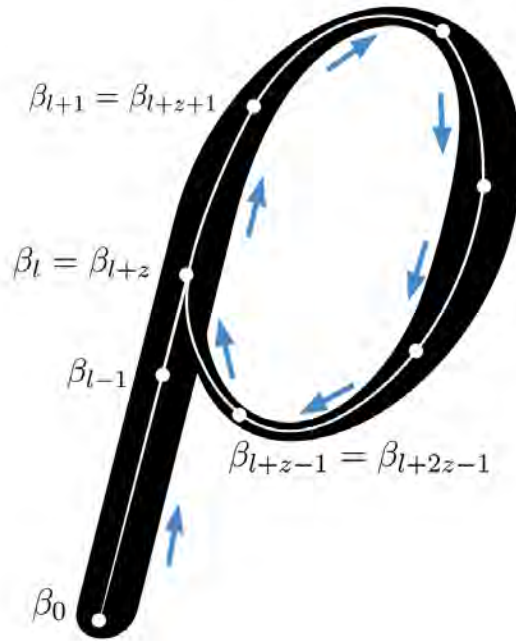


Abbildung 4.2: Grafische Visualisierung der Rho-Periodizität

4.2.4 Optimierung der Speicherkomplexität

Ausgehend von diesem Wissen können wir die Suche nach identischen Folgengliedern (Schritt 2) modifizieren, um die Speicherkomplexität zu verbessern. Dazu werden zwei verschiedene Ansätze diskutiert:

Speichern von Elementen einer Gruppe

Baumslag et al. [5, vgl. S. 105] nennen die Möglichkeit, nur die Folgenglieder β_i (bzw. die zugehörigen v_i, w_i) zu speichern, die in einer fest gewählten der drei Mengen G_1, G_2 oder G_3 liegen.

Wenn die Elemente der primen Restklassengruppe zufällig (bzw. unabhängig von Folgen-sequenzen), z.B. durch die oben genannte Methode mit Modulo-3, auf die drei Mengen G_1, G_2 und G_3 verteilt werden, ist es sehr wahrscheinlich, dass in jeder der drei Mengen mindestens ein Element aus der Periode liegt. Dadurch ist es möglich eine Kollision zu finden, obwohl wir nur maximal $\frac{n}{3}$ Elemente speichern müssen. Der wahrscheinliche Speicherbedarf würde sich dadurch auf ein Drittel gegenüber der bisherigen Variante reduzieren. Allerdings würde dies die Komplexität nicht beeinflussen, da für die erwartete Speicherkomplexität $\frac{1}{3} \cdot \mathcal{O}(\sqrt{n}) = \mathcal{O}(\sqrt{n})$ gilt.

Eine negative Eigenschaft dieser Methodik ist allerdings, dass es - im unwahrscheinlichen Fall - möglich ist, dass keines der Periodenelemente in der gewählten Menge liegt. Somit würden nur Elemente der Vorperiode gespeichert werden, wodurch es nicht möglich wäre Kollisionen zu finden. Ohne weitere Modifikation des Algorithmus würde dieser in einem

solchen Fall fortwährend weiterlaufen und nicht terminieren.

Brents Phasenansatz

Eine bessere Variante ist hingegen der Phasenansatz, der von Brent [8] 1980 veröffentlicht wurde und zu einer konstanten Speicherkomplexität führt.

Das Berechnen der Folgeelemente β_i wird dabei nach Diekert et al. [11, vgl. S. 101] in Phasen unterteilt, die mit einem Index q zusammenhängen. Zur Vereinfachung der Schreibweise wird das erste Folgeelement statt β_0 nun mit β_1 bezeichnet.

Initial wird $q = 1$ gesetzt. Am Anfang jeder Phase werden die bereits berechneten Werte β_q (sowie w_q und v_q) als aktuelle Phasenwerte gespeichert. Anschließend berechnen wir in q Runden die nächsten Folgeelemente: $\beta_{q+k}, k \in \{1, \dots, q\}$ mit den zugehörigen Werten w_{q+k} und v_{q+k} . Jedes dieser Folgeelemente wird mit dem zu Beginn der Phase gespeicherten Wert β_q verglichen. Gilt $\beta_q = \beta_{q+k}$ für ein $k \in \{1, \dots, q\}$, so haben wir eine Übereinstimmung gefunden und kann mit dem dritten Schritt des bisherigen Algorithmus fortfahren. Wird kein identisches Folgenglied gefunden, so wird q verdoppelt und eine neue Phase begonnen.

Im Gegensatz zur ersten Modifikation aus dem vorherigen Abschnitt terminiert diese Abwandlung sicher, was nun gezeigt wird:

Die Länge der Vorperiode wird mit c bezeichnet, die der Periode weiterhin mit z .

Spätestens, wenn $q \geq c + z$ gilt, ist β_q ein Element aus der Periode. In dieser Phase werden nun q weitere Elemente $k \in \{1, \dots, q\}$ berechnet, wobei im Fall $k = z$ für das erzeugte Element $\beta_{q+k} = \beta_{q+z} = \beta_q$ gilt und somit eine Kollision stattfindet.

Durch diese Abwandlung des Algorithmus kommt es zu einer konstanten Speicherkomplexität $\mathcal{O}(1)$. Anstelle sich wie bisher alle Folgenglieder β_i und die jeweils zugehörigen Exponenten w_i, v_i bis zum Auffinden einer Kollision merken zu müssen, genügt es nun, pro Phase das Tupel (β_q, w_q, v_q) zu speichern. Zusätzlich ist eine Speichereinheit nötig um das k -te Tupel $(\beta_{q+k}, w_{q+k}, v_{q+k})$, das in dieser Runde berechnet wurde, abzuspeichern, um es mit dem Phasentupel zu vergleichen. Gibt es mit dem k -ten Tupel keine Kollision, so wird der Speicherplatz für das $k + 1$ -te Element genutzt.

Bereits Pollard konnte durch Verwendung von Floyds Cycle-Finding-Algorithmus [4, vgl. S. 2] den Speicher auf einen konstanten Bedarf zurückregeln. Allerdings liefert Brents Ansatz zusätzlich eine verbesserte Laufzeit. Nach Bai et al. [4, vgl. S. 3] ist diese Variante allgemein 25% schneller als Pollards Variante. Jedoch ändert diese Optimierung nichts an der Laufzeitkomplexität des Algorithmus, da $\frac{3}{4} \cdot \mathcal{O}(\sqrt{n}) = \mathcal{O}(\sqrt{n})$ gilt.

4.2.5 Zusammenfassung des Algorithmus

Lege die Unterteilung für G fest, sodass $\beta_i \in G_i, i \in \{1, 2, 3\}$ bei den folgenden Berechnungen schnell bestimmt werden kann.

```

Definiere  $\beta_1 = 1$ ,  $w_1 = 0$  und  $v_1 = 0$ , sowie  $q = 1$ 
do
  Setze die Phasenelemente auf  $\beta_q$ ,  $w_q$  und  $v_q$ 
  for ( $k = 1$ ;  $k \leq q$ ;  $k++$ )
    Bestimme die Mengenzugehörigkeit (MZK) von  $\beta_{q+k-1}$ 
    Berechne  $\beta_i = \beta_{q+k}$ ,  $w_i = w_{q+k}$ ,  $v_i = v_{q+k}$  mittels  $f$  und der MZK
    if (Kollision gefunden, also  $\beta_i = \beta_q$ )
      Beende die innere Schleife
  Verdopple den Phasenwert:  $q = q * 2$ 
while (Es wurde keine Kollision gefunden)
if (Kollision nicht geeignet, also  $v_i == v_q$ )
  Starte von vorne mit  $v_1 = 0$ , zufälligem  $w_1$  und daran angepasstem  $\beta_1 = y^{w_1}$ 
Bestimme  $t = w_q - w_i$ ,  $s = v_i - v_q$  und  $ggT(s, n) = d$ 
Bestimme mit dem erweiterten euklidischen Algorithmus die Lösung für  $\hat{x}$ 


$$\frac{s}{d} \cdot \hat{x} \equiv \frac{t}{d} \pmod{\frac{n}{d}}$$


if (Kongruenz nicht lösbar)
  Terminiere ohne Lösung durch Fehler in der Angabe
for (int  $i = 0$ ;  $i < d$ ;  $i++$ )
  Berechne


$$x \equiv \hat{x} + \frac{in}{d} \pmod{n}$$


if ( $y^x = a$ )
  Terminiere mit  $x$  als Lösung.

```

4.2.6 Beispiel

Das Problem sei wiederum $DL_6(15) = x$ in $(\mathbb{Z}/41\mathbb{Z})^*$. $p = 41$, $y = 6$, $a = 15$
 Nach der Eulerschen φ -Funktion gilt: $\varphi(41) = 41-1 = 40 = n$ $n = 40$

Die Einteilung der Mengen G_1, G_2, G_3 wird bezüglich der Äquivalenzklassen von Modulo-3 vorgenommen. Um zu vermeiden, dass das neutrale Element 1 in der zweiten Menge liegt, wird die folgende Zuordnung vorgenommen.

$$\begin{aligned}
 G_1 &= \{x \in (\mathbb{Z}/41\mathbb{Z})^* : x \pmod{3} \equiv 2\} \\
 G_2 &= \{x \in (\mathbb{Z}/41\mathbb{Z})^* : x \pmod{3} \equiv 0\} \\
 G_3 &= \{x \in (\mathbb{Z}/41\mathbb{Z})^* : x \pmod{3} \equiv 1\}
 \end{aligned}$$

Jetzt wird mittels Brents Methodik nach Kollisionen gesucht.

Die initialen Werte sind mit $\beta_1 = 1$, $w_1 = 0$ und $v_1 = 0$ gewählt. Es gilt $\beta_1 \in G_3$.

Phase $q = 1$

Die Phasenelemente sind $\beta_q = \beta_1 = 1$, $w_q = w_1 = 0$ und $v_q = v_1 = 0$.

In jeder Phase werden in k Runden die neuen Elemente β_i berechnet. Da nach Definition $k \in \{1, \dots, q\}$ gilt, wird nur eine Runde mit $k = 1$ ausgeführt.

Folgenindex i	Menge	β_i	w_i	v_i	Runde k
2	G_2	15	0	1	1

Tabelle 4.1: Berechnungen der Phase $q = 1$

Da das neue Element β_2 nicht mit dem Phasenelement β_1 übereinstimmt, beginnt eine neue Phase mit $q = 2$.

Phase $q = 2$

Die Phasenelemente sind $\beta_q = \beta_2 = 15$, $w_q = w_2 = 0$ und $v_q = v_2 = 1$. Es werden mit $k \in \{1, 2\}$ zwei Runden ausgeführt.

Folgenindex i	Menge	β_i	w_i	v_i	Runde k
3	G_1	20	0	2	1
4	G_1	38	1	2	2

Tabelle 4.2: Berechnungen der Phase $q = 2$

Da keines der generierten Elemente mit dem Phasenelement β_2 übereinstimmt, beginnt eine neue Phase mit $q = 4$.

Phase $q = 4$

Die Phasenelemente sind $\beta_q = \beta_4 = 38$, $w_q = w_4 = 1$ und $v_q = v_4 = 2$. Es werden mit $k \in \{1, \dots, 4\}$ vier Runden ausgeführt.

Folgenindex i	Menge	β_i	w_i	v_i	Runde k
5	G_1	23	2	2	1
6	G_2	15	3	2	2
7	G_1	20	6	4	3
8	G_1	38	7	4	4

Tabelle 4.3: Berechnungen der Phase $q = 4$

Für das in Runde 4 generierte Element gilt $\beta_8 = \beta_4$. Somit gibt es eine Kollision mit dem Phasenelement und es gilt $t = w_4 - w_8 = 1 - 7 = -6$ sowie $s = v_8 - v_4 = 4 - 2 = 2$. Hinweis: Aufgrund der Kongruenz 4.4 könnten wir genauso auch mit $t = w_8 - w_4$ sowie $s = v_4 - v_8$ weiterrechnen. Nun kann mit Schritt 3 fortgefahren werden.

Da $s \neq 0$ gilt, wurde eine valide Kollision gefunden. Mit den bisher erhaltenen Werten bestimmen wir $d = \text{ggT}(s, n) = \text{ggT}(2, 40) = 2$. Anschließend berechnen wir die Lösung für

$$\frac{s}{d} \cdot \hat{x} \equiv \frac{t}{d} \pmod{\frac{n}{d}}$$

Setzt man die erhaltenen Werte ein, erhalten wir:

$$\frac{2}{2} \cdot \hat{x} \equiv \frac{-6}{2} \pmod{\frac{40}{2}} \Rightarrow 1 \cdot \hat{x} \equiv -3 \pmod{20} \Rightarrow \hat{x} \equiv 17 \pmod{20}$$

Hinweis: Wäre $\frac{s}{d} \neq 1$, müsste man diesen Wert bezüglich $\pmod{\frac{n}{d}}$ invertieren und die Gleichung mit diesem Inversen multiplizieren.

Schließlich gibt es nach Lemma 41 d -viele mögliche Lösungen, die nacheinander ausprobiert werden.

$$x \equiv \hat{x} + \frac{in}{d} \pmod{n}, \quad i = 0, 1, \dots, d-1$$

Einsetzen von $i = 0$ liefert:

$$x \equiv 17 + \frac{0 \cdot 40}{2} \pmod{40} \equiv 17$$

Da nun $6^{17} \pmod{41} = 26 \neq 15$ gilt, muss die zweite Lösung (mit $i = 1$) die richtige sein. Einsetzen von $i = 1$ liefert:

$$x \equiv 17 + \frac{1 \cdot 40}{2} \pmod{40} \equiv 37$$

Es gilt $6^{37} \pmod{41} \equiv 15$, womit das DLP gelöst ist.

Selbige Lösung erhalten wir auch durch das selbst erstellte Programm:

```

----- Pollards-Rho-Algorithmus -----

~~~~ Initialisierung ~~~~
Primzahl p:      41
Primitivwurzel y:  6
Potenz a:        15
Ordnung n:       40

~~~~ Neue Phase: q = 1 ~~~~
> Phasenwerte:1 0 0
15 0 1

~~~~ Neue Phase: q = 2 ~~~~
> Phasenwerte:15 0 1
20 0 2
38 1 2

~~~~ Neue Phase: q = 4 ~~~~
> Phasenwerte:38 1 2
23 2 2
15 3 2
20 6 4
38 7 4

Eine Kollision wurde gefunden!
38 1 2
38 7 4

Das Ergebnis ist:      37

```

Abbildung 4.3: Ausgabe für das DLP $6^x \equiv 15 \pmod{41}$ mit dem Rho-Algorithmus

4.3 Pohlig-Hellman-Algorithmus

Der nun folgende Algorithmus wurde nach seinen beiden Entwicklern Stephen Pohlig und Martin Hellman benannt[32]. Die folgende Darstellung orientiert sich an Buchmann [9, vgl. S. 223f] und Diekert et al. [11, vgl. S. 102f].

Dieses Verfahren ist kein eigenständiger Algorithmus zur Berechnung des diskreten Logarithmus. Vielmehr wird durch die Anwendung dieses Verfahrens für ein gegebenes DLP die Berechnung des diskreten Logarithmus unter Zuhilfenahme von z.B. Shanks Babystep-Giantstep-Algorithmus oder dem Pollard ρ -Algorithmus erheblich vereinfacht. Die Effizienzsteigerung ist umso größer, in je mehr Primfaktoren n zerlegt werden kann und je kleiner diese sind. Deswegen ist neben dem Wissen über $y \in G$ als erzeugendes Element, $a \in G$ als gegebene Potenz und über die Gruppenordnung $\text{ord}(G) = n$ insbesondere die Kenntnis der Primfaktorzerlegung von n wichtig.

Die Berechnung des DLPs $y^x = a$ mit $x \in \mathbb{Z}/n\mathbb{Z}$ kann in zwei großen Schritten vereinfacht werden:

4.3.1 Reduktion auf Gruppen mit Primzahlpotenzordnungen

Die Faktorisierung von n ist nützlich, da wir die Berechnung des ursprünglich einen Logarithmus in der gegebenen Gruppe mit Ordnung n auf mehrere Berechnungen in Gruppen von kleineren Primzahlpotenzordnungen aufteilen können [11, vgl. S. 102]. Unter Anwendung des chinesischen Restsatzes kann im Anschluss die Lösung bestimmt werden.

Satz 43 (Der chinesische Restsatz)

Mit dem chinesischen Restsatz lässt sich für simultane Kongruenzen (verschiedener Restklassen) eine gemeinsame Lösung berechnen [34, S. 194]:

Gegeben seien $k_1, k_2, \dots, k_i \in \mathbb{N}$, die paarweise teilerfremd sind. Dazu existieren die Werte $x_1 \in \mathbb{Z}/k_1\mathbb{Z}$, ..., $x_i \in \mathbb{Z}/k_i\mathbb{Z}$, mit denen sich ein System simultaner Kongruenzen aufstellen lässt:

$$x \equiv x_1 \pmod{k_1}, \dots, x \equiv x_i \pmod{k_i}$$

Sofern die Lösung x existiert, mit der alle obigen Kongruenzen erfüllt sind, ist diese eindeutig. x liegt dabei in der Restklasse $\mathbb{Z}/k\mathbb{Z}$, wobei $k = k_1 \cdot k_2 \cdot \dots \cdot k_i$ entspricht. Der konkrete Lösungsalgorithmus um ein solches x zu bestimmen, findet sich bei Reiss et al. [34, S. 194f] und auch der zugehörige Beweis.

