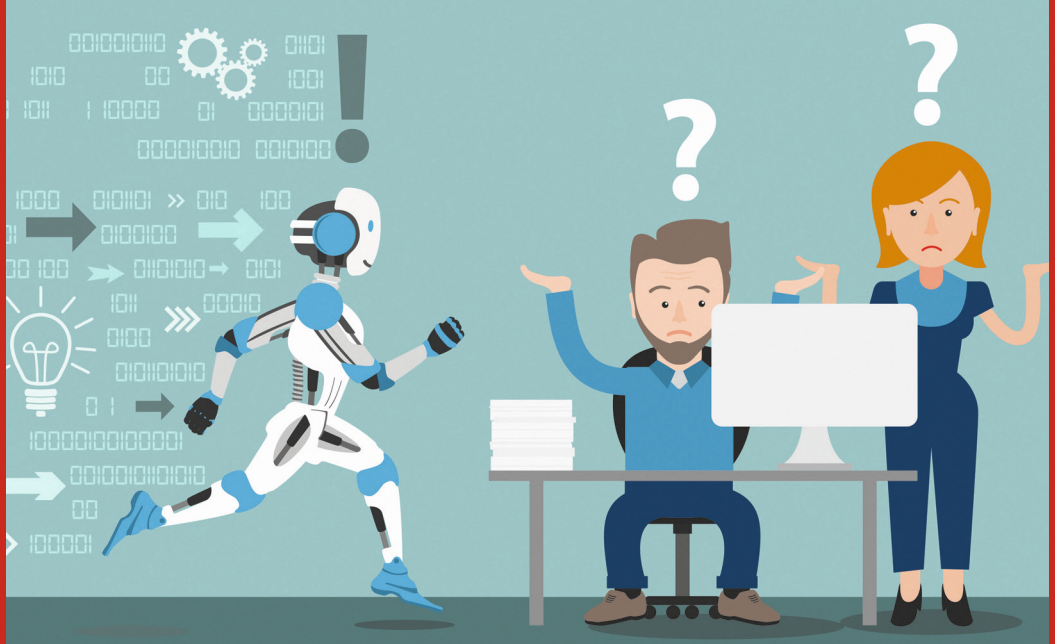

René Krooß

ALGORITHMEN UND DATEN-STRUKTUREN

Praktische Übungen für die Vorlesungen und Praktika

Quellcode zum Buch unter:
plus.hanser-fachbuch.de

HANSER

Krooß

Algorithmen und Datenstrukturen

René Krooß

Algorithmen und Datenstrukturen

Praktische Übungen für die Vorlesungen und Praktika

HANSER

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autor und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Lektorat: Brigitte Bauer-Schieweck
Copy editing: Petra Kienle, Früstenfeldbruck
Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de
Umschlagrealisation: Max Kostopoulos
Titelmotiv: © stock.adobe.com/Alexander Limbach
Satz: Eberl & Koesel Studio, Altusried-Krugzell
Druck und Bindung: Eberl & Koesel GmbH, Altusried-Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

Print-ISBN: 978-3-446-47222-8
E-Book-ISBN: 978-3-446-47241-9
E-Pub-ISBN: 978-3-446-47303-4

Inhalt

Vorwort

Bücher über Algorithmen gibt es mittlerweile viele und natürlich gibt es noch viel mehr Projekte, die komplexe Algorithmen verwenden. Warum muss es dann noch ein weiteres Buch über Algorithmen geben? Die Antwort ist, dass bei der Flut an Büchern, die es zum Thema Algorithmen mittlerweile gibt, das Thema Studium oft außen vor bleibt. In diesem Buch geht es deshalb um die Frage, welche Algorithmen speziell für das Studium wichtig sind und welche nicht. Eine zweite wichtige Frage ist natürlich auch die, ob Sie einen einfachen Einstieg in das komplexe Thema Algorithmen und Datenstrukturen finden können, der Ihnen im Studium weiterhilft. Die Antwort ist: Ja, auf jeden Fall, und das mit einem vertretbaren Aufwand. Die Frage, warum Sie sich gerade für dieses Buch entscheiden sollten, ist hiermit beantwortet: Bei den meisten Büchern über Algorithmen und Datenstrukturen, die zumindest ich selbst verwenden musste, steigt die Lernkurve oft schon am Anfang steil an. Auch wird oft Programmcode mit komplexen mathematischen Abhandlungen und Beweisen vermischt und dadurch kommt die Programmierpraxis zu kurz. Ich dagegen möchte den Schwerpunkt auf die Praxis legen und Ihnen Programmcode vorstellen, der sozusagen out oft the box lauffähig ist. Dabei fange ich ganz einfach an und erkläre zunächst einmal, wie Sie zwei Werte miteinander vertauschen oder Zahlen sortiert in ein Array einfügen können. Ich möchte an dieser Stelle aber nicht an Bücher wie „Algorithmen und Datenstrukturen für Dummies“ anknüpfen, die die meisten Dinge im Comic- oder Pseudocode-Stil erklären. Ich möchte schon ein oder zwei Stufen höher ansetzen und Ihnen ausführbaren Quellcode zeigen. Von mir aus können Sie sogar „Algorithmen und Datenstrukturen für Dummies“ parallel zu diesem Buch lesen, oder – falls Sie dieses Buch schon gelesen haben – gleich mit diesem Buch weitermachen. Sie haben in diesem Fall quasi Level 1 und 2 schon geschafft und können sich nun Level 3 vornehmen. Ich denke, 10 000 Erfahrungspunkte könnte ich Ihnen als Dungeon-Master schon für die Dummies anschreiben.