Verfahren

Zunächst wird die Gruppenordnung n als Produkt von Primzahlen $\tau \in \mathbb{P}$ und ihren zugehörigen Exponenten $e(\tau) \in \mathbb{N}$ analog zu Bemerkung 18 geschrieben:

$$\text{ord}(G) = n = \prod_{\tau \in \mathbb{P}, \tau | n} \tau^{e(\tau)}$$

Nun fassen wir die Primfaktoren von n in einer Menge zusammen.

$$T = \{\tau \in \mathbb{P} : \tau \text{ teilt } n\}$$

Für jedes $\tau \in T$ definieren wir folgende Elemente:

$$n_\tau := \frac{n}{\tau^{e(\tau)}}, \quad y_\tau := y^{n_\tau}, \quad a_\tau := a^{n_\tau} \quad (4.6)$$

Für alle τ gilt nun: y_τ ist Erzeuger einer Untergruppe U_τ von G . y_τ bzw. U_τ haben die Ordnung $\tau^{e(\tau)}$ [23, vgl. S. 9]. Dies lässt sich aus folgender Umformung erkennen:

$$\begin{aligned} (y_\tau)^{\tau^{e(\tau)}} &= (y^{n_\tau})^{\tau^{e(\tau)}} \\ &= (y^{\frac{n}{\tau^{e(\tau)}}})^{\tau^{e(\tau)}} \\ &= y^{\frac{n}{\tau^{e(\tau)}} \cdot \tau^{e(\tau)}} \\ &= y^n \\ &= 1 \end{aligned} \quad (4.7)$$

Für die folgende Umformung wird ohne Einschränkung ein $\tau \in T$ festgelegt. Die Berechnungen erfolgen alle in der zugehörigen Untergruppe U_τ . Darin liegt nun auch das Element a_τ , was wie folgt gezeigt werden kann [25, vgl. S. 17]:

$$\begin{aligned}
a &= y^x \\
\Rightarrow a^{n_\tau} &= (y^x)^{n_\tau} \\
\Leftrightarrow a_\tau &= (y^x)^{n_\tau} \\
&= y^{n_\tau \cdot x} \\
&= (y^{n_\tau})^x \\
&= (y_\tau)^x
\end{aligned}$$

Somit sieht man, dass der diskrete Logarithmus $DLog_{y_\tau}(a_\tau) \equiv x \pmod{\tau}$ für alle τ existieren muss. Da wir mit dieser Berechnung aber nicht die eigentliche Lösung $x \pmod{n}$, sondern nur den Wert $x \pmod{\tau}$ erhalten, der das DLP in der von τ abhängigen Untergruppe löst, wird dieser im Folgenden mit x_τ bezeichnet.

Anschließend wird in allen Untergruppen $U_\tau, \tau \in T$ dieses zugehörige DLP gelöst. Das kann z.B. mit dem Babystep-Giantstep-Algorithmus (4.1) oder dem Pollard- ρ -Algorithmus (4.2) geschehen, weil diese auch in Untergruppen arbeiten.

Hier wird ersichtlich, dass anstelle einer einzigen DLP-Berechnung in einer Gruppe mit Ordnung n nun also mehrere DLPs in den kleineren Untergruppen U_τ mit Ordnung $\tau^{e(\tau)}$ zu bestimmen sind.

Dadurch werden die Lösungen $DLog_{y_\tau}(a_\tau) = x_\tau$ mit $x_\tau \in \{0, \dots, \tau^{e(\tau)} - 1\}$ geliefert, mit denen wir die simultanen Kongruenzen $x \equiv x_\tau \pmod{\tau^{e(\tau)}}$ aufstellen. Da alle $\tau^{e(\tau)}$ relativ prim zueinander sind, lässt sich nun mit dem chinesischen Restsatz ein Ergebnis x bestimmen, das die Lösung aller simultanen Kongruenzen ist. Das so bestimmte x entspricht auch der Lösung des diskreten Logarithmus $DLog_y(a) = x$.

Dass dies gilt, wird im Rest des Abschnitts gezeigt [23, vgl. S. 9]:

$$\begin{aligned}
(y_\tau)^{x_\tau} &= a_\tau \\
\Rightarrow 1 &= a_\tau \cdot (y_\tau)^{-x_\tau} \\
&= a^{n_\tau} \cdot (y^{n_\tau})^{-x_\tau} \\
&= a^{n_\tau} \cdot y^{n_\tau \cdot (-x_\tau)} \\
&= a^{n_\tau} \cdot (y^{-x_\tau})^{n_\tau} \\
&= (a \cdot y^{-x_\tau})^{n_\tau} & (4.8) \\
&= (a \cdot y^{-x})^{n_\tau} & (4.9)
\end{aligned}$$

Die Umformung von (4.8) zu (4.9) erfolgt, da $x \equiv x_\tau \pmod{\tau^{e(\tau)}}$ gilt.

In der Darstellung von Gleichung 4.9 erkennen wir, dass $a \cdot y^{-x}$ das erzeugende Element einer weiteren zyklischen Untergruppe Δ von G ist, denn es wird mit der Potenz n_τ auf das neutrale Element 1 abgebildet. Allerdings ist noch unbekannt, ob n_τ auch die Ordnung dieser neuen Untergruppe Δ ist, denn wir wissen nicht, ob n_τ der kleinste Exponent ist, für den das erzeugende Element $a \cdot y^{-x}$ den Wert 1 annimmt. Allerdings

ist damit bekannt, dass die Ordnung $o(a \cdot y^{-x})$ ein Teiler von n_τ ist. Weil diese Aussage für alle $\tau \in T$ gilt und $o(a \cdot y^{-x})$ somit alle n_τ teilen muss steht fest, dass $o(a \cdot y^{-x})$ auch den $ggT(n_\tau : \tau \in T)$ teilt [11, vgl. S. 102]. Nach der Definition in 4.6 ist $ggT(n_\tau : \tau \in T) = 1$. Deswegen muss $o(a \cdot y^{-x}) = 1$ gelten. Nach Satz 10 entspricht dies auch der Ordnung der von $a \cdot y^{-x}$ erzeugten Untergruppe Δ , und somit hat diese nur ein Element. Daraus formen wir um:

$$\begin{aligned} 1 &= (a \cdot y^{-x})^{n_\tau} \\ &= a \cdot y^{-x} \\ \Rightarrow y^x &= a \end{aligned}$$

Insgesamt folgt, dass das mithilfe des chinesischen Restsatzes gefundene x die simultanen Kongruenzen $x \equiv x_\tau \pmod{\tau^{e(\tau)}}$ löst und somit die Lösung des Problems $DLog_y(a) = x$ ist.

4.3.2 Laufzeitkomplexität und Speicherbedarf

Weil mit dem Pohlig-Hellman-Verfahren alleine kein diskreter Logarithmus berechnet werden kann, gibt es dafür keine eindeutige Komplexitätsanalyse. Diese erfolgt in Abhängigkeit von dem verwendeten Algorithmus zur Berechnung der DLPs.

Laufzeitkomplexität

Anstatt die Lösung über einen einzelnen diskreten Logarithmus zu bestimmen, werden in diesem Verfahren mehrere DLPs berechnet. Es wird für alle $\tau \in T$ der diskrete Logarithmus in der jeweiligen Untergruppe U_τ mit Ordnung $\tau^{e(\tau)}$ bestimmt. Dominierend ist somit die Berechnung des Logarithmus in der Untergruppe mit der größten Ordnung: $\max(ord(U_\tau)) = \max(\tau^{e(\tau)})$. Diese Berechnung erfolgt z.B. mit einem der beiden bereits vorgestellten Algorithmen. Diese besitzen beide die gleiche Laufzeitkomplexität von $\mathcal{O}(\sqrt{\max(\tau^{e(\tau)})})$.

Das entspricht auch der Gesamtkomplexität, da die Berechnung der simultanen Kongruenzen mittels des chinesischen Restsatzes insbesondere bei großen Primfaktoren zu vernachlässigen ist [23, vgl. S. 10]. Bei vielen kleinen Primfaktoren ist dieser Algorithmus laut Buchmann [9, vgl. S. 224] bereits wesentlich schneller als $\mathcal{O}(\sqrt{n})$, was in Kapitel 6 durch Laufzeittests praktisch überprüft wird.

Speicherkomplexität

Auch die zu speichernden Elemente sind (bei Umsetzung mit Shanks Algorithmus) von der Untergruppe mit der größten Ordnung abhängig, sodass auch hier die Komplexität $\mathcal{O}(\sqrt{\max(\tau^{e(\tau)})})$ vorliegt. Verwendet man hingegen Pollards Algorithmus so bleibt der Speicheraufwand bei $\mathcal{O}(1)$.

4.3.3 Reduktion auf Gruppen mit Primzahlzordnungen

Mit dem Pohlig-Hellman-Verfahren können wir sogar noch einen Schritt weiter gehen: Die Berechnung in einer Gruppe mit Primzahlpotenzordnung $\tau^{e(\tau)}$ lässt sich nämlich nochmals auf die Lösung mehrerer diskreter Logarithmen in einer Gruppe mit lediglich Primzahlordnung τ vereinfachen (siehe Abbildung 4.4). Dabei wird die Darstellung von x_τ in τ -adischer Schreibweise ausgenutzt [25, vgl. S. 18]. Mit den dabei gefundenen Werten kann wiederum der diskrete Logarithmus für die zugehörige Gruppe mit Primzahlpotenzordnung bestimmt werden. Dadurch erhalten wir, wie nach dem ersten Schritt, für alle Untergruppen simultane Kongruenzen, die anschließend, wie oben, mit dem chinesischen Restsatz gelöst werden.

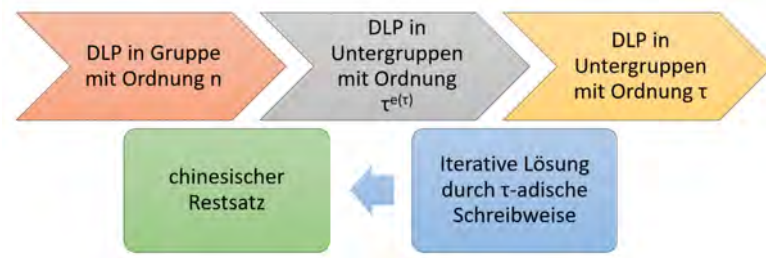


Abbildung 4.4: Der Ablauf des Pohlig-Hellman-Algorithmus

Verfahren

Zunächst betrachten wir wieder die oben definierten Untergruppen $U_\tau = \langle y_\tau \rangle$. Sie haben, wie bereits bekannt (siehe Formel 4.7), die Ordnung $\tau^{e(\tau)}$. Außerdem liegt die Lösung des zugehörigen diskreten Logarithmus in der Untergruppe: $DL_{y_\tau}(a_\tau) = x_\tau$ mit $x_\tau < \tau^{e(\tau)}$. Um im Folgenden Unübersichtlichkeiten durch lange Formeln zu vermeiden, werden Ersetzungen vorgenommen:

$$\xi := e(\tau), \quad \chi := x_\tau, \quad \alpha := a_\tau, \quad \mu := y_\tau$$

Da $\chi \in \{0, \dots, \tau^\xi - 1\}$, gibt es nun nach Diekert et al. [11, vgl. S. 102] eine eindeutige Schreibweise für χ in τ -adischer Darstellung:

$$\chi = \sum_{i=0}^{\xi-1} \chi_i \cdot \tau^i. \quad (4.10)$$

Das DLP in der Untergruppe U_τ kann mit den Umbenennungen wie folgt geschrieben und umgeformt werden:

$$\begin{aligned} a_\tau &= (y_\tau)^{x_\tau} \\ \Leftrightarrow \alpha &= \mu^\chi \\ \Rightarrow \alpha^{\tau^{\xi-1}} &= (\mu^\chi)^{\tau^{\xi-1}} \\ &= \mu^{\chi \cdot \tau^{\xi-1}} \end{aligned} \quad (4.11)$$

Im nächsten Schritt wird der Exponent $\chi \cdot \tau^{\xi-1}$ von μ genauer betrachtet. Die Darstellung von χ in τ -adischer Darstellung ermöglicht Folgendes:

$$\begin{aligned}
\chi \cdot \tau^{\xi-1} &= \sum_{i=0}^{\xi-1} \chi_i \cdot \tau^i \cdot \tau^{\xi-1} \\
&= \chi_0 \cdot \tau^{\xi-1} + \sum_{i=1}^{\xi-1} \chi_i \cdot \tau^{i+\xi-1} \\
&= \chi_0 \cdot \tau^{\xi-1} + \tau^{\xi} \cdot \sum_{i=1}^{\xi-1} \chi_i \cdot \tau^{i-1} \\
&= \chi_0 \cdot \tau^{\xi-1} + \tau^{\xi} \cdot (\chi_1 + \chi_2 \cdot \tau + \dots + \chi_{\xi-1} \cdot \tau^{\xi-2})
\end{aligned}$$

μ hat als erzeugendes Element der Gruppe U_{τ} die Ordnung τ^{ξ} . Mit dieser Kenntnis kann die Gleichung 4.11 weiter umgeschrieben werden ([23, vgl. S. 10], [25, vgl. S. 18]):

$$\begin{aligned}
\alpha^{\tau^{\xi-1}} &= \mu^{\chi \cdot \tau^{\xi-1}} \\
&= \mu^{(\chi_0 \cdot \tau^{\xi-1} + \tau^{\xi} \cdot (\chi_1 + \chi_2 \cdot \tau + \dots + \chi_{\xi-1} \cdot \tau^{\xi-2}))} \\
&= \mu^{\chi_0 \cdot \tau^{\xi-1}} \cdot \mu^{(\tau^{\xi} \cdot (\chi_1 + \chi_2 \cdot \tau + \dots + \chi_{\xi-1} \cdot \tau^{\xi-2}))} \\
&= \mu^{\chi_0 \cdot \tau^{\xi-1}} \cdot (\mu^{\tau^{\xi}})^{(\chi_1 + \chi_2 \cdot \tau + \dots + \chi_{\xi-1} \cdot \tau^{\xi-2})} \\
&= \mu^{\chi_0 \cdot \tau^{\xi-1}} \cdot (1)^{(\chi_1 + \chi_2 \cdot \tau + \dots + \chi_{\xi-1} \cdot \tau^{\xi-2})} \\
&= \mu^{\chi_0 \cdot \tau^{\xi-1}} \\
&= (\mu^{\tau^{\xi-1}})^{\chi_0}
\end{aligned}$$

Weil $(\mu^{\tau^{\xi-1}})^{\tau} = (\mu^{\tau^{\xi}})$ gilt, hat $\mu^{\tau^{\xi-1}}$ die Ordnung τ . χ_0 ist also die Lösung von $DLog_{(\mu^{\tau^{\xi-1}})}(a^{\tau^{\xi-1}})$ in einer Gruppe mit Ordnung τ . Diese Berechnung geschieht wieder mit einem der in dieser Arbeit erwähnten Verfahren (z.B. Babystep-Giantstep- oder Pollard- ρ -Algorithmus).

χ besteht allerdings aus einer Summe, zu deren Bestimmung noch weitere χ_i benötigt werden. Auch diese lassen sich mit der Kenntnis von χ_0 vorteilhafterweise genauso einfach in der Gruppe mit Ordnung τ berechnen. Bei dieser iterativen Vorgehensweise sind im i -ten Schritt die Werte $\chi_0, \dots, \chi_{i-1}$ bekannt:

$$\begin{aligned}
\alpha &= \mu^{\chi} \\
&= \mu^{(\chi_0 + \dots + \chi_{i-1} \cdot \tau^{i-1} + \chi_i \cdot \tau^i + \dots + \chi_{\xi-1} \cdot \tau^{\xi-1})} \\
&= \mu^{(\chi_0 + \dots + \chi_{i-1} \cdot \tau^{i-1})} \cdot \mu^{(\chi_i \cdot \tau^i + \dots + \chi_{\xi-1} \cdot \tau^{\xi-1})} \\
\Leftrightarrow \alpha \cdot \mu^{-(\chi_0 + \dots + \chi_{i-1} \cdot \tau^{i-1})} &= \mu^{(\chi_i \cdot \tau^i + \dots + \chi_{\xi-1} \cdot \tau^{\xi-1})} \tag{4.12}
\end{aligned}$$

Nun nehme ich aus Gründen der Übersicht wieder eine Umbenennung vor:

$$\alpha_i := \alpha \cdot \mu^{-(\chi_0 + \dots + \chi_{i-1} \cdot \tau^{i-1})} \tag{4.13}$$

Der initiale Wert α_0 wird dabei auf $\alpha_0 = \alpha \cdot \mu^{-0} = \alpha$ gesetzt.
Anschließend formen wir obige Gleichung (4.12) wie folgt um:

$$\begin{aligned}
\alpha_i &= \mu^{(\chi_i \cdot \tau^i + \dots + \chi_{\xi-1} \cdot \tau^{\xi-1})} \\
\Leftrightarrow (\alpha_i)^{\tau^{\xi-(i+1)}} &= (\mu^{(\chi_i \cdot \tau^i + \dots + \chi_{\xi-1} \cdot \tau^{\xi-1})})^{\tau^{\xi-(i+1)}} \\
&= (\mu^{(\chi_i \cdot \tau^i + \sum_{j=i+1}^{\xi-1} \chi_j \cdot \tau^j)})^{\tau^{\xi-(i+1)}} \\
&= \mu^{(\chi_i \cdot \tau^i + \sum_{j=i+1}^{\xi-1} \chi_j \cdot \tau^j) \cdot (\tau^{\xi-(i+1)})} \\
&= \mu^{(\chi_i \cdot \tau^i) \cdot \tau^{\xi-(i+1)}} \cdot \mu^{(\sum_{j=i+1}^{\xi-1} \chi_j \cdot \tau^j) \cdot (\tau^{\xi-(i+1)})} \\
&= \mu^{\chi_i \cdot \tau^{\xi-1}} \cdot \mu^{(\sum_{j=i+1}^{\xi-1} \chi_j \cdot \tau^j) \cdot (\tau^{\xi-(i+1)})} \tag{4.14}
\end{aligned}$$

Zur besseren Lesbarkeit wird der Exponent des zweiten Faktors gesondert betrachtet:

$$\left(\sum_{j=i+1}^{\xi-1} \chi_j \cdot \tau^j \right) \cdot (\tau^{\xi-(i+1)}) = \left(\sum_{j=i+1}^{\xi-1} \chi_j \cdot \tau^{j-(i+1)} \right) \cdot \tau^{\xi} \tag{4.15}$$

Nun setzen wir die Umformung des Exponenten aus (4.15) in (4.14) ein. Wichtig ist dabei, dass die Umformung weiterhin in der Gruppe U_τ erfolgt [25, vgl. S. 19]:

$$\begin{aligned}
(\alpha_i)^{\tau^{\xi-(i+1)}} &= \mu^{\chi_i \cdot \tau^{\xi-1}} \cdot \mu^{(\sum_{j=i+1}^{\xi-1} \chi_j \cdot \tau^j) \cdot (\tau^{\xi-(i+1)})} \\
&= \mu^{\chi_i \cdot \tau^{\xi-1}} \cdot \mu^{(\tau^{\xi}) \cdot (\sum_{j=i+1}^{\xi-1} \chi_j \cdot \tau^{j-(i+1)})} \\
&= \mu^{\chi_i \cdot \tau^{\xi-1}} \cdot (\mu^{\tau^{\xi}})^{(\sum_{j=i+1}^{\xi-1} \chi_j \cdot \tau^{j-(i+1)})} \\
&= \mu^{\chi_i \cdot \tau^{\xi-1}} \cdot 1 \\
&= (\mu^{\tau^{\xi-1}})^{\chi_i} \tag{4.16}
\end{aligned}$$

Wie bereits oben erwähnt, entspricht diese Berechnung einem DLP in einer Gruppe mit Ordnung τ .