Ich will mit dem letzten Absatz keinesfalls ausdrücken, dass all die Bücher, die Sie in Ihrem Studium durcharbeiten müssen, überflüssig sind. Wenn Sie aber vorher gelernt haben, wie Sie so einfache Dinge wie den größten gemeinsamen Teiler oder die Primfaktoren einer natürlichen Zahl berechnen können, dann können Sie später auch komplexe mathematische Verfahren in ein laufendes Programm umsetzen. Grundlegende Kenntnisse in der Programmierung möchte ich natürlich voraussetzen, Erfahrungen mit der Programmiersprache C oder Java sind an dieser Stelle von Vorteil. Dies heißt natürlich nicht, dass ich Sie mit allerhand unverständlichen, komplexen Klassendiagrammen aus der objektorientierten Ent-

wicklung bombardiere, die nur ein Profi verstehen kann. Obwohl ich dies durchaus könnte, möchte ich versuchen, die Programmierung und auch die Mathematik, die hinter vielen Algorithmen steckt, möglichst leicht verdaulich zu präsentieren.

Kommen wir nun zum Aufbau dieses Buchs. Die einzelnen Kapitel sind jeweils ähnlich gegliedert. Zuerst erfolgt eine kurze Einleitung, worum es geht, und dann schließt sich, falls erforderlich, eine Beschreibung der eventuell erforderlichen Programmiertechniken an. Wenn es mathematische Grundlagen gibt, werden die entsprechenden Funktionen erläutert und auch, welche Variablen diese Funktionen verwenden. Ich habe besonders darauf geachtet, dass Sie zu jedem Algorithmus mindestens ein Listing vorfinden, das anschließend ausführlich erklärt wird. In den Kapiteln selbst finden sich oft verschiedene Tipps und Hinweise, die auch durch spezielle Kästchen mit der Überschrift „Hinweis“ gekennzeichnet sind. Ich empfehle Ihnen, die Hinweise ernst zu nehmen, denn so erleichtern Sie sich das Leben.

Ich empfehle Ihnen auch, auf die in den Kapiteln angegebenen Webseiten oder weiterführenden Links zu gehen, denn auch dort erhalten Sie wertvolle Informationen. Online stehen Ihnen außerdem alle Quellcodes zum Buch zur Verfügung. Auf dieser Website des Hanser-Verlags

https://plus.hanser-fachbuch.de/

geben Sie folgenden Zugangscode ein:

```
plus-35ipn-en7sk
```

Kommen wir nun zur verwendeten Programmiersprache. Sämtliche Programme in diesem Buch sind in C, C++ oder Java geschrieben. Dies sind die Programmiersprachen, die auch im Studium verwendet werden. Natürlich können Sie diese Programme nicht auf e-Book-Readern wie z. B. dem Tolino ausführen, obwohl diese unter Umständen sogar angezeigt werden, wenn Sie auf den entsprechenden Link klicken. Bei Tablets mag sich die Sache anders verhalten, weil dort zumindest Java manchmal vorinstalliert ist. Am Ende jedes Kapitels befindet sich eine Rubrik mit Übungen, die Sie möglichst alle bearbeiten sollten. Dies hat den einfachen Grund, dass die Übungen an den Praktika und Klausuren orientiert sind, deshalb können ähnliche Aufgaben durchaus auch in den Prüfungen vorkommen.

Nun möchte ich noch ein paar Worte über die Gliederung dieses Buchs verlieren. In den ersten Kapiteln des Buchs lernen Sie alle Grundlagen in Bezug auf Algorithmen kennen. Sie lernen, was ein Algorithmus ist, wie und ob man entscheiden kann, ob ein bestimmtes mathematisches Problem überhaupt berechenbar ist, und welche grundlegenden (und einfachen) Algorithmen es gibt. An dieser Stelle lernen Sie z. B., wie Sie die Werte zweier Variablen vertauschen können, wie Sie den GGT zweier Zahlen berechnen, wie Sie eine Zahl in ihre Primfaktoren zerlegen oder wie Sie mit nur wenigen Programmzeilen ein einfaches Such- oder Sortierverfahren implementieren können. Auch der Umgang mit beliebig langen Zahlen wird hier erklärt. So erfahren Sie z. B., wie Sie mit Hilfe der ägyptischen Division beliebig lange Zahlen durcheinander dividieren können.