Die Frage ist nun, wie wir nach den eben dargestellten Beweisen und Umformungen die für die DLPs nötigen Elemente (also Basis und Potenz) konkret bestimmen können: Das erzeugende Element der Untergruppe auf der rechten Seite der Gleichung 4.16 ist unabhängig von den Iterationen i . Es gibt also für alle DLPs bezüglich einer Primzahl nur eine Basis, die wir wie folgt erhalten:

$$\mu^{\tau^{\xi-1}} = (y_\tau)^{\tau^{e(\tau)-1}} = (y^{n_\tau})^{\tau^{e(\tau)-1}} = (y^{\frac{n}{\tau^{e(\tau)}}})^{\tau^{e(\tau)-1}} = y^{\frac{n}{\tau}}$$

Die Potenz (auf der linken Seite der Gleichung 4.16) ist jedoch von der jeweiligen Iteration i abhängig. Zur Bestimmung der α_i muss jedoch nicht immer die ganze Berechnung 4.13

durchgeführt werden, denn wir können α_i , wie im Folgenden gezeigt, iterativ erhalten. Dazu sind aus dem vorherigen Schritt χ_{i-1} und $\alpha_{i-1} := \alpha \cdot \mu^{-(\chi_0 + \dots + \chi_{i-2} \cdot \tau^{i-2})}$ bekannt.

$$\begin{aligned}\alpha_i &= \alpha \cdot \mu^{-(\chi_0 + \dots + \chi_{i-2} \cdot \tau^{i-2} + \chi_{i-1} \cdot \tau^{i-1})} \\ &= (\alpha \cdot \mu^{-(\chi_0 + \dots + \chi_{i-2} \cdot \tau^{i-2})}) \cdot \mu^{-(\chi_{i-1} \cdot \tau^{i-1})} \\ &= \alpha_{i-1} \cdot \mu^{-(\chi_{i-1} \cdot \tau^{i-1})}\end{aligned}$$

Insgesamt ergibt sich mit dem Ergebnis 4.16 und den gerade durchgeführten Umformungen also:

$$\begin{aligned}(\alpha_i)^{\tau^{\xi-(i+1)}} &= (\mu^{\tau^{\xi-1}})^{\chi_i} \\ \Leftrightarrow (\alpha_{i-1} \cdot \mu^{-(\chi_{i-1} \cdot \tau^{i-1})})^{\tau^{\xi-(i+1)}} &= (y^{\frac{n}{\tau}})^{\chi_i} \\ \Leftrightarrow (\alpha_{i-1} \cdot (y^{\frac{n}{\tau^{e(\tau)}}})^{-(\chi_{i-1} \cdot \tau^{i-1})})^{\tau^{\xi-(i+1)}} &= (y^{\frac{n}{\tau}})^{\chi_i}\end{aligned}$$

Für ein $\tau \in T$ müssen somit $\xi = e(\tau)$ viele diskrete Logarithmen in Gruppen der Ordnung τ gelöst werden. Aus den jeweiligen Ergebnissen χ_i lässt sich mit der Formel 4.10 anschließend $\chi = x_\tau$ errechnen.

Dieses Verfahren muss für alle $\tau \in T$ vollzogen werden, um die simultanen Kongruenzen $x \equiv x_\tau \pmod{\tau^{e(\tau)}}$ aufzustellen, welche schließlich mit dem chinesischen Restsatz gelöst werden können.

4.3.4 Optimierte Komplexitätseigenschaften

Laufzeitkomplexität

Für jedes τ , das Teiler von n ist, müssen $e(\tau)$ viele Ziffern χ_i berechnet werden.

$$\sum_{\tau|n} e(\tau)$$

Für jeden Teiler $\tau \in T$ müssen $e(\tau)$ viele DLPs gelöst werden. Jede Bestimmung von χ_i benötigt wegen der Darstellung bzw. Berechnung in τ -adischer Form $\mathcal{O}(\log(n))$ Schritte [19, vgl. S. 2]. Wenn für die Berechnung der diskreten Logarithmen in den Gruppen mit Ordnung τ einer der beiden bisher vorgestellten Algorithmen angewandt wird, werden dafür $\mathcal{O}(\sqrt{\tau})$ Operationen benötigt [9, vgl. S. 227]. Insgesamt ergibt sich nach Diekert et al. [11, vgl. S. 103] die folgende Komplexität:

$$\mathcal{O}\left(\sum_{\tau|n} e(\tau) \cdot (\log(n) + \sqrt{\tau})\right)$$

Wenn der Aufwand für das Lösen der simultanen Kongruenzen mit dem chinesischen Restsatz wieder vernachlässigt wird, entspricht dies dem Gesamtaufwand.

Wir können erkennen, dass der dominierende Term $\sqrt{\tau}$, vom größten Primfaktor der Gruppenordnung $\tau \in T$ abhängt. Verallgemeinernd gilt somit:

$$\sum_{\tau|n} e(\tau) \cdot (\log(n) + \sqrt{\tau}) = \mathcal{O}(\sqrt{\max(\tau)})$$

Zusammenfassend lässt sich nun feststellen, dass der Pohlig-Hellman-Algorithmus besonders effizient arbeitet, wenn die Gruppenordnung n viele, möglichst kleine Primfaktoren besitzt. Dies ist jedoch zugleich auch eine große Schwäche, da eine solche Faktorisierung insbesondere bei großen Zahlen nur sehr schwer zu finden ist [39, vgl. S. 1].

Speicherkomplexität

Auch hier gibt es keine eindeutige Analyse, da die Umsetzung z.B. mit Shanks oder Pollards Algorithmus erfolgen kann. Im Vergleich zum ersten Reduktionsschritt müssen nun noch mehr DLPs berechnet werden, wobei dies in Gruppen mit noch kleineren Ordnungen geschieht. Bei sequentieller Abarbeitung gilt für Pollards Algorithmus, wieder eine Speicherkomplexität von $\mathcal{O}(1)$.

Shanks Algorithmus kann wie folgt analysiert werden: Es kann maximal $\log_2(n)$ viele Primfaktoren von n geben, da 2 die kleinste Primzahl ist. Nach [35, vgl. S. 2] ist für Logarithmen die gewählte Basis für die \mathcal{O} -Notation irrelevant, da Basiswechsel vorgenommen werden können. Also gibt es $\mathcal{O}(\log(n))$ viele Primfaktoren. Für jeden Primfaktor τ müssen fünf Werte $(p, e(\tau), n_\tau, y_\tau$ und $a_\tau)$ abgespeichert werden.

Bei Berechnung mit dem Babystep-Giantstep-Algorithmus ist es, wie bereits bekannt, nötig, für die Berechnung in Gruppen der jeweiligen Ordnung τ , $\mathcal{O}(\sqrt{\max(p)})$ viele Elemente abzuspeichern. Damit ergibt sich als maximale Komplexität:

$$\mathcal{O}(\log(n) \cdot (5 + \sqrt{\max(p)})) = \mathcal{O}(\log(n) \cdot \sqrt{\max(p)})$$

4.3.5 Zusammenfassung des Algorithmus

Bilde die Menge $T = \{\tau \in \mathbb{P} : \tau \text{ teilt } n\}$

Schreibe die Gruppenordnung n als Produkt dieser Primzahlen

$$\text{ord}(G) = n = \prod_{\tau \in T} \tau^{e(\tau)}$$

Bestimme für jedes $\tau \in T$ folgende Elemente

$$n_\tau = \frac{n}{\tau^{e(\tau)}}, \quad y_\tau = y^{n_\tau}, \quad a_\tau = a^{n_\tau}$$

Berechne $DL_{y_\tau}(a_\tau) = x_\tau$ mit $x_\tau \leq \tau^{e(\tau)}$ in allen Untergruppen U_τ .

- Berechne diesen entweder direkt in der Untergruppe U_τ oder
- Reduziere auf Gruppen mit Primzahlordnung für einen Faktor τ

Setze $\mu = y_\tau$ und $\alpha_0 = a_\tau$.

Berechne den Erzeuger der Untergruppe $y_{DLP} = y^{\frac{n}{\tau}}$.

for($i = 0$; $i < e(\tau)$; $i++$)

Bestimme $a_{DLP} \equiv \alpha_i^{\tau^{e(\tau)-1-i}} \pmod{p}$
 Löse $DL_{y_{DLP}}(a_{DLP}) = \chi_i$ in einer Untergruppe mit Ordnung τ .
 Berechne $\alpha_{i+1} = \alpha_i \cdot \mu^{-(\chi_i \cdot \tau^i)}$

Bestimme über die τ -adische Darstellung den Wert von $\chi = \sum_{i=0}^{e(\tau)-1} \chi_i \cdot \tau^i$.
 Erhalte somit die simultane Kongruenz $x \equiv \chi \pmod{\tau^{e(\tau)}}$ für die Untergruppe U_τ .

Löse die simultanen Kongruenzen $x \equiv x_\tau \pmod{\tau^{e(\tau)}}$ mit dem chinesischen Restsatz.

4.3.6 Beispiel

Das Problem sei $DL_6(15) = x$ in $(\mathbb{Z}/41\mathbb{Z})^*$. $p = 41, y = 6, a = 15$
 Nach der Eulerschen φ -Funktion gilt: $\varphi(41) = 41 - 1 = 40 = n$ $n = 40$
 Die Primfaktorisierung von n ist $n = 2^3 \cdot 5^1$.

Teiler $\tau = 2$

Bestimme $n_\tau = \frac{40}{8} = 5$, $y_\tau \equiv 6^5 \equiv 27 \pmod{41}$ und $a_\tau \equiv 15^5 \equiv 14 \pmod{41}$.
 Es gilt nun also das DLP $27^x \equiv 14 \pmod{41}$ zu lösen, was mit der vollständigen Reduktion erfolgt.
 Setze dazu $\mu = y_\tau = 27$ und $\alpha_0 = a_\tau = 14$.
 Außerdem berechnen wir: $y_{DLP} \equiv y^{\frac{n}{\tau}} \equiv 6^{\frac{40}{2}} \equiv 40 \pmod{41}$.

$i = 0$

$a_{DLP} \equiv \alpha_0^{\tau^{e(\tau)-1-0}} \equiv 14^4 \equiv 40 \pmod{41}$
 Das zugehörige DLP $40^{\chi_0} = 40$ kann mit $\chi_0 \equiv 1 \pmod{41}$ gelöst werden.
 Aktualisiere zuletzt $\alpha_1 \equiv \alpha_0 \cdot \mu^{-(\chi_0 \cdot \tau^0)} \equiv 14 \cdot 27^{-1} \equiv 14 \cdot 38 \equiv 40 \pmod{41}$

$i = 1$

$a_{DLP} \equiv \alpha_1^{\tau^{e(\tau)-1-1}} \equiv 40^2 \equiv 1 \pmod{41}$
 Das zugehörige DLP $40^{\chi_1} = 1$ kann mit $\chi_1 \equiv 0 \pmod{41}$ gelöst werden.
 Aktualisiere zuletzt $\alpha_2 \equiv \alpha_1 \cdot \mu^{-(\chi_1 \cdot \tau)} \equiv 40 \cdot 27^0 \equiv 40 \pmod{41}$

$i = 2$

$a_{DLP} \equiv \alpha_2^{\tau^{e(\tau)-1-2}} \equiv 40^1 \equiv 40 \pmod{41}$
 Das zugehörige DLP $40^{\chi_2} = 40$ kann mit $\chi_2 \equiv 1 \pmod{41}$ gelöst werden.
 Da dies der letzte Schritt für $\tau = 2$ ist, kann die Berechnung von α_3 entfallen.

Schließlich bestimmen wir die Lösung für die Kongruenz über die Darstellung in τ -adischer Form: $\chi = \chi_0 + \chi_1 \cdot \tau + \chi_2 \cdot \tau^2 = 1 + 0 \cdot 2 + 1 \cdot 2^2 = 5$.
 Damit lässt sich die Kongruenz $x \equiv 5 \pmod{2^3}$ aufstellen.

Teiler $\tau = 5$

Das gleiche Schema wird nun auch für den Teiler 5 vollzogen.

Bestimme $n_\tau = \frac{40}{5} = 8$, $y_\tau \equiv 6^8 \equiv 10 \pmod{41}$ und $a_\tau \equiv 15^8 \equiv 18 \pmod{41}$.

Es gilt nun also das DLP $10^x \equiv 18 \pmod{41}$ zu lösen.

Setze dazu $\mu = y_\tau = 10$ und $\alpha_0 = a_\tau = 18$.

Außerdem berechnen wir: $y_{DLP} \equiv y^{\frac{n}{\tau}} \equiv 6^{\frac{40}{5}} \equiv 10 \pmod{41}$.

$i = 0$

$$a_{DLP} \equiv \alpha_0^{\tau^{e(\tau)-1}-0} \equiv 18^1 \equiv 18 \pmod{41}$$

Das zugehörige DLP $10^{x_0} \equiv 18 \pmod{41}$ kann mit $\chi_0 \equiv 2 \pmod{41}$ gelöst werden.

Da dies der letzte Schritt für $\tau = 5$ ist, kann die Berechnung von α_1 entfallen.

Schließlich erhalten wir die Lösung für die Kongruenz über die Darstellung in τ -adischer Form: $\chi = \chi_0 = 2$.

Damit lässt sich die Kongruenz $x \equiv 2 \pmod{5^1}$ aufstellen.

Lösung der simultanen Kongruenzen mit dem chinesischen Restsatz

Für die beiden erhaltenen simultanen Kongruenzen wird abschließend die gemeinsame Lösung x ermittelt, welche auch das DLP löst. Die Lösung für $x \equiv 5 \pmod{8}$, $x \equiv 2 \pmod{5}$ ist $x = 37$.

Das gleiche Ergebnis kann auch mit dem implementierten Programm erhalten werden:

```
----- Pohlig-Hellman-Algorithmus (vollständige Reduktion auf Primzahlordnung) -----
~~~ Initialisierung ~~~
Primzahl p:      41
Primitivwurzel y: 6
Potenz a:        15
Ordnung n:       40

Die übergebenen Werte sind eine korrekte Primfaktorisierung der Ordnung.

~~~ Teiler: 2 ~~~
Löse das Untergruppen-DLP  $27^x = 14 \pmod{41}$  mit der vollständigen Pohlig-Hellman Reduktion.

- Die Lösung des DLP  $40^x = 40 \pmod{41}$  in der Untergruppe mit Ordnung 2 lautet: 1
- Die Lösung des DLP  $40^x = 1 \pmod{41}$  in der Untergruppe mit Ordnung 2 lautet: 0
- Die Lösung des DLP  $40^x = 40 \pmod{41}$  in der Untergruppe mit Ordnung 2 lautet: 1
Damit ergibt sich die Kongruenz:  $x = 5 \pmod{8}$ 

~~~ Teiler: 5 ~~~
Löse das Untergruppen-DLP  $10^x = 18 \pmod{41}$  mit der vollständigen Pohlig-Hellman Reduktion.

- Die Lösung des DLP  $10^x = 18 \pmod{41}$  in der Untergruppe mit Ordnung 5 lautet: 2
Damit ergibt sich die Kongruenz:  $x = 2 \pmod{5}$ 

Löse die simultanen Kongruenzen mit dem chinesischen Restsatz.

Das Ergebnis ist:      37
```

Abbildung 4.5: Ausgabe für das DLP $6^x \equiv 15 \pmod{41}$ mit dem Pohlig-Hellman-Algorithmus

5 Implementierung

Um die Funktionsweise der drei Algorithmen in der Praxis bestätigen zu können, implementierte ich diese als Konsolenprogramm. Dabei sollte das Ziel sein, DLPs mit den drei unterschiedlichen Algorithmen lösen zu können. Außerdem war es mir wichtig, den Speicher- bzw. Leistungsbedarf zu analysieren, und diesen ggf. durch kleine Änderungen im Code zu verbessern.

5.1 Entwicklungsumgebung

5.1.1 Software

Damit die Software auf möglichst vielen Rechnern einfach ausgeführt werden kann, entschied ich mich dazu das Programm mit Java zu entwickeln. Dazu verwendete ich zunächst Java 8 in der 64bit Variante. Während der Implementierung stieg ich aus Effizienzgründen auf Java 9 64bit um, mit dem letztendlich auch das Programm kompiliert wurde. Der Vorteil der 64bit Version ist, dass sie theoretisch bis zu maximal 16 Exabytes ($16 * 10^6$ Terabyte) Arbeitsspeicher ausnutzen kann [1], während die 32bit Variante auf maximale 4GB limitiert ist [30]. Diese Entscheidung war, wie sich im Laufe der Entwicklung herausgestellt hat, auch sinnvoll, da Shanks Algorithmus in den Testbeispielen bis zu 14 GB an Speicherplatz benötigte (siehe Abschnitt 6.2.2).

Auf meinem Rechner, der mit Windows 10 Pro 64bit läuft, nutzte ich als IDE JetBrains IntelliJ IDEA, welches auch für die Versionsverwaltung (Git) zum Einsatz kam. Zusätzlich verwendete ich zu diesem Zweck das Tool GitKraken der Firma Axosoft. Als Test-Framework diente mir das seit September 2017 veröffentlichte JUnit 5. Die Auslastung des RAM-Speichers beobachtete ich mit dem Programm JProfiler.

Für einige mathematische Berechnungen, wie z.B. Primfaktorisationen, das Berechnen von diskreten Exponentationen und das Finden von Primzahlen oder Primitivwurzeln, griff ich auf WolframAlpha zurück.

5.1.2 Hardware

Die Implementierung erfolgte vollständig auf meinem Rechner, der folgende Spezifikationen besitzt: Intel i7-4700MQ, 4x 2,4 GHz, 16GB DDR3 RAM

5.2 Bericht zum Entwicklungsprozess

Im Folgenden erläutere ich mein ursprünglich geplantes Klassendesign, die zugehörigen Funktionen und welche Änderungen daran jeweils während des Entwicklungsprozesses

vorgenommen wurden.

In meinen Vorüberlegungen ging ich zunächst von vier Klassen aus: Einer IO-Klasse „Shell“ und jeweils einer Klasse für die Berechnung der Algorithmen.

5.2.1 Shell

In der Shell-Klasse (ähnlich zur Vorlesung „Programmierung 2“) werden die Ein- und Ausgaben verarbeitet. Dazu gehört die Interpretation der Benutzerbefehle, die durch Buchstaben aufgerufen werden, wobei die Groß- und Kleinschreibung vernachlässigt wird.

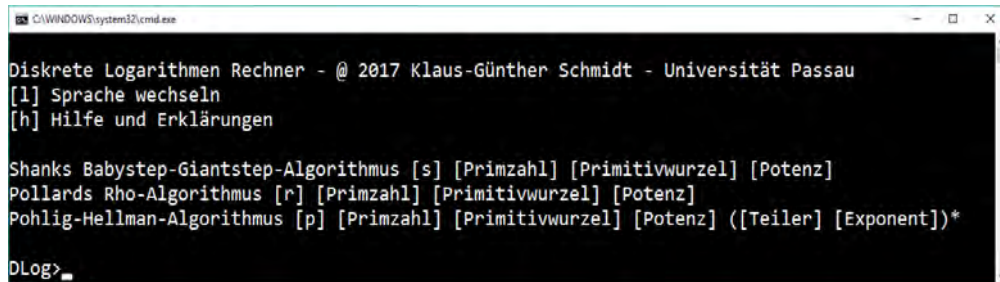


Abbildung 5.1: Startbildschirm „Diskrete Logarithmen Berechner“

Beim Aufruf der Algorithmen erfolgt in der Shell bereits eine Überprüfung der übergebenen Parameter. U. a. wird kontrolliert, ob die Anzahl der Argumente für den gewählten Algorithmus korrekt ist oder die Parameter fehlerfrei in Zahlen umgewandelt werden können. Dazu gibt das Programm geeignete Meldungen aus und vermeidet somit unnötigen Rechenaufwand und Fehler.