Im zweiten Teil (ab dem 4. Kapitel) werden die fortgeschrittenen Themen behandelt, die direkt auf dem ersten Teil aufbauen. Hier geht es um Dinge, die im Studium wichtig sind und die auch in den Vorlesungen und Praktika immer wieder vorkommen. Dies sind z. B. verkettete Listen, schnelle Sortierverfahren, effektive Mustersuche in Texten und Bäume.

Ab Kapitel 7 behandelt der dritte Teil spezielle Themen, die unter Umständen in Wahlpflichtfächern vorkommen und so nicht unbedingt in den zweiten Teil gehören. Dies umfasst z. B. die grafische Bildverarbeitung, die Signalverarbeitung oder spezielle Algorithmen für neuronale Netze. Viele Algorithmen im letzten Teil werden Sie wahrscheinlich niemals verwenden, denn vor allem die Wahlpflichtfächer sind oft eine Sache des Geschmacks und der eigenen Vorlieben. Ich habe versucht, möglichst viele wichtige Themenbereiche abzudecken, kann aber auch nicht ausschließen, dass die eine oder andere Sache fehlt. So gibt es z. B. über hundert Sortierverfahren und ebenso viele Suchalgorithmen. Auch über reguläre Ausdrücke, die in diesem Buch nicht enthalten sind, gibt es inzwischen mehrere gute Bücher, die ich einfach nicht nochmal schreiben möchte.

Nun wünsche ich Ihnen viel Spaß mit diesem Buch. Sollten während des Lesens Unklarheiten oder Fragen aufkommen, so scheuen Sie sich nicht, mir eine E-Mail zu schicken:

renekrooss@t-online.de

Ich freue mich über ein Feedback von Ihnen.

René Krooß

René Krooß ist Diplom-Informatiker, Programmierer und Experte für Computer-Hardware, Videoverarbeitung und 3D-Rendering. Seine Hobbys sind Elektronik, Modellbau und Retro-Computing.

Teil I: Grundlagen

- Kapitel 1: Einführung
- Kapitel 2: Basisalgorithmen
- Kapitel 3: Rekursive Algorithmen

1 Einführung

„Was genau ist ein Algorithmus?“ Auf diese Frage müssen wir zuerst eine Antwort finden, wenn wir uns weiter mit dem Thema beschäftigen wollen – sonst ist alles weitere Vorgehen sinnlos. Die Antwort ist aber eigentlich nicht so schwer zu finden, denn was ein Algorithmus ist, ist streng definiert. Die Definition lautet in etwa wie folgt:

Definition Algorithmus

Ein Algorithmus ist eine streng formale, ausführbare Rechenvorschrift, die in einer überschaubaren Zeit für eine bestimmte Ausgangsbedingung (z. B. in Form einer Zahl) ein reproduzierbares Ergebnis liefert. Hierbei ist es zulässig, dass einzelne Rechenschritte mehrmals wiederholt werden.

Ähnliche Definitionen gibt es zuhauf in zahlreichen Lehrbüchern, manchmal sind sie länger, manchmal kürzer. Ich habe eine möglichst kurze Fassung gewählt, weil es mir auf *die Essenz* ankommt. Außerdem möchte ich von Anfang an Missverständnissen vorbeugen, denn oft werden Algorithmen mit Dingen verglichen, die sie gewiss nicht sind. Einige Lehrbücher vergleichen z. B. Algorithmen mit Kochrezepten, bei deren strikter Einhaltung ein reproduzierbares Gericht herauskommt. Leider ist bei einem Kochrezept schon die Reproduzierbarkeit nicht gegeben, denn jeder Koch kocht anders. Bei dem einem Italiener ist z. B. die Tomatensauce sämiger, bei dem anderen Italiener ist sie schärfer. Außerdem enthalten Kochrezepte Anweisungen, wie „eine Prise Salz“ hinzuzugeben oder das Gericht „auf kleiner Stufe“ zu garen. Keine dieser Anweisungen ist exakt ausführbar. Aber selbst eine Anweisung wie „1,534 Gramm Salz hinzufügen“ oder „bei 102,885 Grad 582,35 Sekunden lang garen“ beachtet noch nicht, dass jeder Herd anders ist und dass Wasser bei hohem Luftdruck später kocht als bei niedrigem Luftdruck. Ein klassisches Kochrezept ist also weit davon entfernt, ein Algorithmus zu sein, weshalb Algorithmen auch eher im mathematischen Bereich angesiedelt sind. Dort und nur dort sind streng formale, ausführbare Vorschriften mit einer genau festgelegten Ausgangsbedingung und reproduzierbarem Ergebnis denkbar. Ein gutes erstes Beispiel für einen Algorithmus ist die schriftliche Division zweier Dezimalzahlen a und b. Wenn Sie für a = 12 und für b = 5 einsetzen, dann erhalten Sie als Ergebnis stets 2,4 – egal wie oft Sie (natürlich in korrekter Weise) nachrechnen.

Kommen wir nun zum letzten Punkt, nämlich zu der Aussage, dass ein Algorithmus „in einer überschaubaren Zeit“ ein Ergebnis liefern muss. Für die Division von 12 durch 5 tut

die schriftliche Division genau dies: Sie kommt in nur wenigen Schritten zu einem korrekten Ergebnis. Leider ändert sich die Sache schon dramatisch, wenn Sie z. B. 10 durch 3 teilen: Ab dem dritten Schritt wiederholen sich die Ziffern endlos und Sie bekommen 3,3333 … heraus, ohne jemals den Rest 0 zu erhalten. Ihr Algorithmus endet also nicht in einer überschaubaren Zeit, wenn Sie keine zusätzliche Abbruchbedingung einfügen. Die Abbruchbedingung, die Sie wahrscheinlich schon aus der Schule kennen, ist die Perioden-Schreibweise. Sobald sich ein Rechenschritt wiederholt, wird der Algorithmus abgebrochen und man gibt eine Periode an.

1.1 Berechenbarkeit von Algorithmen

Ein großes (und wahrscheinlich das größte) Problem von Algorithmen ist also zu entscheiden, ob diese berechenbar sind. „Berechenbar“ im mathematischen Sinne heißt, dass ein Algorithmus in einer endlichen Anzahl von Schritten anhält und dann auch ein korrektes Ergebnis liefert. Leider ist die Aussage, dass ein bestimmtes mathematisches Problem berechenbar ist, erst bewiesen, wenn es einen Algorithmus für dieses Problem gibt, der für jede mögliche Eingabe (z. B. in Form einer Zahl) stets in einer endlichen Anzahl von Schritten ein korrektes Ergebnis liefert. Genau dieser Beweis ist sehr schwierig und kann oft nur mit einem mathematischen Beweisverfahren wie der vollständigen Induktion erbracht werden. Man kann also innerhalb der Mathematik bestimmte Aussagen beweisen, aber eben nicht alle. So gibt es z. B. keinen perfekten Algorithmus, der mir innerhalb einiger Sekunden für eine beliebig lange Zahl sagt, ob diese eine Primzahl ist, und dieser Algorithmus kann wahrscheinlich auch überhaupt nicht gefunden werden. Weil die Sache mit der Berechenbarkeit in der Mathematik nicht so einfach ist, hat man sie in späteren Definitionen von Algorithmen abgeschwächt – sie ist nun für ein Berechnungsverfahren, das ein Algorithmus „sein möchte“, nicht mehr unbedingt nötig, sondern nur noch wünschenswert.

Trotzdem ist die Frage nach der Berechenbarkeit mathematischer Probleme an dieser Stelle immer noch nicht ganz beantwortet. Gibt es zumindest ernstzunehmende Versuche, um herauszufinden, wie wir feststellen können, ob eine bestimmte Fragestellung in einer endlichen bzw. vernünftigen Zeit beantwortbar ist? Leider ist es so, dass wir diese Frage bis jetzt nicht beantworten können. In der Vergangenheit, vor allem in den 20er-Jahren des letzten Jahrhunderts, gab es viele Ansätze und Vorschläge aus der Gruppentheorie, die aber spätestens aufgegeben wurden, als der Mathematiker Kurt Gödel seinen Unvollständigkeitssatz aufstellte. Der Unvollständigkeitssatz besagt Folgendes: In jedem mathematischen Axiomensystem (dies ist quasi eine Sammlung von Regeln, die für eine bestimmte Zahlengruppe wie z. B. die natürlichen Zahlen gilt) gibt es immer Annahmen, die weder beweisbar noch widerlegbar sind. D. h. natürlich auch, dass es immer Algorithmen gibt, von denen man nicht sagen kann, ob sie in einer endlichen Anzahl von Schritten ein Ergebnis liefern. Dennoch gibt es einige populäre Ansätze, um herauszufinden, ob ein bestimmter Algorithmus zumindest nachvollziehbare Ergebnisse liefert. Einer der populäreren Ansätze, den man auch immer im Studium kennenlernt, ist das verwenden einer Turing-Maschine. Auf den folgenden Seiten werden Sie nun an Beispielen erfahren, wie eine Turing-Maschine arbeitet.

1.2 Wie eine Turing-Maschine arbeitet

In seinem berühmten Werk „On computable numbers“ (über berechenbare Zahlen) entwickelte Alan Turing ein Modell, mit dem man feststellen kann, ob ein mathematisches Problem berechenbar ist. Dieses Modell ist die später nach ihm benannte Turing-Maschine. Die Turing-Maschine ist so konstruiert, dass sie immer dann anhält, wenn ein mathematisches Problem berechenbar ist.