```
/**
 * Checks if all the inputs are correct.
 *
 * @param tokens The user inputs that are checked for correctness.
 * @return If no error occurs return the to BigInteger parsed tokens[]. Else null.
 */
private static BigInteger[] checkInput(String tokens[], DLPTYPE typ) {
    int numberarguments;
    //Check correct number arguments
    if (typ != DLPTYPE.POHLIG_HELLMAN) {
        numberarguments = 3;
        if (tokens.length != numberarguments + 1) {
            System.out.println(messages.getString(key: "warning_arguments_invalid"));
            return null;
        }
    }
}
```

Abbildung 5.2: Überprüfung auf Parameterfehler

Außerdem übernimmt die Shell auch die Benutzerführung im Programm und zeigt Hinweise zu den eingegebenen Befehlen und dem Ergebnis einer DLP-Rechnung an. Neben den Befehlen zum Starten der drei Algorithmen gibt es in der Shell die Möglichkeit

das Programm zu beenden, sich die Hilfe [h] anzusehen und einen erweiterten Ausgabe-Modus [v] zu starten. Mit letzterem ist es möglich, Informationen zu Zwischenschritten in der Rechnung zu erhalten. Dieser Modus ist insbesondere zum Debuggen und zum Nachvollziehen der Funktionsweise sinnvoll, erfordert aber, dass neben der Shell auch die Algorithmen-Klassen Ausgaben anzeigen können. Im Verlauf der Entwicklung zeigte sich, dass eine große Anzahl solcher Ausgaben den Programmfluss enorm verlangsamen kann. Deswegen sollte dieser Modus nur zu Verständniszwecken bei beispielhaften, mit der Hand nachrechenbaren Problemen verwendet werden. Ein Exempel einer solchen Ausgabe findet sich bei der Beispiel-Aufgabe zu Shanks BSGS in Abbildung 4.1.

Als erste Änderung baute ich während der Implementierung einen Modus ein, mit dem es möglich ist, sich die zu einer DLP-Berechnung gehörenden Ausgaben in einer Text-Datei abspeichern zu lassen [f]. Somit bleiben gewonnene Daten auch nach dem Beenden des Programms erhalten und das Programm erkennt, ob die gleiche Berechnung schon einmal getätigt wurde. Konkret realisierte ich dies mit `TreeOutputStream` und einer eigenen Klasse „`LogWriter`“, womit die Ausgaben auf die Streams für die Datei und Konsole aufgespalten werden.

Da wie später beschrieben, für den Pohlig-Hellman- und Pollards Rho-Algorithmus jeweils zwei verschiedene Varianten implementiert wurden, war es nötig in der Shell weitere Unterscheidungsmöglichkeiten für die Eingaben hinzuzufügen. Dazu hob ich die Vernachlässigung von Groß- und Kleinschreibung für die beiden Algorithmen auf, sodass zwischen den zwei Modi unterschieden werden kann.

```
[r] [Primzahl] [Primitivwurzel] [Potenz] berechnet mit Pollards Rho-Algorithmus (und einem festen Startwert) einen diskreten Logarithmus.
z.B. r 37 2 11
[R] [Primzahl] [Primitivwurzel] [Potenz] verwendet ebenso Pollards Rho-Algorithmus, nutzt dabei aber einen zufälligen Startwert.
z.B. R 37 2 11
```

Abbildung 5.3: Zwei Varianten von Pollards Rho-Algorithmus

Als weitere Modifikation erachtete ich es als sinnvoll eine Benachrichtigung zu erhalten, wenn eine Berechnung abgeschlossen worden ist. Dazu wurde eine weitere, über die Shell einstellbare Option implementiert, die - sofern aktiviert - einen Hinweiston abspielt. Falls möglich, wird zusätzlich eine visuelle Tray-Benachrichtigung angezeigt.

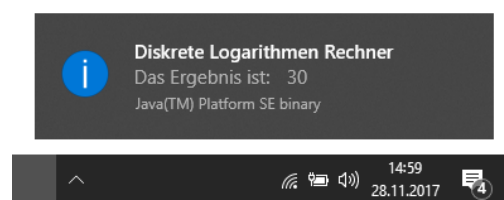


Abbildung 5.4: Benachrichtigung

Als letztes zu erwähnendes Feature fügte ich die Unterstützung für Skripte hinzu. Dadurch können automatisierte Skripte (z.B. batch Dateien) mit mehreren zu lösenden DLPs von der Kommandozeile sequentiell abgearbeitet werden. Das war besonders bei der Durchführung der Tests von großem Vorteil, da der Rechner somit unbeaufsichtigt

über längeren Zeitraum unterschiedlichste DLPs abarbeiten konnte.

```
java -jar DiscreteLogarithms.jar R 2112017 3 20001
java -jar DiscreteLogarithms.jar s 982451653 11 358519430
java -jar DiscreteLogarithms.jar p 2252943079 6 1234
```

Abbildung 5.5: Skript für automatisierte Berechnungen

5.2.2 Zusätzliche Klassen

Um die Laufzeit der Algorithmen genauer einschätzen zu können, wurde die Klasse „TimeMeasurement“ zur Zeitmessung eingeführt. Sie erlaubt es, die Zeit zwischen Aufruf und Beendung eines Algorithmus zu messen.

Die nächste eingeführte Modifikation war eine zusätzliche Klasse „Locale“, in die ich anfangs aus Gründen der Übersichtlichkeit sämtliche Strings für Hilfetexte, Hinweise und Berechnungen verschob. Um weitere Evolutionsmöglichkeiten offenzuhalten, gab ich dieses Design jedoch wieder auf und ersetzte es durch eine Lokalisationskomponente. Dadurch ist es nun möglich das Programm in weitere Sprachen zu übersetzen und für einen größeren Benutzerkreis zugänglich zu machen. Deswegen fügte ich auch die Sprache Englisch hinzu und setzte sie als Standardsprache. Über den Befehl [1] ist es möglich in der Shell zwischen den beiden Sprachen zu wechseln.

5.2.3 Algorithmen

Bevor ich auf die genaue Entwicklung der jeweiligen Algorithmen eingehe, finden sich hier noch ein paar allgemeine Bemerkungen:

Die Umsetzung der Algorithmen in Java erfolgte parallel zur Erarbeitung der jeweiligen Beweise. Jede Algorithmenimplementierung wurde zunächst auf Funktionalität und Korrektheit getestet, um Fehler auszubessern, und schließlich an die Shell angeschlossen. Im Verlauf der Umsetzung schrieb ich dazu 17 allgemeine Tests zur Lösung von DLPs verschiedener Schwierigkeit. Zusätzlich gibt es für jeden Algorithmus noch spezielle Tests um gewisse Implementierungseigenheiten zu überprüfen.

Bereits bei der Implementierung des zweiten Algorithmus fielen mir einige Gemeinsamkeiten bezüglich der verwendeten Methodenköpfe und Variablen auf. Dies hat mich veranlasst eine abstrakte Klasse „DLPsolver“ einzuführen, um gemeinsame Komponenten an die konkreten Algorithmen zu vererben. Dadurch war es mir möglich redundanten Code zu deduplizieren. Außerdem integrierte ich dort eine Methode, die triviale DLPs der Form $y^x \equiv 1 \pmod{p} \Rightarrow x \equiv 0$ oder $y^x \equiv y \pmod{p} \Rightarrow x \equiv 1$ erkennt. Somit können sie mit konstantem Aufwand gelöst werden und müssen nicht an die rechenaufwendigen Algorithmen weitergereicht werden. Die Einführung dieser abstrakten Klasse bietet eine weitere Evolutionsmöglichkeit, da nun mit geringem Aufwand weitere Algorithmen zum Lösen von DLPs hinzugefügt werden können.

Zunächst ging ich davon aus, dass der in Java größte primitive Datentyp für ganze Zahlen „long“ eine ausreichende Länge hat, um damit bereits aufwendiger zu berechnende Probleme kreieren zu können. Seit Java 8 kann ein unsigned long-Datentyp Werte zwischen 0 und $2^{64} - 1$ annehmen [29]. Für die benötigten Algorithmen wie Potenzen mit Modulo, den ggT oder das multiplikative Inverse schrieb und testete ich selbst Methoden, da diese für long oder den zugehörigen kapselnden Datentypen Long nicht verfügbar sind. Für diese mathematisch grundlegenden Algorithmen erstellte ich die neue Klasse „Basics“. Als es dann zur Implementierung kam, stellte ich fest, dass die Berechnungen ohne Überlauf sehr schnell (in wenigen ms) zu lösen waren, während komplexere DLPs in größeren Einheitengruppen zu Überläufen geführt hatten. Aus diesem Grund wechselte ich vom Datentyp long zum Typ BigInteger. Zwar werden damit Schleifen aufwendiger, doch kann dieser Datentyp Werte zwischen $-2^{\text{Integer.MAX_VALUE}} + 1$ und $+2^{\text{Integer.MAX_VALUE}} - 1$ enthalten [28]. Da $\text{Integer.MAX_VALUE} = 2^{31} - 1$ gilt, ist BigInteger damit besonders für Berechnungen mit großen Werten gedacht. Auf BigInteger gibt es Standardimplementierungen für Potenzen mit Modulo, den ggT und das multiplikative Inverse, sodass sich die Neuimplementierung dafür erübrigte. Die „Basics“-Klasse konnte ich aber für eine Implementierung der Quadratwurzel und des chinesischen Restsatzes mit BigInteger benutzen.

Shanks Babystep-Giantstep-Algorithmus

Die Implementierung verlief nach dem Umstieg auf BigInteger problemlos. Für die Speicherung der Babysteps verwendete ich eine HashMap, da diese unsynchronisiert ist und somit bei nur einem Thread performanter agiert als ein Hashtable [31]. Allerdings hat eine Hashmap den Nachteil, dass sie nicht sortiert ist. Deswegen fügte ich für den erweiterten Ausgabemodus eine Methode hinzu, die die Hashmap nach der zweiten Komponente (also in der Reihenfolge der Erzeugung ihrer Elemente) sortiert. Die anschließenden Tests mit kleineren Primzahlen (bis zum Bereich 10^4) fielen positiv aus und liefen überraschend schnell. Allerdings kam das Programm bei größeren Werten erheblich ins Stocken. Bei einer Analyse mit JProfiler konnte ich erkennen, dass die Größe der JVM Heap (bei 64bit JRE Versionen) standardmäßig auf 4GB begrenzt ist. Mehr Informationen zur Java Heap und dem Speichermanagement befinden sich im Abschnitt 6.2.1.

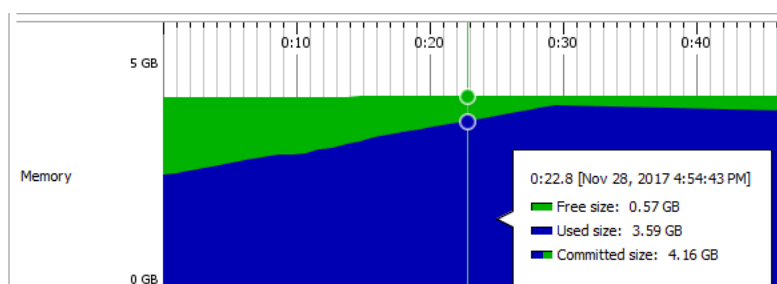


Abbildung 5.6: Begrenzter Heapspace (Screenshot JProfiler)

Mit der Option `-Xmx [Filesize]` kann aber, bei dementsprechend viel freiem RAM, mehr Speicher für die JVM Heap alloziiert werden. Beispielsweise kann mit

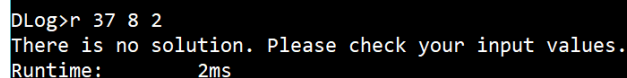
```
java -jar -Xmx8g [File].jar
```

einem Java-Prozess eine maximale Heap-Größe von 8GB zugewiesen werden. Nachdem diese Parameter im Skript bzw. in IntelliJ angepasst wurden, konnte ich auch rechenaufwendigere Probleme mit der erwarteten Performanz ausführen.

Pollards Rho-Algorithmus

Auch die Implementierung dieses Algorithmus lief zunächst problemlos.

Um dem auf Seite 24 geschilderten Fall ($(v_l - v_k) = 0$) eine Lösung zu bieten, wurde das Finden einer Kongruenz mit einer for-Schleife umgeben, die maximal 100 mal läuft bzw. bis passende Werte gefunden wurden. Beim ersten Durchlauf verwendet w_1 den Startwert 1, sofern weitere Schleifendurchläufe folgen, wird für w_i ein Zufallswert verwendet. Die for-Schleife begrenzte ich auf maximal 100 Durchläufe, da es, wie bereits erwähnt, sehr unwahrscheinlich ist, dass ein solcher Fall auftritt. Wird innerhalb der 100 Versuche keine geeignete Kongruenz gefunden, so handelt es sich also höchstwahrscheinlich nicht um ein korrekt gestelltes DLP.



```
DLog>r 37 8 2
There is no solution. Please check your input values.
Runtime: 2ms
```

Abbildung 5.7: Pollards Algorithmus findet keine Lösung für fehlerhaftes DLP

Pollards Rho-Methode ist, sofern sie mit w_1 im ersten Schleifendurchlauf passende Kongruenzen findet, reproduzierbarer Art, sodass es uns möglich ist, die Laufzeit auf verschiedenen Rechnern zu vergleichen. Diese Methode ist damit aber nicht geeignet, den erwarteten Rechenaufwand von $\mathcal{O}(\sqrt{n})$ zu überprüfen. Aus diesem Grund ergänzte ich noch einen zweiten Modus für Pollards Rho-Algorithmus, der mit zufälligen Startwerten arbeitet. Dies wird auch im Abschnitt 6.1.2 betrachtet.

Pohlig-Hellman Algorithmus

Im Gegensatz zu den beiden vorherigen Algorithmen gab es bei der Umsetzung von Pohlig-Hellman mehr Probleme.

Obwohl sich bei diesem Algorithmus eine Effizienzsteigerung in Bezug auf die Laufzeitkomplexität einstellen sollte, dauerte die Berechnung eines DLPs mit Pohlig-Hellman zunächst länger als mit den beiden anderen Algorithmen. Den Fehler konnte ich nach kurzer Zeit ausmachen: Da durch die Reduktion mehrere DLPs in Untergruppen gelöst werden müssen, haben die Untergruppen verständlicherweise eine andere Ordnung als die Primitivwurzel. Meine bisherigen Konstruktoren von Shanks BSGS und Pollards Rho-Algorithmus hatten aber immer die Ordnung der Primitivwurzel übergeben bekommen, sodass sich die Berechnung nicht wirklich vereinfachte. Nach der Behebung dieses

Problems durch einen neuen Konstruktor geschah die Berechnung verblüffend schnell.

Nun eröffnete sich jedoch ein weiteres Problem in Bezug auf die Untergruppen. Zwar sind, wie bereits erwähnt, die beiden anderen in dieser Arbeit behandelten Algorithmen geeignet, DLPs in Untergruppen zu berechnen, jedoch traten mit dem Rho-Algorithmus immer wieder Fälle auf, die nicht gelöst werden konnten. Bei sehr kleinen Untergruppen-Ordnungen (<50) kann es nämlich aufgrund der Einteilung (mod 3) geschehen, dass in ungünstigen Konstellationen immer der Fall $((v_l - v_k) = 0)$ eintritt.

Um dieses Problem zu beheben, modifizierte ich die Berechnung der DLPs im Pohlig-Hellman-Algorithmus dahingehend, dass Probleme mit Ordnungen, in denen Shanks Algorithmus akzeptabel viel RAM verbraucht, mit ebendiesem Algorithmus gelöst werden und Probleme mit größeren Ordnungen mit Pollards Algorithmus. Wie viel RAM-Verbrauch angemessen ist, hängt von der RAM-Größe des Systems ab. Ich wählte in der Implementierung die Grenze bei einer Ordnung von 10^{14} , da Shanks Algorithmus in einem solchen Fall ca. 2GB RAM verbraucht. Da somit nur Untergruppen-DLPs mit Ordnungen größer 10^{14} mit Pollards Rho-Methode bearbeitet werden, ist es nahezu ausgeschlossen, dass das DLP nicht von Pollard gelöst werden kann. Sollte dies jedoch trotzdem der Fall sein, wird durch Shanks Algorithmus eine Absicherung angeboten.

```
private BigInteger improvedSolver(BigInteger generator, BigInteger exp, BigInteger order) {
    BigInteger solution;
    //Use Shank's algorithm if the order is smaller than 10^14
    if (order.compareTo(new BigInteger("100000000000000")) == -1) {
        solution = new Shanks(PRIME_P, generator, exp, order).solve(verbose: false);
    } else {
        //use Pollard's algorithm
        solution = new Pollard(PRIME_P, generator, exp, order).solve(verbose: false);
        //If Pollard's algorithm found no solution use Shank's algorithm as backup
        if (solution == null) {
            solution = new Shanks(PRIME_P, generator, exp, order).solve(verbose: false);
        }
    }
    return solution;
}
```

Abbildung 5.8: Optimierter Pohlig-Hellman-Algorithmus

Auch beim Pohlig-Hellman Algorithmus gibt es zwei verschiedene Modi: Während der eine die vollständige Reduktion auf Primzahlordnung ausnutzt, verwendet der andere nur die Reduktion auf Primzahlpotenzordnung.

5.3 Statistiken

Insgesamt besteht das Projekt aus 2094 Zeilen, die sich wie folgt aufgliedern:

Klasse	Zeilen						
	Gesamt	Quellcode	Quellcode %	Kommentare	Kommentare %	Leerzeilen	Leerzeilen %
Basics.java	73	35	48%	31	42%	7	10%
DLPsolver.java	111	42	38%	56	50%	13	12%
LogWriter.java	75	40	53%	27	36%	8	11%
PohligHellman.java	292	143	49%	97	33%	52	18%
Pollard.java	306	161	53%	104	34%	41	13%
Shanks.java	177	94	53%	64	36%	19	11%
Shell.java	428	238	56%	133	31%	57	13%
TimeMeasurement.java	37	11	30%	17	46%	9	24%
Teilsomme:	1499	764	51%	529	35%	206	14%
BasicsTest.java	131	102	78%	6	5%	23	18%
PohligHellmanTest.java	191	144	75%	17	9%	30	16%
PollardTest.java	134	94	70%	16	12%	24	18%
ShanksTest.java	139	96	69%	16	12%	27	19%
Teilsomme:	595	436	73%	55	9%	104	17%
Gesamt:	2094	1200	57%	584	28%	310	15%

Abbildung 5.9: statistische Daten - ausgewertet durch IntelliJ

Der hohe Kommentaranteil kommt daher, dass ich den Code sehr ausführlich dokumentierte, um es Interessierten einfacher zu machen, den Code und die Arbeitsweise der Algorithmen nachzuvollziehen. Die Kommentare wurden vollständig in Englisch verfasst, um sie einem möglichst großen Publikum zur Verfügung stellen zu können.

5.4 Zusammenfassung

Das Programm unterstützt in seinem jetzigen Zustand folgende Befehle:

h zeigt die Hilfe.

q beendet das Programm.

l wechselt die Sprache zwischen Englisch und Deutsch.

v startet oder beendet den erweiterten Anzeige-Modus.

f startet oder beendet die Dateiausgabe.

n startet oder beendet die visuelle/ akustische Benachrichtigung nach der Berechnung eines DLP.

s [Primzahl] [Primitivwurzel] [Potenz] berechnet mit Shanks Babystep-Giantstep Algorithmus einen diskreten Logarithmus: z.B. „s 37 2 11“.

r [Primzahl] [Primitivwurzel] [Potenz] berechnet mit Pollards Rho-Algorithmus (und einem festen Startwert) einen diskreten Logarithmus: z.B. „r 37 2 11“.

R [Primzahl] [Primitivwurzel] [Potenz] verwendet ebenso Pollards Rho-Algorithmus, nutzt dabei aber einen zufälligen Startwert: z.B. „R 37 2 11“.

- p [Primzahl] [Primitivwurzel] [Potenz] ([Teiler] [Exponent])* berechnet mit der vollständigen Pohlig-Hellman Reduktion einen diskreten Logarithmus. Dabei muss die Primfaktorzerlegung der Ordnung angegeben werden: z.B. „p 37 2 11 2 2 3 2“.
- P [Primzahl] [Primitivwurzel] [Potenz] ([Teiler] [Exponent])* berechnet nur mit der ersten Pohlig-Hellman Reduktion einen diskreten Logarithmus. Dabei muss die Primfaktorzerlegung der Ordnung angegeben werden: z.B. „p 37 2 11 2 2 3 2“.

Ein lauffähiges Version des Programm, sowie der Quellcode und die Unit-Tests befinden sich auf der beigelegten CD.

6 Analyse der Testdaten

In diesem Kapitel werden die erhobenen Testdaten in Bezug auf reale Laufzeiten und tatsächlichen Speicherbedarf analysiert. Dazu wurden 91 Testfälle erstellt, von denen im Folgenden 16 ausgewählt betrachtet werden. Die zugehörigen Informationen finden sich auf der nächsten Seite. Die Fälle wurden auf fünf Rechnern durchgeführt, deren Daten nun folgen:

1. Windows 10 Pro 64bit, Intel i7-4700MQ @ 4x 2,4 GHz, 16GB DDR3 RAM
2. Windows 8 Pro 64bit, Intel i7-3770 @ 4x 3,4 GHz, 16GB DDR3 RAM
3. Windows 7 Professional 64bit, Intel i7-4790 @ 4x 3,6 GHz, 16GB DDR3 RAM
4. Windows 10 Pro 64bit, Intel Core2 Duo E8500 @ 2x 3,16 GHz, 8GB DDR2 RAM
5. ArchLinux 4.13.12 64bit, AMD Ryzen 7 1700X @ 8x 3.4GHz, 32GB DDR4 RAM

6.1 Laufzeitanalyse

Um Aussagen über die Programmevolution treffen zu können, führte ich auf meinem eigenen Rechner (PC1) eine Längsschnittanalyse ausgewählter Tests durch. Dabei wurde die Laufzeit der Testfälle der ersten funktionierenden Variante (Version 1) mit der Laufzeit bei der finalen Implementierung (Version 2) verglichen. Des Weiteren wurde eine Querschnittsanalyse durchgeführt, die die Rechenzeit der einzelnen Fälle mit dem finalen Programm auf mehreren PCs testet. Dadurch können Informationen über Systemabhängigkeiten herausgefunden bzw. die allgemeine Effizienz der Implementierung begutachtet werden.

Die Laufzeiten sind im Folgenden in Millisekunden angegeben.

6.1.1 Längsschnittanalyse

Auf der nächsten Seite sind 12 ausgewählte DLPs zu sehen, die den Testfällen zu Grunde liegen. Sie wurden je nach ihrer Ordnung in drei Kategorien eingeteilt. In der anschließenden Abbildung 6.1 ist die Entwicklung der Laufzeiten der Testfälle mit verschiedenen Algorithmen und deren Varianten dargestellt. Die Berechnungen wurden ausschließlich auf PC1 durchgeführt.

Hinweis: Der Testfall #4groß konnte mit Shanks Algorithmus nicht gelöst werden, da dieser mehr als 14GB Arbeitsspeicher benötigt (siehe auch Abschnitt 6.2.2).

Einfache DLPs:

$$\#1_{\text{klein}} : 3^x \equiv 5 \pmod{7} \quad \#2_{\text{klein}} : 6^x \equiv 7 \pmod{13}, \quad \#3_{\text{klein}} : 2^x \equiv 11 \pmod{37}, \quad \#4_{\text{klein}} : 2^x \equiv 3 \pmod{701}$$

Mittlere DLPs:

$$\begin{aligned} \#1_{\text{mittel}} : 3^x &\equiv 12147021287 \pmod{32416190071}, & \#2_{\text{mittel}} : 3^x &\equiv 41060412595 \pmod{99999999977}, \\ \#3_{\text{mittel}} : 2^x &\equiv 683790757227 \pmod{999999999989}, & \#4_{\text{mittel}} : 11^x &\equiv 8180801819537 \pmod{10000000020001} \end{aligned}$$

Große DLPs:

$$\begin{aligned} \#1_{\text{groß}} : 3^x &\equiv 15885234851091 \pmod{20000000020001}, & \#2_{\text{groß}} : 19^x &\equiv 16004283708240 \pmod{100000000000031}, \\ \#3_{\text{groß}} : 3^x &\equiv 107589536716291 \pmod{361000000000049}, & \#4_{\text{groß}} : 37^x &\equiv 177917621779460413 \pmod{2305843009213693951} \end{aligned}$$

	Kleine DLPs																Gesamtzeit
	PohligHellman (1)				PohligHellman (2)				Shanks				Pollard Rho				
Test #	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	
Version 1	33,00	33,00	30,25	34,75	31,50	31,00	33,25	31,25	0,75	1,00	0,25	0,75	0,25	0,50	0,50	0,25	262,25
Version 2	1,77	2,54	3,69	0,54	2,23	0,54	0,31	0,46	0,38	0,15	0,10	0,10	0,15	0,10	0,10	0,10	13,27
Speedup	18,65	13,00	8,19	64,54	14,12	57,57	108,06	67,71	1,95	6,50	2,50	7,50	1,62	5,00	5,00	2,50	

	Mittlere DLPs																Gesamtzeit
	PohligHellman (1)				PohligHellman (2)				Shanks				Pollard Rho				
Test #	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	
Version 1	55,00	67,75	47,25	42,00	58,00	79,75	55,75	47,50	307,75	637,75	2145,00	6943,50	347,25	900,25	1385,50	9496,50	22616,50
Version 2	15,50	31,25	7,25	8,00	16,00	31,33	12,00	3,75	191,25	332,25	968,75	3847,75	312,50	644,50	949,00	5645,00	13016,08
Speedup	3,55	2,17	6,52	5,25	3,63	2,55	4,65	12,67	1,61	1,92	2,21	1,80	1,11	1,40	1,46	1,68	

	Große DLPs																Gesamtzeit
	PohligHellman (1)				PohligHellman (2)				Shanks				Pollard Rho				
Test #	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	
Version 1	16,00	35,00	939,00	11,00	18,00	33,00	918,00	14,00	7334,00	23360,00	59187,00	MEMORY	10822,00	14522,00	18161,00	5368079	5503449,00
Version 2	5,33	15,50	924,33	7,83	10,33	23,33	890,00	8,00	6398,50	21743,50	51455,50	MEMORY	10331,00	13828,50	18014,33	5073064	5196720,00
Speedup	3,00	2,26	1,02	1,40	1,74	1,41	1,03	1,75	1,15	1,07	1,15		1,05	1,05	1,01	1,06	

Abbildung 6.1: Entwicklung der Laufzeit (in ms) bei ausgewählten Testfällen

Wie in Abbildung 6.1 zu sehen, konnten beim Langzeittest sowohl große als auch kleine Effizienzsteigerungen erzielt werden. Dabei war das Maximum eine 108,06-fache Leistungsverbesserung (Testfall #3klein mit „Pohlig-Hellman (2)“ = vollständige Pohlig-Hellman-Reduktion) und das Minimum eine 1,01-fache Geschwindigkeitssteigerung (Testfall #3groß mit Pollards Rho-Algorithmus). Hauptgründe für die Leistungssteigerungen konnten durch feingranulare Tests ausgemacht werden. Dazu zählen unter anderem der Umstieg von Java 8 auf Java 9 und das Hinzufügen der Überprüfung, ob ein DLP mit einem trivialen Exponent gelöst werden kann (siehe Abschnitt 5.2.3). Außerdem gab es den bereits erwähnten Fehler in der Implementierung des Pohlig-Hellman-Algorithmus, der einem auf Primzahl(potenz)ordnung reduziertem DLP nicht die korrekte Ordnung zuwies. Zusätzlich überarbeitete ich die Algorithmen, sodass durch die Verringerung von Redundanzen, die Wiederverwendung von bereits berechneten Werten oder die Entfernung unnötiger Bedingungen die Berechnungen effizienter ablaufen. Insgesamt ergibt sich für die abgebildeten Testfälle somit ein durchschnittlicher Speedup von $\sim 9,78$.

6.1.2 Querschnittanalyse

Für die Querschnittsanalyse nutzte ich fünf verschiedene Testrechner, auf denen die 12 Testfälle mit vier der Algorithmen bzw. ihren Varianten ausgeführt wurden, sodass sich insgesamt 240 Testszenarien ergaben. Da Pollards Rho-Methode mit dem zufälligen Startwert [R] (siehe Kapitel 5.2.3) stets unterschiedliche Berechnungen ausführt und deswegen keinen reproduzierbaren Laufzeitaufwand besitzt, gehe ich auf Seite 58f. gesondert auf diesen Modus ein.

Auf der folgenden Seite sind in Abbildung 6.2 die Laufzeiten aller Testszenarien zu finden. Um für die Tests repräsentative Daten zu erhalten, wurde jedes Testszenario auf den einzelnen Endgeräten mindestens viermal ausgeführt. Dazu nutzte ich die finale Version meiner Implementierung.

Im Folgenden sind drei Aspekte vermerkt, die beim Betrachten der Tabelle auffällig sind:

Bei den kleinen DLPs können wir erkennen, dass beide Varianten des Pohlig-Hellman-Algorithmus nahezu immer länger brauchen als Shanks oder Pollards Algorithmus. Eine Erklärung dafür ist, dass bei Pohlig-Hellman das eine zu berechnende DLP in mehrere kleine zerlegt wird. Für diese DLP müssen z.B. zunächst erst die jeweiligen Basiswerte und Potenzen bestimmt werden. Dies kann gerade bei kleinen multiplikativen Gruppen zu einem zusätzlichen Aufwand führen, der nicht gerechtfertigt ist.

Auf den ersten Blick verwundert möglicherweise auch, dass die vollständig reduzierte Pohlig-Hellman Variante (2) nicht wesentlich schneller als Pohlig-Hellman (1), sondern teilweise sogar deutlich langsamer ist. Auch dies kann durch die zusätzlichen DLPs begründet werden, die Pohlig-Hellman (2) im Vergleich zu Pohlig-Hellman (1) aufstellen und ausrechnen muss. Die Benutzung von Pohlig-Hellman (2) macht also erst Sinn, wenn

Algorithmus		Test	PC1	PC2	PC3	PC4	PC5
Kleine DLPs	Pohlig- Hellman (1)	#1	1,77	3,75	0,00	7,50	2,00
		#2	2,54	3,75	3,75	0,00	2,00
		#3	3,69	7,50	4,00	11,75	2,00
		#4	0,54	0,00	3,75	4,00	3,00
	Pohlig- Hellman (2)	#1	2,23	0,00	0,00	0,00	2,25
		#2	0,54	0,00	0,00	4,00	2,00
		#3	0,31	4,00	4,00	3,75	2,50
		#4	0,46	7,75	0,00	11,75	3,00
	Shanks BSGS	#1	0,38	0,00	0,00	0,00	0,25
		#2	0,15	0,25	0,00	0,00	0,50
		#3	0,10	0,00	0,00	0,00	0,50
		#4	0,10	0,00	0,00	0,00	1,00
	Pollard - Rho	#1	0,15	0,00	7,75	0,00	0,50
		#2	0,10	0,00	0,00	0,00	0,50
		#3	0,10	0,00	0,00	8,00	0,75
		#4	0,10	4,00	0,00	3,75	1,00
Mittlere DLPs	Pohlig- Hellman (1)	#1	15,50	30,75	23,50	38,75	14,25
		#2	31,25	46,00	43,00	74,00	26,00
		#3	7,25	15,50	16,00	27,25	10,75
		#4	8,00	15,50	3,75	12,00	8,00
	Pohlig- Hellman (2)	#1	16,00	30,75	19,25	43,25	14,50
		#2	31,33	50,25	42,75	74,00	29,50
		#3	12,00	15,50	7,75	27,25	12,00
		#4	3,75	15,00	4,00	15,75	10,25
	Shanks BSGS	#1	191,25	154,75	144,00	491,25	172,50
		#2	332,25	282,00	276,75	791,25	276,50
		#3	968,75	819,00	819,25	2652,00	764,50
		#4	3847,75	3924,75	3888,25	14172,75	2487,25
	Pollard - Rho	#1	312,50	243,25	241,75	425,00	237,25
		#2	644,50	529,50	546,25	955,75	485,75
		#3	949,00	757,00	744,75	1329,75	668,75
		#4	5645,00	4674,00	4605,50	8579,75	4093,75
Große DLPs	Pohlig- Hellman (1)	#1	5,33	15,00	7,75	47,00	9,25
		#2	15,50	31,00	19,50	77,50	20,75
		#3	924,33	827,00	788,50	1490,00	717,25
		#4	7,83	16,00	13,00	23,50	9,50
	Pohlig- Hellman (2)	#1	10,33	16,00	7,75	24,00	10,75
		#2	23,33	31,00	19,50	47,00	22,75
		#3	890,00	834,00	770,00	1287,00	702,00
		#4	8,00	15,50	12,50	16,00	11,00
	Shanks BSGS	#1	6398,50	5210,00	5090,50	15857,50	3343,00