Eine Turing-Maschine wird oft in Form einer Eingabeeinheit dargestellt, in die ein unendlich langes Band hineinführt. Dieses Band, das Eingabeband, führt im einfachsten Fall auch direkt wieder aus der Maschine heraus und ist so auch gleichzeitig das Ausgabeband. Auf dem Band können nun – auch schon am Anfang – Symbole stehen. Diese Symbole sind beliebig wählbar, in den meisten Einführungsbeispielen in Bezug auf Turing-Maschinen sind dies aber fast immer Punkte und Striche, Nullen und Einsen oder Zahlen und Buchstaben. Wird die Turing-Maschine nun gestartet (dies kann z. B. durch einen Startknopf oder Hebel geschehen), liest sie zuerst das Symbol ein, über dem sich der Lesekopf zurzeit befindet. Je nachdem, welches Symbol sich nun gerade unter dem Lesekopf befindet, und je nachdem, in welchem Zustand sich die Maschine aktuell befindet, wird eine entsprechende Aktion ausgeführt. Diese Aktion ist meistens eine Ersetzung des Symbols, das sich unter dem Lesekopf befindet, durch ein anderes Symbol und ein anschließendes Verschieben des Bands nach rechts oder links. Natürlich kann der Ersetzungsschritt auch entfallen. Nach Ausführen einer Aktion wechselt die Maschine dann in einen anderen Zustand. Welcher Zustand dies ist, entnimmt die Maschine einer internen Tabelle. Gleichzeitig zu den Zustandswechseln können in der internen Tabelle auch noch zusätzliche Kommandos stehen, die z. B. das Band anhalten, wenn ein bestimmter Endzustand erreicht wird.

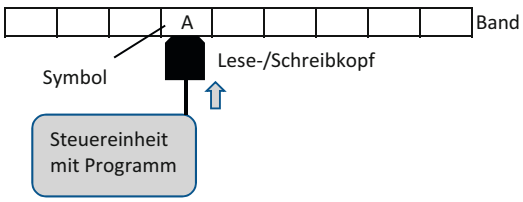

Bild 1.1 Arbeitsweise einer Turing-Maschine

Wie Sie in Bild 1.1 sehen können, hat das Eingabeband kein Ende und keinen Anfang. Ferner sind die Programme, die die Steuereinheit ausführt, fest verdrahtet. Weil die Turing-Maschine so ähnlich arbeitet wie ein sehr einfacher Computer, wird sie auch im Studium immer irgendwann besprochen. Meistens geschieht dies schon in den Vorlesungen zu den Grundlagen der Informatik, also in den ersten zwei Semestern. Deswegen habe auch ich die Turing-Maschine und die grundlegenden Überlegungen zur Berechenbarkeit mathematischer Probleme an den Anfang dieses Buchs gestellt. Die Betonung liegt hier auf „grundlegend“, ich werde also einige einfache Beispiele zu Turing-Maschinen anführen. Die ganzen komplexen mathematischen Formeln aus der Gruppentheorie, die Turing selbst verwendete, um sein Modell zu definieren, werde ich Ihnen natürlich ersparen.

Hinweise für Linux und den Raspberry Pi

Ich habe durchgehend dafür gesorgt, dass sämtliche Listings und Programme in diesem Buch sowohl unter Windows 10 als auch unter Linux ausführbar sind. Dies gilt auch für den inzwischen sehr populären Raspberry Pi – sämtliche Listings sind auf dem Pi übersetzbar, auch die Java-Beispiele. Wenn Sie auf dem Pi bestimmte Dinge beachten müssen, wird dies durch separate Kästchen mit dem Titel „Hinweise für den Raspberry Pi“ angezeigt.

1.2.1 Beispiel 1: Addition zweier Zahlen mit einer Turing-Maschine

Sie kennen wahrscheinlich die Binär-Schreibweise von Zahlen und wissen bereits, dass die Zahl 2 im Computer als „10“ abgebildet wird. Für die Addition von zwei Zahlen durch eine Turing-Maschine möchte ich aber eine noch einfachere Schreibweise mit Nullen und Einsen verwenden, nämlich die längencodierte Darstellung, die auch schon die ENIAC verwendete. Die ENIAC war der erste wirklich funktionierende Computer, den die Amerikaner entwickelt haben. Längencodierte Zahlendarstellung bedeutet nichts anderes, als eine Zahl in der entsprechenden Anzahl von Einsen darzustellen. Wenn Sie also in diesem Beispiel die Zahl 5 darstellen wollen, dann schreiben Sie „111110“. Dies sind fünf Einsen gefolgt von einer 0, die den Abschluss einer Zahl angibt. In diesem Beispiel müssen Sie also zwei Zahlen durch genau eine Null getrennt hintereinanderschreiben, die Trennung von Zahlen durch mehr als eine Null sei hier nicht erlaubt.

Kommen wir nun zu unserem ersten Beispiel, in dem die Zahlen 5 und 7 addiert werden sollen. Diese müssen Sie zuerst in folgender Form auf das Band schreiben:

111110111111110

In den folgenden Beispielen wird angenommen, dass ein Buchstabe im Text genau ein Symbol auf dem Band der Turingmaschine abbildet und dass sich der Lesekopf über dem ersten, linken Symbol im Text befindet. Nehmen wir nun an, dass die Turing-Maschine in diesem Beispiel einen Starthebel besitzt, den Sie zuerst umlegen müssen, um die Maschine in Gang zu setzen. Nehmen wir weiter an, dass Sie die Möglichkeit haben, das Band vorher korrekt einzulegen, und so den Lesekopf gezielt auf der ersten Eins (also ganz links) platzieren können. Ferner nehmen wir an, dass die Maschine immer zuerst Zustand Nr. 0 (den Initialisierungszustand) annimmt. Um zwei Zahlen zu addieren, müssen Sie nun folgende Regeln aufstellen:

- Wenn sich die Maschine in Zustand 0 befindet und das Symbol unter dem Lesekopf 1 ist, verschiebe das Band um eine Position nach links (dies ist quasi dasselbe, als ob der Lesekopf nach rechts wandert).
- Wenn sich die Maschine in Zustand 0 befindet und das Symbol unter dem Lesekopf 0 ist, so drucke das Symbol 1, verschiebe das Band um eine Position nach links und wechsele zu Zustand 1.
- Wenn sich die Maschine in Zustand 1 befindet und das Symbol unter dem Lesekopf 1 ist, verschiebe das Band um eine Position nach links.