		#2	21743,50	17245,50	17537,50	57361,50	7801,00
		#3	51455,50	40863,00	41437,50	198596,00	17213,00
		#4	NOT ENOUGH MEMORY				
	Pollard - Rho	#1	10331,00	8993,00	8545,50	16810,00	7650,25
		#2	13828,50	12284,00	11991,00	22948,00	10800,75
		#3	18014,33	14975,50	15588,00	26793,00	13408,75
		#4	5073064,00	4733996,00	4599906,50	8342130,00	3878206,00

Abbildung 6.2: Vergleich der Laufzeit (in ms) verschiedener Endsysteeme bei ausgewählten Testfällen

$\mathcal{O}(\sqrt{\max(\tau)})$ so viel kleiner als $\mathcal{O}(\sqrt{\max(\tau^{e(\tau)})})$ ist, dass dies die Rechendauer deutlich beeinflusst. Das ist insbesondere dann der Fall, wenn der Exponent von τ sehr groß und $\max(\tau^{e(\tau)})$ somit deutlich größer als $\max(\tau)$ ist.

Ein Beispiel, bei dem sich der Aufwand der vollständigen Pohlig-Hellman-Reduktion lohnt, ist die Berechnung des DLPs von

$$11^x \equiv 1234 \pmod{74217034874881}$$

Die Ordnung des Modulwerts wird wie folgt faktorisiert:

$$74217034874881 = 2^{39} * 3^3 * 5^1$$

Pohlig-Hellman (2) muss dabei 39 DLPs der Ordnung 2 sowie drei DLPs der Ordnung 3 und eines der Ordnung 5 berechnen. Pohlig-Hellman (1) hingegen muss an sich nur drei DLPs berechnen. Wie oben ist ein DLP der Ordnung 5 zu bestimmen. Außerdem muss ein DLP der Ordnung 2^{39} und ein DLP der Ordnung 3^3 gelöst werden. Da diese DLPs nun mit Shanks oder Pollards Algorithmus berechnet werden und deren Laufzeitkomplexität von der Ordnung der DLPs abhängt, braucht Pohlig-Hellman (1) länger als Pohlig-Hellman (2). Dies ist auch in den sechs Testläufen der folgenden Abbildung zu erkennen:

Testlauf Nr.	1	2	3	4	5	6	Durchschnitt
Pohlig-Hellman (1)	688	942	744	849	606	886	785,83
Pohlig-Hellman (2)	1	3	1	5	5	2	2,83

Abbildung 6.3: Vergleich der Laufzeit (in ms) des obigen DLPs

Der dritte bemerkenswerte Aspekt ist, dass PC5 gerade bei sehr geringen Rechenzeiten mehrmals länger braucht als die anderen Rechner mit bis zu 9 Jahre alten Prozessoren, obwohl PC5 bezüglich seiner Spezifikation eindeutig der leistungstärkste ist. Eine Ursache dieser Auffälligkeit könnte sein, dass dies durch das Linux-Betriebssystem verursacht wird. Dennoch kann PC5 gerade bei den großen DLPs mit langer Rechenzeit (also bei #1-#4 groß mit Shanks und Pollards Algorithmus) seine Leistung demonstrieren. Diesbezüglich fällt auch auf, dass PC5 gerade bei den drei großen gelösten DLPs mit Shanks-Algorithmus deutlich schneller ist als die anderen Testgeräte. Das könnte an den verbauten 32GB DDR4 Arbeitsspeicher liegen.

Im Hinblick auf Pollards Rho-Algorithmus implementierte ich, wie bereits erwähnt, neben dem „reproduzierbaren“ Modus [r], der immer mit dem Startwert $\beta_0 = 1$ beginnt, eine weitere Variante [R], die mit einem zufälligen β_0 startet. Die erwartete Laufzeit beträgt nach Abschnitt 4.2.2 $\mathcal{O}(\sqrt{n})$, kann jedoch auch davon abweichen. Durch die Verwendung des Zufallsmodus ist es nun möglich, mehr über die variierenden Laufzeiten des Rho-Algorithmus herauszufinden, da sich die Berechnung eines DLP für jeden Startwert anders verhält.

Die Testergebnisse (siehe Abbildung 6.4) stammen bei dieser Analyse nur von einem

Rechner (PC1), wodurch sie vergleichbar bleiben und somit größtmöglich nur vom gewählten Startwert abhängen.

Da die Abweichungen bei den kleinen DLPs mit Messungen im Millisekundenbereich nur sehr gering ausfallen, ließ ich diese Testfälle hier weg und gebe nur die Laufzeit der mittleren sowie großen DLPs an, da diese dort deutlicher schwanken. Dabei stehen #1-#4 jeweils wieder für die bereits oben definierten DLPs.

	Test	Ausführung				Mittelwert	Standard-abweichung
		1.	2.	3.	4.		
Mittlere DLPs	#1	297	297	375	281	312,50	36,67
	#2	281	594	406	610	472,75	136,71
	#3	907	250	1282	344	695,75	421,53
	#4	5407	5406	6844	5359	5754,00	629,61
Große DLPs	#1	10125	2828	10219	2985	6539,25	3633,33
	#2	14765	14297	15172	14875	14777,25	314,71
	#3	10562	17532	23547	11422	15765,75	5234,85
	#4	5249752	5225505	5118833	5642694	5309196,00	198744,03

Abbildung 6.4: Laufzeitvarianz (in ms) bei Pollards Rho-Algorithmus mit PC1

In der obigen Abbildung ist gut zu erkennen, dass die Standardabweichung umso stärker wird, je größer die Ordnung des zugehörigen DLPs ist. Diese Variabilität ist auf eine höhere bzw. niedrigere Anzahl an nötigen Berechnungen zur Kollisionsfindung zurückzuführen. Das bedeutet auch, dass die Stärke der Abweichungen durch die Rechenleistung hoch- bzw. herunterskaliert wird.

Zum Vergleich mit PC1 sind in der folgenden Abbildung die Laufzeiten der gleichen Testfälle mit dem langsameren PC4 angegeben, bei dem die errechnete Standardabweichung nun auch größer ausfällt.

	Test	Ausführung				Mittelwert	Standard-abweichung
		1.	2.	3.	4.		
Mittlere DLPs	#1	421	405	490	367	420,75	44,53
	#2	748	359	561	537	551,25	137,81
	#3	812	2561	1388	1535	1574,00	630,66
	#4	9095	8780	7542	7846	8315,75	640,72
Große DLPs	#1	6506	16473	11922	15700	12650,25	3943,24
	#2	22323	33181	32470	29654	29407,00	4297,36
	#3	53633	40018	36847	52745	45810,75	7469,54
	#4	8599278	8299278	8619278	8929278	8611778,00	222864,87

Abbildung 6.5: Laufzeitvarianz (in ms) bei Pollards Rho-Algorithmus mit PC4

6.2 Speicheranalyse

Zuletzt möchte ich noch auf den in meiner Implementierung benötigten Speicher der Algorithmen eingehen.

6.2.1 Management des JVM Heap

Der Arbeitsspeicher eines Java-Programms wird „JVM Heap“ genannt. In Analyse-Tools wie JProfiler werden bezüglich des Speicherplatzes zwei Werte angezeigt. Der erste Wert „used size“, der in den Abbildungen blau eingefärbt ist, stellt die tatsächliche Größe der JVM dar. In diesem benutzten Speicher „used memory“ sind nicht nur aktuelle benötigte Variablen und Objekte enthalten, sondern unter Umständen auch veraltete, auf die im Programm nicht mehr referenziert wird.

Um Platz zu sparen, werden die überflüssigen Speicherrückstände bei Bedarf vom Garbage Collector (GC) aus dem Speicher entfernt. Dies kann an dem Abfallen des „used memory“ erkannt werden. Der zweite Wert „committed size“ (grün) gibt die Größe des Speichers an, den die JVM für sich reserviert hat, bzw. die ihr vom RAM-Management zugestanden wurde. Der „committed memory“ enthält den ganzen „used memory“, sowie einigen freien Speicher als Puffer. Der Wert des „committed memory“ wird von der JVM dynamisch, je nach Bedarf alloziiert und verändert sich während der Ausführung eines Java-Programms. Wenn der „used memory“ immer weiter ansteigt und sich dem Wert des „committed memory“ nähert, wird dieser weiter erhöht (siehe Abbildung rechts), um sicher zu stellen, dass auch nachfolgende Daten noch im Arbeitsspeicher gelagert werden können. Allerdings gibt es eine maximale Größe des „committed memory“ und damit auch für den „used memory“.

Sie beträgt, wie bereits angesprochen, bei Java 64bit Programmen standardmäßig 4GB. Man kann diesen maximalen Wert aber auch manuell mit dem Parameter `-Xmx [Filesize]` (siehe Abschnitt 5.2.3) höher setzen.

Für die Analyse des Speicherbedarfs ist nur der Wert des „used memory“ nötig, da dieser den tatsächlich benutzten Speicherplatz angibt.

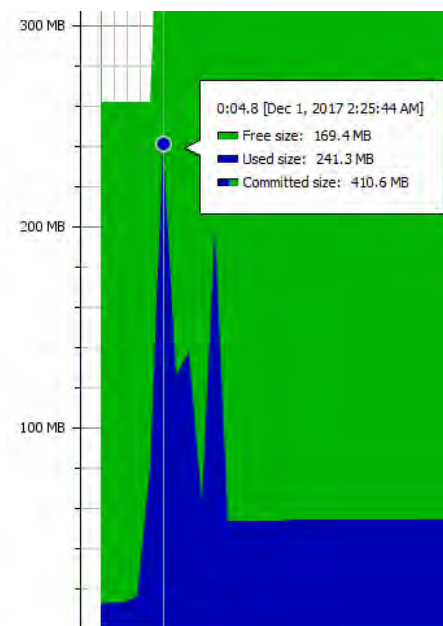


Abbildung 6.6: #4mittel mit Pollard

6.2.2 Speicherbedarf der Algorithmen

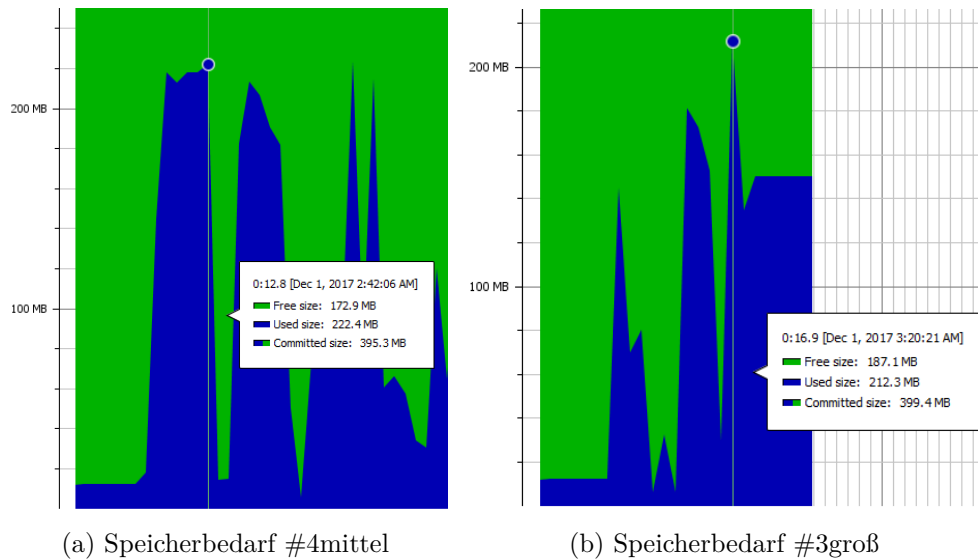
Die Größe des JVM Heap überwachte ich, wie bereits erwähnt, mit dem Programm JProfiler, von dem auch die Grafiken in diesem Kapitel als Screenshot entnommen sind.

Für die Speichertests führte ich jeden Fall mehrfach aus. Bevor ich zum nächsten Fall überging, hatte ich manuell (über JProfiler) den GC gestartet, um den Speicher wieder zurückzusetzen.

Pollards Rho-Algorithmus

Bei diesem Algorithmus belegten die kleinen Testfälle #1-#4 jeweils einen Speicher von max. 14,3 MB. Bei den mittleren DLPs wurde bereits mehr Speicher benötigt. Zum Beispiel wurde für den Fall #1 eine maximale Speichernutzung von 199 MB angezeigt. Leider kann nicht genau differenziert werden, welcher Anteil der Daten davon wirklich aktuell zur Berechnung genutzt wird und welcher Anteil bereits nicht mehr benötigte Daten (β_i, w_i, v_i) sind. Das liegt daran, dass der Garbage Collector veraltete Daten nicht sofort entfernt, sondern dafür eine eigene Heuristik besitzt.

Aus diesem Grund und dem deutlichen Anstieg an benötigtem Speicher ging ich zunächst davon aus, dass die anderen, mittleren DLPs #2-#4 ebenfalls deutlich mehr an Speicher benötigen. Doch meine Vermutung wurde nicht bestätigt, denn auch diese DLPs benötigten in jedem Testlauf maximale ~ 220 MB. Selbst das mehrfache Ausführen des Problems #4 (siehe Abbildung 6.7a) oder das große DLP #3 (siehe Abbildung 6.7b) blieben in diesem Rahmen.



Um zu überprüfen, ob der Speicherplatzbedarf sich wirklich auf diesem Level hält, startete ich schließlich die Berechnung des folgenden DLPs mit 12 GB erlaubter RAM-Größe:

$$2^x \equiv 95162 \pmod{1606938044258990275541962092341162602522202993782792835301611}$$

Bemerkenswerterweise blieb der RAM-Bedarf selbst in diesem Fall stets im Rahmen von 200 MB (siehe Abbildung 6.8). Anschließend überprüfte ich bei welchen RAM-Speicherständen der Garbage Collector eingeschritten ist. Da die niedrigste Speicherauslastung, bei der der GC aktiv wurde, 26,01 MB betrug, schlussfolgerte ich, dass bereits

zu diesem Zeitpunkt etliche veraltete Daten übrig sein müssten. Weiterhin beobachtete ich, was die niedrigste Datenmenge im RAM ist, die nach dem Einsatz des Garbage Collector vorhanden war. Dabei maß ich Werte im Rahmen von ~ 5 MB.

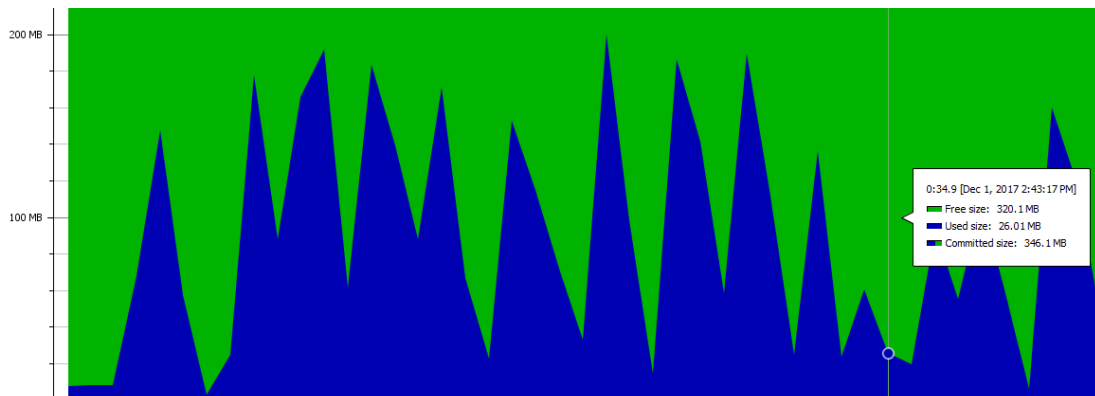


Abbildung 6.8: Speicherbedarf für ein DLP mit Ordnung im Bereich 2^{200}

Aus den ermittelten Daten schließe ich, dass der Speicheraufwand von Pollards Rho-Algorithmus selbst in der praktischen Umsetzung annähernd konstant ist.

Shanks Babystep-Giantstep-Algorithmus

Die kleinen Probleme #1-#4 verhalten sich bezüglich des Speicherbedarfs beim BSGS genauso, wie bei Pohligs Algorithmus, sodass maximal 14,3 MB benötigt werden. Ganz anders sieht es hingegen schon bei den mittleren DLPs aus: #1 benötigt ~ 60 MB, #2 ~ 70 MB, #3 bereits ~ 210 MB und #4 ~ 600 MB.

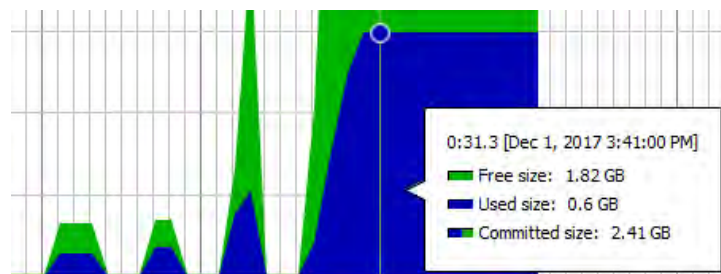


Abbildung 6.9: Speicherbedarf der vier mittleren DLPs mit Shanks Algorithmus

Dieser Bedarf an RAM erhöht sich immer weiter: Während #2 groß mit $\sim 3,28$ GB noch unter der standardmäßigen Heap-Größe von 4GB bleibt (siehe Abbildung 6.10), nutzt #4 groß bereits die vollen 14 GB an zugewiesenem RAM-Speicher aus (siehe Abbildung 6.11). Die 14 GB sind allerdings nicht ausreichend, um das DLP mit gleichbleibender Geschwindigkeit zu knacken, da der volle RAM-Speicher immer wieder auf die Festplatte geschrieben werden muss. Deswegen brach ich die zugehörige Berechnung wieder ab, nachdem auch mein Festplattenspeicher nahezu voll geschrieben war.

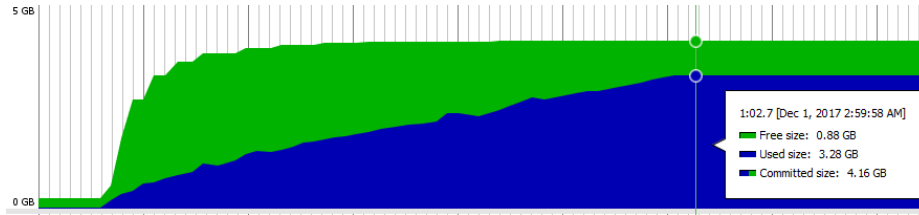


Abbildung 6.10: Speicherbedarf #2groß

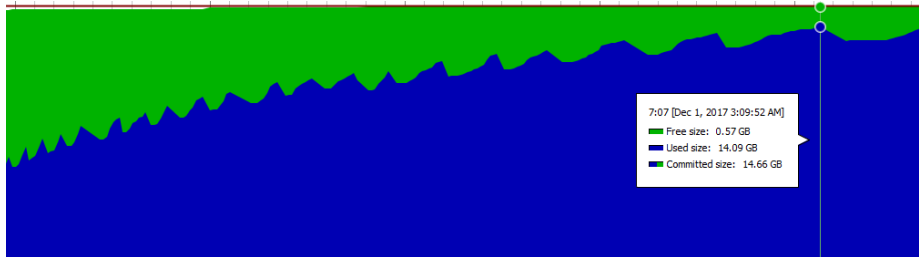


Abbildung 6.11: Ausschnitt: Speicherbedarf #4groß

Um Vorhersagen über den Speicherbedarf solch großer Probleme treffen zu können, stellte ich abschließend einen Bezug zwischen dem gemessenen Speicherbedarf und der Komplexität $\mathcal{O}(\sqrt{n})$ her. Dazu habe ich jeweils den benötigten Speicherplatz der getesteten Probleme (in Bytes) durch \sqrt{n} geteilt und einen gemittelten Faktor von $c \approx 256$ Byte erhalten.

Nun kann durch eine Multiplikation von $c \cdot \sqrt{n}$ der benötigte Speicherbedarf für andere Probleme abgeschätzt werden. Bezüglich des Problems #4groß ergibt sich damit ein erwarteter Speicherbedarf von:

$$256 \text{ Byte} \cdot \sqrt{2305843009213693951} \approx 3887361 \text{ MB} = 3.887 \text{ TB}$$

Infolgedessen erübrigte sich ein Test in noch größeren primitiven Einheitsgruppen, da dies noch mehr Arbeits- bzw. Festplattenspeicher erfordert hätte.

Pohlig-Hellman Algorithmus

Da der Pohlig-Hellman-Algorithmus in meiner Implementierung die generierten Untergruppen-DLPs mit Shanks bzw. Pollards Algorithmus löst, wird hier keine genauere Analyse bezüglich des Speicherbedarfs mehr durchgeführt. Durch die sequentielle Abarbeitung der einzelnen DLPs, ist der Speicherbedarf zum Zeitpunkt des Lösens eines Untergruppen-DLPs also etwa so hoch, wie wenn man das jeweilige Problem direkt mit Shanks bzw. Pollards Algorithmus löst. In den Pohlig-Hellman Versionen wird noch zusätzlicher Speicherplatz benötigt, um die Kongruenzen zu sichern. Dieser Speicherbedarf ist jedoch bei der Existenz von großen Primfaktoren in n so gering, dass er vernachlässigt werden kann.

6.2.3 Zusammenfassung

Die Wahl des Algorithmus hängt von den zur Verfügung stehenden Ressourcen und Informationen ab. Deshalb ist die Entscheidung für einen bestimmten Algorithmus immer individuell abzuwägen.

- Shanks Babystep-Giantstep-Algorithmus ist nur dann sinnvoll, wenn genügend schneller Arbeits- bzw. Festplattenspeicher zur Verfügung steht. Dieser kann im Voraus mit $256 \cdot \sqrt{n}$ Bytes abgeschätzt werden. Sofern man große Rechencluster mit genügend Arbeitsspeicher oder schnellem Festplattenspeicher besitzt, kann der Algorithmus eingesetzt werden.

$$\text{Laufzeitkomplexität: } \mathcal{O}(\sqrt{n}) \quad \text{Speicherkomplexität: } \mathcal{O}(\sqrt{n})$$

- Pollards Rho-Algorithmus kommt hingegen mit konstantem Speicherbedarf aus, was einen großen Vorteil darstellt.

Ein Negativum bei diesem Algorithmus ist allerdings, dass die Laufzeit bei unterschiedlichen Startwerten für ein DLP schwanken kann. Darin sehe ich jedoch auch einen Vorteil, insbesondere dann, wenn man die Berechnung eines DLPs auf mehreren Rechnern (oder Prozessorkernen) gleichzeitig laufen lässt. Dadurch ist es wahrscheinlich, dass ein Startwert ausgewählt wird, der eine kürzere Laufzeit zur Folge hat, als der Erwartungswert.

Da die erwartete Laufzeit beim Rho-Algorithmus die gleiche Komplexität wie Shanks Algorithmus besitzt und zudem weniger Speicherplatz benötigt, bevorzuge ich Pollards Algorithmus.

$$\text{Laufzeitkomplexität: } \mathcal{O}(\sqrt{n}) \quad \text{Speicherkomplexität: } \mathcal{O}(1)$$

- Der Pohlig-Hellman-Algorithmus ist die beste Option, sofern eine Primfaktorzerlegung des Modulowerts p bekannt ist. Damit kann eine Reduktion des ursprünglichen DLPs auf mehrere DLPs in Gruppen mit Primzahlordnungen ausgenutzt werden. Da die resultierenden DLPs mit einem anderen Verfahren gelöst werden müssen, ergibt sich insgesamt:

- Mit Shanks BSGS-Algorithmus:

$$\text{Laufzeitkomplexität: } \mathcal{O}(\sqrt{\max(\tau)}) \quad \text{Speicherkomplexität: } \mathcal{O}(\log(n) \cdot \sqrt{\max(p)})$$

- Mit Pollards Rho-Algorithmus

$$\text{Laufzeitkomplexität: } \mathcal{O}(\sqrt{\max(\tau)}) \quad \text{Speicherkomplexität: } \mathcal{O}(1)$$

7 Einbindung der Thematik in die Oberstufe des bayerischen Gymnasiums

Im Folgenden wird der Einfachheit halber die geschlechtsspezifische Unterscheidung zwischen Lehrerin und Lehrer mit der Bezeichnung Lehrer abgekürzt. Diese Verkürzung schließt jedoch immer beide Geschlechter mit ein. Analog gilt dies für die Unterscheidung von Schülerinnen und Schülern.

Diskrete Logarithmen in den klassischen Unterricht zu integrieren ist schwierig, da die Inhalte, die für das Verständnis dieser Arbeit notwendig sind, im Gymnasium nicht behandelt werden. Außerdem ist das Anwendungsgebiet nicht alltäglich, wodurch bei den Schülern und Schülerinnen (SuS) im Gymnasium zunächst die Motivation geschaffen werden muss, sich mit dieser Thematik auseinander zu setzen. Ferner fehlt die grundsätzliche Legitimation, diesen speziellen Inhalt im regulären Unterricht durchzunehmen, denn „das bayerische Gymnasium vermittelt seinen Schülern eine vertiefte Allgemeinbildung“ ([7]).

Dennoch bin ich der Meinung, dass dieser Inhalt in bestimmten Unterrichtssituationen und Interessensgruppen geeignet ist, um die spannende Anwendung der Mathematik vertiefend näher zu bringen. Denn das bayerische Kultusministerium schreibt bezüglich seiner Bildungsschwerpunkte auch, dass das Gymnasium zur Aufgabe hat, die Schüler „bestmöglich auf ein Hochschulstudium“ vorzubereiten ([7]). Deshalb kann ich mir gut vorstellen, dass das Thema für interessierte SuS der Oberstufe im Rahmen eines W-Seminars oder in einer Ferienakademie der Begabtenförderung unterrichtet werden kann (vgl. [6]).

Meine Idee ist, die diskreten Logarithmen als Teilaspekt einer Veranstaltung unter dem Leitmotto „Kryptologie“ zu behandeln. Dabei soll der Fokus nicht auf die Vermittlung der mathematischen Grundlagen gelegt werden. Stattdessen werden ausgewählte theoretische Inhalte durch die Verwendung in praktischen Beispielen - je nach Zielgruppe - mehr oder weniger ausführlich vermittelt. Durch die anwendbaren Beispiele der Ver- und Entschlüsselung, die die Schüler selbst per Hand durchführen, wirken die Inhalte auf sie ansprechender.

Nachdem die SuS ausreichende Kenntnis über die Konzepte sowie Vor- und Nachteile der symmetrischen und asymmetrischen Verschlüsselung erlangt haben, kann mit dem DHS das wirkungsvolle Zusammenspiel der beiden Prinzipien erklärt werden. Dazu muss eine angemessene vertikale didaktische Reduktion vorgenommen werden, um die Schüler nicht zu überfordern. Das bedeutet, dass man die für ein Thema nötigen Inhalte verein-

facht oder nur in Auszügen darstellt:

Von grundlegender Bedeutung sind die Operation des Modulo-Rechnens sowie die Definition, welche Elemente zur Menge einer primen Restklasse gehören. Um die Anwendung zu vereinfachen, werden in diesem Kontext als Modulowert für die primen Restklassen lediglich Primzahlen verwendet. Die genaue Definition von Gruppen und zyklischen Gruppen kann ebenso unerwähnt bleiben wie die konkrete Definition von Primitivwurzeln. Für die SuS genügt zu wissen, dass es in primen Restklassen (mit Primzahlen als Modulowert) Elemente gibt, die durch Potenzieren (unter der Anwendung der Modulo-Operation) wieder auf sich selbst abgebildet werden. Darauf aufbauend kann die Lehrkraft bereits die diskrete Exponentiation einführen und den Diffie-Hellman-Schlüsselaustausch erklären sowie ausprobieren.

In einem weiteren Schritt kann der Lehrer mit einer dritten, „bösen“ Person, argumentieren, die gerne den Schlüssel der symmetrischen Verschlüsselung herausfinden würde, um der Konversation zu lauschen. Dafür wird nun der diskrete Logarithmus, die Schwierigkeit seiner Berechnung und der dadurch entstehende Sicherheitsaspekt erklärt.

Für einen bereits mit diesen grundlegenden Kenntnissen geeigneten Algorithmus zum Berechnen eines DLPs halte ich Pollards Rho-Methode (mit fixem Startwert). Da er, wie in dieser Arbeit vorgestellt, jedoch für das Anspruchsniveau der Schüler zu komplex ist, wird die Optimierung der Speicherkomplexität durch das Arbeiten mit Phasen und Runden nicht erwähnt. Stattdessen würde ich den Algorithmus so verwenden, wie er in Abschnitt 4.2.1 erklärt wird. Zusätzlich bietet es sich dem Dozierenden an, die Problemstellung so zu wählen, dass das DLP immer gelöst werden kann und der Fall $v_i == v_q$ nicht auftritt.

So ergibt sich durch die vertikale didaktische Reduktion ein vereinfachter Algorithmus (siehe Anhang S. IX). Diesen kann der Lehrer den SuS als Teil eines Arbeitsblatts ausgeben, bei dem DLPs berechnet werden sollen.

Im Sinne einer horizontalen didaktischen Reduktion, also der Veranschaulichung eines Sachverhalts, kann ein Lehrer mit der grafischen Visualisierung des griechischen Rhos die Zyklenhaftigkeit der Folge beschreiben (siehe Abbildung 4.2). Danach lernen die Schüler, dass es bei einer unendlichen Folge von Elementen, denen eine endliche Menge an Elementen zu Grunde liegt, zwingend zu einer Kollision kommen muss. Dies kann im Anschluss z.B. durch ein Würfelbeispiel mit entsprechenden Erklärungen der Lehrkraft verständlich vermittelt werden:

„Du hast einen Würfel mit sechs Seiten vor dir liegen. Würfle einmal und schreibe dir die erhaltene Zahl auf. Wiederhole dies, bis du eine gewürfelte Zahl zum zweiten Mal aufschreibst. Du hast nun eine Kollision erzeugt!“

Der Vorteil an Pollards Rho-Variante mit fixem Startwert ist, dass sich der Lehrer Beispielaufgaben überlegen kann, die nicht das Problem $v_i == v_q$ beinhalten und dadurch von den SuS genau so reproduzierbar sind. Sofern der erweiterte euklidische Algorithmus vermieden werden soll, müsste der Dozent die Aufgaben auch so stellen, dass keine inversen Elemente zu berechnen sind. Alternativ kann er den Schülern auch eine Zuord-

nung von Elementen und ihren Inversen geben, oder diese Invertierung mit Hilfe eines Programmes vornehmen lassen.

Um die SuS bei der Anwendung des Algorithmus zu unterstützen, kann die Lehrkraft zusätzlich zur algorithmischen Beschreibung Arbeitsblätter mit auszufüllenden Tabellen und Feldern austeilen. Einzelne Werte in den Feldern oder Tabellen können bereits ausgefüllt sein, damit die Schüler kontrollieren können, ob ihre Berechnungen soweit stimmen. Im Anhang (siehe S. Xf.) findet sich Übungsblatt 1 als ein solches Beispiel inklusive eines Lösungsvorschlags.

Um mehr auf den visuellen Aspekt einzugehen wäre es auch denkbar, die SuS die gefundenen Folgeelemente in der Form eines Rhos aufschreiben zu lassen. Dort erkennen sie selbst, dass eine Kollision stattfindet. Auch hier ist ein Vordruck mit einer entsprechenden Anzahl an (teilweise ausgefüllten) Feldern geeignet, wie er im Anhang (siehe S. XII) als Übungsblatt 2 (inklusive Lösung) zu finden ist.

Alternativ können die Schüler auch ein Programm, wie den „Diskrete Logarithmen Rechner“ mit seinem erweiterten Ausgabe-Modus verwenden, um ihre Lösungen Schritt für Schritt oder abschließend zu kontrollieren.

Mit diesen Hilfestellungen kann die Lehrkraft den SuS mehrere einfache Aufgaben zur Berechnung geben, die von ihnen gelöst werden sollen.

Wenn sich die Schüler im Umgang mit Pollards Methode sicher sind, kann in einer Beispielaufgabe mit Diffie-Hellman-Schlüsselaustausch der gemeinsame Schlüssel über ein DLP berechnet werden. Zur weiteren Vertiefung kann der Lehrer den SuS eine Menge an Primzahlen mit zugehörigen Primitivwurzeln zur Verfügung stellen, damit sie für sich oder ihre Mitschüler eigene Aufgaben erstellen.

Obwohl die Berechnung von diskreten Logarithmen ein komplexes zahlentheoretisches Gebiet darstellt, lässt sich durch die eben dargestellte Vereinfachung des Inhalts dieses interessante kryptologische Thema mathematisch interessierten Schülern der Oberstufe näher bringen.

Das Informationsblatt und die Übungsaufgaben befinden sich inklusive der Lösungen in digitaler Form auch auf der beigelegten CD.

8 Neue Standards für Quantenkryptografie

In dieser Arbeit wurden mit Shanks Babystep-Giantstep-Algorithmus, Pollards Rho-Verfahren und der Pohlig-Hellman-Reduktion drei Methoden mit unterschiedlichem Laufzeit- und Speicheraufwand betrachtet, um diskrete Logarithmen zu bestimmen. Natürlich gibt es Bestrebungen die anfallenden Berechnungen noch effizienter zu machen. Dazu kann in der Informatik z.B die Parallelisierung auf mehreren Prozessorkernen genutzt werden. In der Mathematik wird an neuen Lösungsalgorithmen geforscht, und auch an Verbesserungen bereits existenter Verfahren. Ein weiterer bisher unerwähnter Algorithmus ist der Index Calculus. Er mit seiner subexponentiellen Laufzeit [13, S. 1] effizienter als die drei in dieser Arbeit vorgestellten Methoden.

Während die einen Forscher Algorithmen zum Berechnen von DLPs effizienter machen wollen, sorgen andere dafür, dass Kryptoverfahren weiterhin sicher bleiben. Aus diesem Grund basieren Kryptosysteme, die mit diskreten Logarithmen zu tun haben, heutzutage seltener auf primitiven Restklassengruppen, sondern eher auf elliptischen Kurven. Der Vorteil dieser mathematischen Struktur ist, dass man mit der Index Calculus Methode keine diskreten Logarithmen darauf berechnen kann [24, vgl. S. 29].

Auch die Physik wird inzwischen mit Kryptografie in Verbindung gebracht, denn dort wird an Quantencomputern gearbeitet. Einige Kommunikationsverfahren die heutzutage als sicher eingestuft sind, könnten durch Quantencomputer mit genügend Qubits mit geringem Aufwand geknackt werden. Unter anderem sind auch Kryptoverfahren wie der Diffie-Hellman Schlüsselaustausch angreifbar, denn das Problem zur Berechnung diskreter Logarithmen ist nach Buchmann [9, S. 231] „auf Quantencomputern leicht lösbar“. Bereits 1994 veröffentlichte Peter Shor einen Algorithmus für Quantencomputer [18, vgl. S. 1], der das Faktorisierungsproblem löst. Dieses Verfahren konnte er später in abgewandelter Form auf das diskrete Logarithmen Problem übertragen. Das bedeutet, dass mit der Verbreitung von Quantencomputern heutzutage gängige Algorithmen wie DHS, RSA oder Elliptische Kurven Kryptografie nicht mehr benutzt werden können.

Wie lange es dauert, bis Quantencomputer mit ausreichender Rechenleistung produziert werden, bleibt eine Frage der Zeit. Erst im Januar dieses Jahres stellte Intel auf der CES einen Quantenrechner mit 49 QuBits vor [21]. Obwohl nicht alle Kryptoverfahren durch die Existenz von Quantencomputern ihre Sicherheit verlieren, hat das amerikanische National Institute of Standards and Technology im Dezember 2016 einen Wettbewerb gestartet ([27, vgl. S. 3]). Nachdem bis zum November 2017 69 Algorithmenvorschläge eingereicht wurden [3], ist nun das Ziel, mehrere quanten-sichere Algorithmen zu standardisieren, um auch in einer Zeit mit Quantencomputern sichere Kryptoverfahren benutzen zu können.

9 Literatur