- Wenn sich die Maschine in Zustand 1 befindet und das Symbol unter dem Lesekopf 0 ist, verschiebe das Band nach rechts (dies ist quasi dasselbe, als ob der Lesekopf nach links wandert), drucke das Symbol 0 und halte an.

Die Maschine startet in diesem Beispiel in Zustand 0 und liest so lange Einsen ein, bis sich die erste Null unter dem Lesekopf befindet. Diese Null wird dann durch eine Eins überschrieben und das Band wird danach um eine Position nach links gerückt. Die zu addierenden Zahlen werden dadurch zusammengefügt. Bevor die Maschine zu Zustand 1 wechselt, befinden sich nun die folgenden Zeichen auf dem Band (die eckigen Klammern geben die aktuelle Position des Lesekopfs an):

111111[1]1111110

Nun haben Sie aber die Zahlen noch nicht richtig addiert. Zählen Sie ruhig nach: Sie haben jetzt 13 Einsen auf dem Band stehen, 5+7 ist aber 12. Um diesen Fehler zu korrigieren, benötigen Sie eben einen zusätzlichen Zustand, nämlich Zustand 1. Dieser Zustand funktioniert zunächst wie Zustand 0 und rückt das Band so lange um eine Position nach links, bis eine Null gefunden wird. Diese Null ist aber nun das Ende des Addiervorgangs, bei dem die letzte Eins entfernt und danach die Maschine angehalten wird. Das Entfernen von Symbolen läuft dabei so ab: Das Band wird um eine Position zurückgeschoben (also nach rechts) und an diese Stelle wird eine Null geschrieben. Jetzt ist das Ergebnis korrekt und auf dem Band steht:

111111111111[0]0

Wahrscheinlich werden Sie mit dem letzten Beispiel einige Probleme gehabt haben, bis Sie es richtig verstanden haben. Das erste Problem war vermutlich die Tatsache, dass die Regeln, nach denen die Maschine arbeitet, in einer relativ unübersichtlichen 4-Punkte-Liste zusammengestellt wurden. Das zweite Problem war wahrscheinlich die Verschiebung des Bands, die umgekehrt zur Verschiebung des Lesekopfs verlaufen musste. Deshalb werde ich ab jetzt annehmen, dass sich der Lesekopf und nicht das Band bewegt, denn dadurch verändere ich nicht die grundlegenden Eigenschaften der Turing-Maschine, sondern nur die Notation der Richtungsangaben (Mathematiker sagen in diesem Fall, ich führe eine eigenschaftserhaltende Transformation durch). Außerdem verwende ich ab jetzt Zustandstabellen, in der Art, wie auch Turing sie später vorschlug, um die Übersichtlichkeit der oft zahlreichen Regeln zu erhöhen. Zustandstabellen können Sie sehr gut in Excel erstellen. Für dieses Beispiel gilt Tabelle 1.1.

Tabelle 1.1 Zustandstabelle der Turing-Maschine aus Beispiel 1.2.1

Zustand	Symbol	Schreibe	Verschiebung	Nächster Zustand
0	1	–	1 rechts	0
0	0	1	1 rechts	1
1	1	–	1 rechts	1
1	0	–	1 links	2
2	–	0	–	Halt

Ein kleiner Nachteil der Zustandstabellen bleibt: Für das Entfernen der letzten Eins am Ende der Addition benötigen Sie einen zusätzlichen Zustand, weil die Tabelle immer nur

einen Zustandswechsel und eine Lesekopf-Verschiebung pro Schritt abbilden kann. Der Vorteil ist hier aber die gute Abbildbarkeit von Zustandstabellen z. B. auf ein C-Programm, mit dem Sie dann beliebige einfache Turing-Maschinen simulieren können. Hierbei sind die sogenannten einfachen Turing-Maschinen Turing-Maschinen, die nur ein Band und nur einen Lese-/Schreibkopf besitzen.