- [1] URL: <https://stackoverflow.com/questions/2457514/understanding-max-jvm-heap-size-32bit-vs-64bit#2457542> (besucht am 10.01.2018).
- [2] AMERICAN NATIONAL STANDARDS INSTITUTE. *Agreement of Symmetric Keys Using Discrete Logarithm Cryptography*. URL: <https://web.archive.org/web/20040903080553/http://csrc.nist.gov/CryptoToolkit/kms/summary-x9-42.pdf> (besucht am 10.01.2018).
- [3] AMERICAN NATIONAL STANDARDS INSTITUTE. *Post-Quantum Cryptography: Round 1 Submissions*. URL: <https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions> (besucht am 10.01.2018).
- [4] BAI, Shi und BRENT, Richard P. *On the Efficiency of Pollard's Rho Method for Discrete Logarithms*. URL: <https://maths-people.anu.edu.au/~brent/pd/rpb231.pdf> (besucht am 10.01.2018).
- [5] BAUMSLAG, Gilbert, FINE, Benjamin, KREUZER, Martin et al. *A Course in Mathematical Cryptography*. 1. Auflage. Berlin: De Gruyter, 2015.
- [6] BAYERISCHES STAATSMINISTERIUM FÜR BILDUNG UND KULTUS, WISSENSCHAFT UND KULTUR. *Begabtenförderung*. URL: <https://www.km.bayern.de/ministerium/institutionen/ministerialbeauftragte-gymnasium/schwaben/begabtenfoerderung.html> (besucht am 10.01.2018).
- [7] BAYERISCHES STAATSMINISTERIUM FÜR BILDUNG UND KULTUS, WISSENSCHAFT UND KULTUR. *Das Gymnasium in Bayern*. URL: <https://www.km.bayern.de/eltern/schularten/gymnasium.html> (besucht am 10.01.2018).
- [8] BRENT, Richard P. *An Improved Monte Carlo Factorization Algorithm*. URL: <http://maths-people.anu.edu.au/~brent/pd/rpb051i.pdf> (besucht am 10.01.2018).
- [9] BUCHMANN, Johannes. *Einführung in die Kryptographie*. 6. Auflage. Wiesbaden: Springer Spektrum, 2016.
- [10] BURTON, David. *Elementary Number Theory*. 7. Auflage. New York: McGraw-Hill Education, 2010.
- [11] DIEKERT, Volker, KUFLEITNER, Manfred und ROSENBERGER, Gerhard. *Diskrete algebraische Methoden - Arithmetik, Kryptographie, Automaten und Gruppen*. 1. Auflage. Berlin: De Gruyter, 2013.
- [12] ECKERT, Claudia. *IT-Sicherheit - Konzepte - Verfahren - Protokolle*. 9. Auflage. München: De Gruyter, 2014.

- [13] ENGE, Andreas und GAUDRY, Pierrick. *A General Framework for Subexponential Discrete Logarithm Algorithms*. URL: <https://www.impan.pl/shop/publication/transaction/download/product/83152> (besucht am 10.01.2018).
- [14] FINE, Benjamin und ROSENBERGER, Gerhard. *Number Theory - an Introduction via the Distribution of Primes*. 1. Auflage. Boston: Birkhauser, 2007.
- [15] FISCHER, Gerd. *Lehrbuch der Algebra*. 2. Auflage. Wiesbaden: Vieweg+Teubner, 2011.
- [16] *Fokus Mathematik - Jg.-Stufe 10*. 1. Auflage. Berlin: Cornelsen, 2008.
- [17] FORSTER, Otto. *Algorithmische Zahlentheorie*. 2. Auflage. Wiesbaden: Springer, 2014.
- [18] GERJUOY, Edward. *Shor's Factoring Algorithm and Modern Cryptography. An Illustration of the Capabilities Inherent in Quantum Computers*. URL: <https://arxiv.org/pdf/quant-ph/0411184.pdf> (besucht am 10.01.2018).
- [19] GERSTNER, Thomas. *Einführung in die Computerorientierte Mathematik*. URL: <https://www.uni-frankfurt.de/68995294/skript.pdf> (besucht am 10.01.2018).
- [20] HOLZ, Michael. *Repetitorium der Algebra*. 3. Auflage. Barsinghausen: Binomi, 2010.
- [21] INTEL. *2018 CES: Intel Advances Quantum and Neuromorphic Computing Research*. URL: <https://newsroom.intel.de/news/2018-ces-intel-advances-quantum-neuromorphic-computing-research/> (besucht am 10.01.2018).
- [22] KATZ, Jonathan und LINDELL, Yehuda. *Introduction to Modern Cryptography - Principles and Protocols*. 1. Auflage. Boca Raton: Chapman & Hall/CRC, 2007.
- [23] KREITZ, Christoph. *Kryptografie und Komplexität*. URL: <https://www.cs.uni-potsdam.de/plone/de/profs/ifi/theorie/lehre/ws1516/crypt-ws1516/slides/slides-5-2.pdf> (besucht am 10.01.2018).
- [24] KREITZ, Christoph. *Kryptographie und Komplexität. Endliche Körper und Elliptische Kurven*. URL: <https://www.cs.uni-potsdam.de/ti/lehre/09ws-Kryptographie/slides/slides-5.4.pdf> (besucht am 10.01.2018).
- [25] LIEBSCH, Oliver. *Diskrete Logarithmen*. URL: <https://www.wil.uni-muenster.de/pi/lehre/ws0708/seminar/Abgaben/Diskrete%20Logarithmen.pdf> (besucht am 10.01.2018).
- [26] MAY, Alexander. *Vorlesung Zahlentheorie - SS12*. URL: http://www.cits.rub.de/imperia/md/content/may/07_zmodnz.pdf (besucht am 10.01.2018).
- [27] MOODY, Dustin. *Post-Quantum Cryptography: NIST's Plan for the Future*. URL: https://pqcrypto2016.jp/data/pqc2016_nist_announcement.pdf (besucht am 10.01.2018).
- [28] ORACLE. *BigInteger (Java SE 9 & JDK 9)*. URL: <https://docs.oracle.com/javase/9/docs/api/java/math/BigInteger.html> (besucht am 10.01.2018).

- [29] ORACLE. *Primitive Data Types*. URL: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html> (besucht am 10.01.2018).
- [30] ORACLE TECHNOLOGY NETWORK. *Frequently Asked Questions About the Java VM*. URL: http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#gc_heap_32bit (besucht am 10.01.2018).
- [31] PAUL, Javin. *What is Difference between HashMap and Hashtable in Java?* URL: <http://javarevisited.blogspot.de/2010/10/difference-between-hashmap-and.html> (besucht am 10.01.2018).
- [32] POHLIG, Stephen und HELLMAN, Martin. „An Improved Algorithm for Computing Logarithms over $GF(p)$ and its Cryptographic Significance“. In: *IEEE Transactions on Information Theory*. Hrsg. von IEEE. Bd. 24. 1978.
- [33] POLLARD, John M. „Monte Carlo Methods for Index Computation“. In: *Mathematics of Computation*. Hrsg. von AMERICAN MATHEMATICAL SOCIETY. 1978.
- [34] REISS, Kristina und SCHMIEDER, Gerald. *Basiswissen Zahlentheorie - Eine Einführung in Zahlen und Zahlbereiche*. 3. Auflage. Berlin Heidelberg New York: Springer, 2014.
- [35] SCHIERMEYER, Ingo und BRAUSE, Christoph. *O-Notation*. URL: http://www.mathe.tu-freiberg.de/~brause/downloads/kt/ss_13/o-notation.pdf (besucht am 10.01.2018).
- [36] SCHMEH, Klaus. *Kryptografie - Verfahren, Protokolle, Infrastrukturen*. 5. Auflage. Heidelberg: dpunkt., 2013.
- [37] SHANKS, Daniel. „Class Number, a Theory of Factorization and Genera“. In: *Proceedings of Symposia in Pure Mathematics*. Hrsg. von AMERICAN MATHEMATICAL SOCIETY. Bd. 20. Providence, Rhode Island, 1971.
- [38] STAATSIINSTITUT FÜR SCHULQUALITÄT UND BILDUNGSFORSCHUNG (ISB). *Lehrplan Mathematik - Jahrgangsstufe 10 - G8*. URL: <http://www.isb-gym8-lehrplan.de/contentserv/3.1.neu/g8.de/index.php?StoryID=26221> (besucht am 10.01.2018).
- [39] VERKHOVSKY, Boris. *Integer Factorization: Solution via Algorithm for Constrained Discrete Logarithm Problem*. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.180.8072&rep=rep1&type=pdf> (besucht am 10.01.2018).
- [40] WAGSTAFF, Jr. Samuel. *Cryptanalysis of Number Theoretic Ciphers*. 1. Auflage. Boca Raton, Florida: CRC, 2002.
- [41] YAN, Song Y. *Number Theory for Computing*. 2. Auflage. Berlin: Springer Science & Business Media, 2002.
- [42] ZIEGENBALG, Jochen. *Elementare Zahlentheorie - Beispiele, Geschichte, Algorithmen*. 2. Auflage. Berlin: Springer, 2014.

10 Anhang: Schulmaterialien

Auf den folgenden vier Seiten befinden sich Arbeits- und Informationsblätter für Schüler sowie die zugehörigen Lösungen für Lehrer.

Infoblatt: Wie berechnet man einen diskreten Logarithmus?

Gegeben ist eine Primzahl p mit der zugehörigen primitiven Restklasse $G = (\mathbb{Z}/p\mathbb{Z})^*$. Das zu lösende Problem lautet $DLog_y(a) = x$, wobei y und a bekannt sind.

Pollards Rho-Algorithmus

1. Die Mengen G_1, G_2, G_3 sind durch die Rechnung mit Modulo-3 eingeteilt:

$$G_1 = \{x \in (\mathbb{Z}/p\mathbb{Z})^* : x \pmod{3} \equiv 2\}$$

$$G_2 = \{x \in (\mathbb{Z}/p\mathbb{Z})^* : x \pmod{3} \equiv 0\}$$

$$G_3 = \{x \in (\mathbb{Z}/p\mathbb{Z})^* : x \pmod{3} \equiv 1\}$$

2. Setze $\beta_1 = 1$, $w_1 = 0$ und $v_1 = 0$, sowie die Folgennummer $i = 0$. Bestimme $n = p - 1$.

3. wiederhole

a) $i = i + 1$

b) Bestimme die Mengenzugehörigkeit (MZK) von β_i

c) Berechne davon ausgehend mit folgender Tabelle β_{i+1} , w_{i+1} und v_{i+1}

	β_{i+1}	w_{i+1}	v_{i+1}
$\beta_i \in G_1$	$y \cdot \beta_i \pmod{p}$	$w_i + 1 \pmod{n}$	$v_i \pmod{n}$
$\beta_i \in G_2$	$(\beta_i)^2 \pmod{p}$	$2 \cdot w_i \pmod{n}$	$2 \cdot v_i \pmod{n}$
$\beta_i \in G_3$	$a \cdot \beta_i \pmod{p}$	$w_i \pmod{n}$	$v_i + 1 \pmod{n}$

d) Vermerke die berechneten Werte (z.B. in einer Tabelle) zusammen mit $i + 1$ bis (ein anderer Wert in der β -Spalte der Liste den selben Wert wie β_{i+1} hat)

4. Nenne die Folgennummern der identischen β -Werte k und l

5. Bestimme $t = w_k - w_l$, $s = v_l - v_k$ und $ggT(s, n) = d$

6. Bestimme mit dem erweiterten euklidischen Algorithmus die Lösung für \hat{x}

$$\hat{x} \equiv \frac{t}{d} \cdot \left(\frac{s}{d}\right)^{-1} \pmod{\frac{n}{d}}$$

7. for (int $j = 0$; $j < d$; $j++$)

Berechne $x \equiv \hat{x} + \frac{jn}{d} \pmod{n}$

if ($y^x = a$)

Terminierte mit x als Lösung.

Übungsblatt 1: Berechnung diskrete Logarithmen mit Pollards Rho-Algorithmus

Nutze für die Lösung den Algorithmus auf dem Informationsblatt

Das gegebene Problem lautet: $D\log_7(4)$ mit Primzahl $p = 11$.

- Gib die Werte von $n = \underline{\hspace{1cm}}$, $y = \underline{\hspace{1cm}}$ und $a = \underline{\hspace{1cm}}$ an.
- Berechne mit dem Infoblatt die erste β – Kollision. Fülle dabei die untere Tabelle aus.

i	β_i	w_i	v_i	$G?$
1	1	0	0	
2				3
3			2	
4		1		
5				
6	9			2
7			8	

- Bestimme die beiden Werte $k = \underline{\hspace{1cm}}$ und $l = \underline{\hspace{1cm}}$. Dabei ist $k < l$.
- Berechne $t = w_k - w_l = \underline{\hspace{1cm}}$ und $s = v_l - v_k = \underline{\hspace{1cm}}$ und $d = \text{ggT}(s, n) = \underline{\hspace{1cm}}$
- Bestimme mit folgender Tabelle das multiplikative Inverse für $(s/d)^{-1} = \underline{\hspace{1cm}}$
Hinweis: Bei negativem Wert muss zuerst $(\text{mod } n/d)$ gerechnet werden.

s/d	1	3	7	9
$(s/d)^{-1}$	1	7	3	9

- Bestimme $\hat{x} \equiv (t/d) * (s/d)^{-1} (\text{mod } n/d)$

$$\hat{x} \equiv (\underline{\hspace{1cm}}) * \underline{\hspace{1cm}} \quad (\text{mod } \underline{\hspace{1cm}}10) \equiv \underline{\hspace{1cm}}$$

- Berechne die Gesamtlösung:

for (int $j = 0$; $j < d$; $j++$)

Berechne $x \equiv \hat{x} + \frac{jn}{d} (\text{mod } n)$

if ($y^x = a$)

Terminiere mit $x = \underline{\hspace{1cm}}$ als Lösung.

Lösung Übung 1: Berechnung diskrete Logarithmen mit Pollards Rho-Algorithmus

Nutze für die Lösung den Algorithmus auf dem Informationsblatt

Das gegebene Problem lautet: $D\text{Log}_7(4)$ mit Primzahl $p = 11$.

1. Gib die Werte von $n = \underline{10}$, $y = \underline{7}$ und $a = \underline{4}$ an.
2. Berechne mit dem Infoblatt die erste β – Kollision. Fülle dabei die untere Tabelle aus.

i	β_i	w_i	v_i	G_i
1	1	0	0	3
2	4	0	1	3
3	5	0	2	1
4	2	1	2	1
5	3	2	2	2
6	9	4	4	2
7	4	8	8	3

3. Bestimme die beiden Werte $k = \underline{2}$ und $l = \underline{7}$. Dabei ist $k < l$.
4. Berechne $t = w_k - w_l = \underline{8}$ und $s = v_l - v_k = \underline{-7}$ und $d = \text{ggT}(s, n) = \underline{1}$
5. Bestimme mit folgender Tabelle das multiplikative Inverse für $(s/d)^{-1} = \underline{7}$
Hinweis: Bei negativem Wert muss zuerst $(\text{mod } n/d)$ gerechnet werden.

s/d	1	3	7	9
$(s/d)^{-1}$	1	7	3	9

6. Bestimme $\hat{x} \equiv (t/d) * (s/d)^{-1} \pmod{n/d}$

$$\hat{x} \equiv \underline{8} * \underline{7} \pmod{\underline{10}} \equiv \underline{6}$$

7. Berechne die Gesamtlösung:

for (int $j = 0$; $j < d$; $j++$)

Berechne $x \equiv \hat{x} + \frac{jn}{d} \pmod{n}$

if ($y^x = a$)

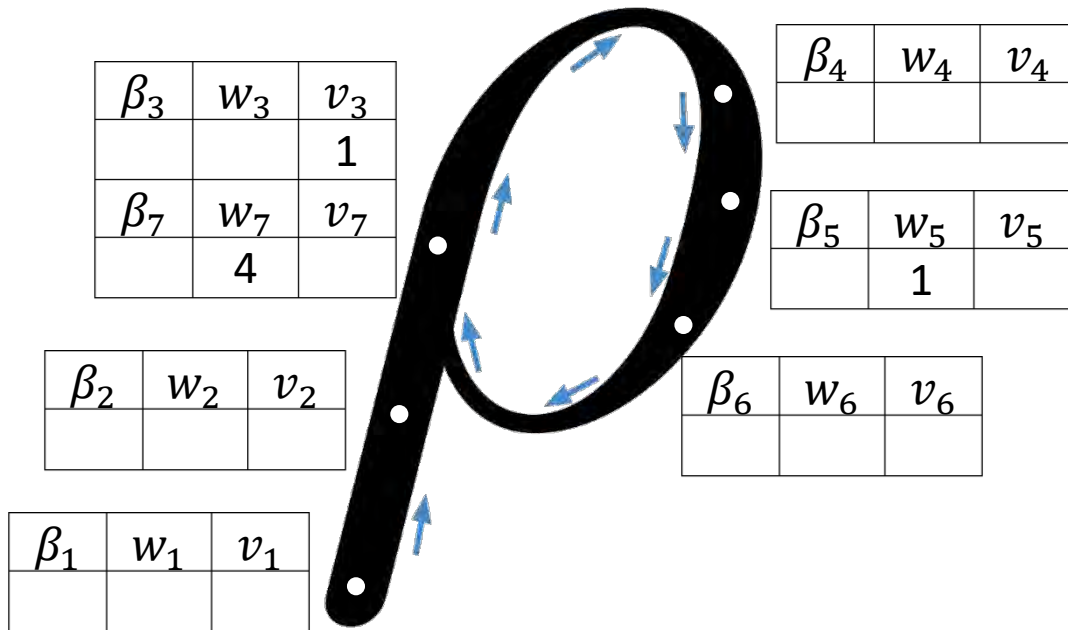
Terminierte mit $x = \underline{6}$ als Lösung.

Übungsblatt 2: Berechnung diskrete Logarithmen mit Pollards Rho-Algorithmus

Nutze für die Lösung den Algorithmus auf dem Informationsblatt.

Das gegebene Problem lautet: $\text{DLog}_6(8)$ mit Primzahl $p = 11$.

1. Gib die Werte von $n = \underline{\quad}$, $y = \underline{\quad}$ und $a = \underline{\quad}$ an.
2. Berechne die erste β – Kollision. Fülle dabei die untere Grafik aus.
3. Bestimme nun den diskreten Logarithmus: $\underline{\quad}$

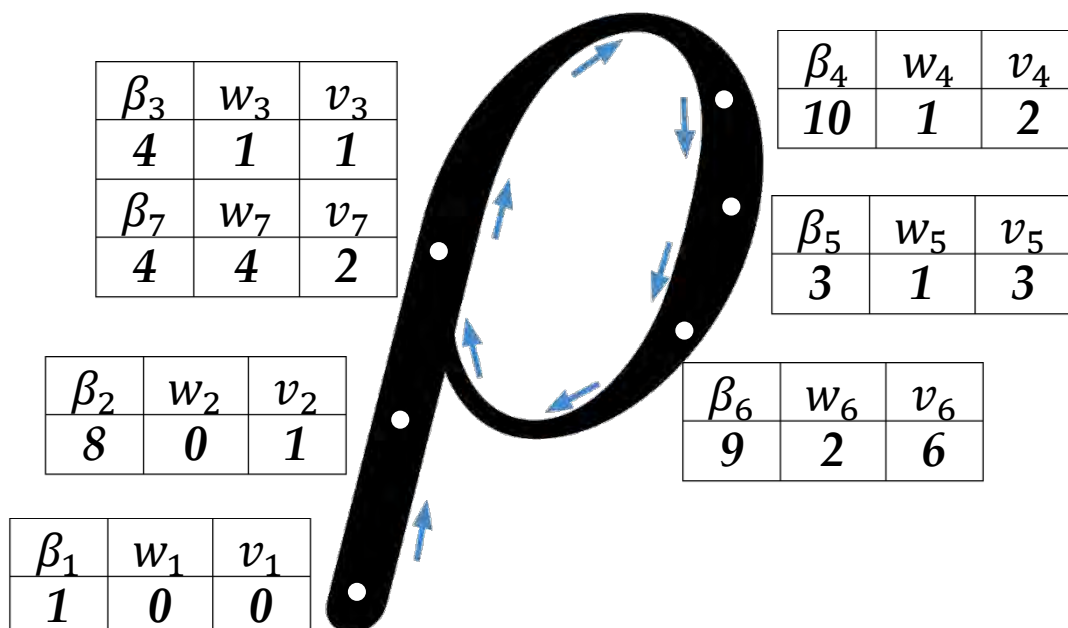


Lösung Übung 2: Berechnung diskrete Logarithmen mit Pollards Rho-Algorithmus

Nutze für die Lösung den Algorithmus auf dem Informationsblatt.

Das gegebene Problem lautet: $\text{DLog}_6(8)$ mit Primzahl $p = 11$.

1. Gib die Werte von $n = \underline{10}$, $y = \underline{6}$ und $a = \underline{8}$ an.
2. Berechne die erste β – Kollision. Fülle dabei die untere Grafik aus.
3. Bestimme nun den diskreten Logarithmus: $\underline{7}$



11 Eigenständigkeitserklärung

Hiermit erkläre ich, dass die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Die Arbeit wurde nicht bereits in derselben oder einer ähnlichen Fassung an einer anderen Fakultät oder in einem anderen Fachbereich zur Erlangung eines akademischen Grades eingereicht.

Passau, 24. Januar 2018