Kommen wir nun zu der Frage, ob Sie es in dem letzten Beispiel mit einem berechenbaren Problem zu tun haben. Bleibt die Turing-Maschine also für jede Eingabe stehen und liefert sie dann auch stets ein korrektes Ergebnis? Um diese Frage zu beantworten, wähle ich nun folgende fehlerhafte Eingabe aus:

111001110

Die Turing-Maschine startet hier wieder in Zustand 0 und liest so lange Einsen, bis sich die erste 0 unter dem Lesekopf befindet. Nach dem Überschreiben der ersten 0 mit einer 1 und dem Verrücken des Lesekopfs nach rechts wechselt die Maschine zu Zustand 1. Das Band sieht nun so aus:

1111[0]1110

Nun ist das Symbol unter dem Lesekopf „0“, und dies bedeutet, dass die Maschine die Eins vor der Null entfernt, und danach anhält. Das Band enthält nun folgende Ausgabe, die sich anschließend auch nicht mehr ändert:

111[0]01110

Dieses Ergebnis lässt sich entweder als die Zahl 3 oder als ungültige Ausgabe interpretieren. Auf jeden Fall ist das Ergebnis falsch, obwohl die Turing-Maschine für jede beliebige Eingabe einer endlichen Anzahl von Einsen stehen bleibt. Haben Sie es aber in diesem Beispiel wenigstens mit einem berechenbaren Problem zu tun? Die Antwort, die Turing hier geben würde, ist „nein“, denn ein Algorithmus sollte immer das korrekte Ergebnis liefern und dies tut der Algorithmus in diesem Beispiel für unkorrekte Eingaben nicht. Wenn Sie aber voraussetzen, dass sämtliche Eingaben stets im korrekten Format vorliegen, dann haben Sie es hier in der Tat mit einem berechenbaren Problem zu tun, sofern Sie eine endliche Anzahl an Einsen auf dem Band voraussetzen.

1.2.2 Beispiel 2: Suchen und ersetzen

Mit dem Wissen aus dem letzten Beispiel können Sie jetzt eine einfache Zeichenersetzung realisieren. Im nächsten Beispiel sollen alle Buchstaben durch Großbuchstaben ersetzt werden. Wenn das gelesene Symbol bereits ein Großbuchstabe ist, können Sie es einfach durch sich selbst ersetzen. Zahlen, Kommas, Bindestriche und Leerzeichen sollen ignoriert werden und ein Satz soll grundsätzlich durch einen Punkt beendet werden. Ein Punkt soll dann auch die Maschine stets zum Anhalten bringen. Der Ausgangstext soll wie folgt lauten:

1937 hat Turing seine Turing-Maschine erfunden.

Die Maschine startet nun wieder in Zustand 0 und findet die Zahl 1. Dieses Symbol wird ignoriert und der Lesekopf rückt um ein Feld nach rechts. Dieser Schritt wird nun für jede Zahl wiederholt und natürlich auch für das Leerzeichen. Das Zeichen „h“ wird dann durch das Zeichen „H“ ersetzt. Das Band enthält nun folgende Symbole:

1937 H[a]t Turing seine Turing-Maschine erfunden.

Auch das Zeichen „a“ wird durch das Zeichen „A“ ersetzt und der Lesekopf befindet sich anschließend auf dem Zeichen „t“. Anscheinend funktioniert unser Ersetzungsalgorithmus einwandfrei, wenn wir Tabelle 1.2 verwenden.

Tabelle 1.2 Zustandstabelle für die Turing-Maschine aus Beispiel 1.2.2

Zustand	Symbol	Schreibe	Verschiebung Lesekopf	Nächster Zustand
0	Leerzeichen	Leerzeichen	1 rechts	0
0	0	0	1 rechts	0
0	1	1	1 rechts	0
…	…	…	…	…
0	9	9	1 rechts	0
0	a	A	1 rechts	0
0	b	B	1 rechts	0
…	…	…	…	…
0	z	Z	1 rechts	0
0	.	.	–	Halt

Ich habe in der letzten Tabelle immer dort drei Punkte eingesetzt, wo Sie sich die entsprechenden Einträge dazu denken müssen. So muss ich nicht z. B. sämtliche Zustandsänderungen zwischen den Zahlen 1 und 9 auflisten, sondern Sie können die entsprechenden Ersetzungen zwischen den Zahlen 2 und 8 im Geist ergänzen. An dieser Stelle sehen Sie wahrscheinlich schon den größten Nachteil von Turing-Maschinen: Die Zustandstabellen können wahrlich umfangreich werden, denn Sie müssen dort jedes Symbol einzeln auflisten. Außerdem sind zumindest die einfachen Turing-Maschinen sehr unflexibel und Sie benötigen quasi für jeden Algorithmus eine separate Maschine. Turing führte später den Gedanken einer universellen Maschine ein, den er aber leider aus gesundheitlichen Gründen nicht mehr zu einer vollständigen Theorie ausbaute.

In diesem Buch geht es aber nicht um das wahrlich tragische Schicksal Alan Turings, sondern darum, wie seine Ideen später in der Informatik eingesetzt wurden. Deshalb ist die Frage nun, ob auch das Ersetzungsverfahren in diesem Beispiel ein Algorithmus ist, der auf jeden Fall anhält. Wenn Sie aber etwas nachdenken, kommen Sie schnell darauf, dass das Ersetzungsverfahren in diesem Beispiel z. B. niemals anhält, wenn der Punkt am Satzende fehlt oder wenn ein Zeichen im Text auftaucht, für das keine Ersetzungsregel existiert (in diesem Fall würde die Turing-Maschine einfach nichts tun und das für immer). Fügen wir nun die folgende implizite Regel ein: „Immer, wenn für ein Symbol keine Regel zutrifft oder das betreffende Feld leer ist, halte an.“ Nun hält unser Algorithmus zwar stets an, liefert aber nicht immer eine vollständige Ersetzung aller Buchstaben durch Großbuchstaben. Nehmen Sie z. B. folgenden Satz als Eingabe an:

Hallo Klaus! Es regnet, darum bleibe ich heute zuhause.

Da für das Zeichen „!“ keine Regel existiert, hält die Turing-Maschine dort an und das Band enthält in diesem Fall folgenden Text:

HALLO KLAUS[!] Es regnet, darum bleibe ich heute zuhause.

Ändern wir nun unsere implizite Regel wie folgt ab: „Immer, wenn für ein Symbol keine Regel zutrifft, ersetze das gelesene Symbol durch sich selbst und bewege den Lesekopf um einen Schritt nach rechts. Außerdem halte an, wenn der Lesekopf auf ein leeres Feld trifft." Nun können wir auch den Rest des Satzes in Großbuchstaben umwandeln und sogar den Punkt am Ende weglassen. Wenn wir voraussetzen, dass das Band nur eine endliche Anzahl nicht leerer Felder enthalten darf und dass ferner der Eingabetext nicht durch leere Felder unterbrochen werden darf, wird unser Beispiel aus Abschnitt 1.2.2 berechenbar und liefert auch stets korrekte Ergebnisse.

1.2.3 Beispiel 3: Multiplikation zweier Zahlen mit einer erweiterten Turing-Maschine

Wollen wir nun versuchen, zwei Zahlen nicht nur zusammenzufügen, sondern auch zu multiplizieren (die Multiplikation war übrigens lange Zeit das größte Problem bei der ENIAC). Wir wollen also statt 5+7 5*7 berechnen. Die Ausgangsschreibweise der Zahlen soll wieder wie im allerersten Beispiel erfolgen, nämlich als:

11111011111110

Sie müssen nun versuchen, die Symbolfolge „11111110" fünfmal hintereinanderzuhängen. Dies können Sie wie folgt realisieren: Sie ersetzen erst einmal die erste Eins durch eine Null und suchen danach die nächste Null. Dies ist die Null vor der Zahl, die hinter dem Produkt steht. Anschließend müssen Sie die Folge „1111111" kopieren. Danach müssen Sie zum ersten Zeichen zurückfinden, das nicht 0 ist, und den Kopiervorgang der Folge „1111111" so lange wiederholen, bis die erste Ziffernfolge vor dem Multiplikanden *aufgebraucht* ist. Ein großes Problem hierbei ist leider, dass Sie den Multiplikanden in dem Moment verändern, in dem Sie dort eine Zeichenkette anhängen. Das Problem der Multiplikation zweier Zahlen ist in der Tat mit einer Art einfacher Turing-Maschinen realisierbar, die sich *fleißige Biber* nennen, aber diese nun darzustellen, würde so manchen Leser schon jetzt *aussteigen* lassen. Ich will natürlich vermeiden, dass Sie schon jetzt aufgeben, Ihnen aber trotzdem ein Gefühl dafür geben, wo die Grenzen der ursprünglichen Turing-Maschinen liegen und was diesen fehlt. Dies wären folgende Dinge:

- Turing-Maschinen besitzen kein Gedächtnis in Form eines wie immer gearteten internen Speichers und können sich z. B. nicht daran erinnern, welches Symbol im letzten Schritt gelesen wurde.
- Turing-Maschinen können nicht an eine bestimmte Position direkt zurückspringen, sondern können den Lesekopf nur relativ zur aktuellen Position bewegen (man sagt auch, Turing-Maschinen können nur relative Sprünge ausführen).
- Bei Turing-Maschinen kann man das Programm, das ihnen einmal eingegeben wurde, nicht mehr ändern.
- Turing-Maschinen können nicht *lernen*, z. B. indem man ihnen später neue Regeln hinzufügt.

Kommen wir nun zurück zur Multiplikation zweier Zahlen. Dieses Problem können Sie nicht so einfach mit einer einfachen Turing-Maschine lösen, deshalb möchte ich die ursprüngliche Maschine durch einen Puffer erweitern. Ein Puffer ist ein Zwischenspeicher, der eine

bestimmte Anzahl an Symbolen aufnehmen kann, um diese später bei Bedarf weiterzuverarbeiten. Ich will nun annehmen, dass ein zusätzlicher Puffer in einer erweiterten Turingmaschine bis zu 100 Symbole aufnehmen kann und dass ein spezielles Kommando den Pufferinhalt auf das Band schreiben kann, wobei der Lesekopf automatisch um die entsprechende Anzahl an Symbolen weiterbewegt wird. Durch einen Puffer können also Kopiervorgänge relativ einfach realisiert werden. Ferner will ich annehmen, dass es ein spezielles Kommando gibt, mit dem die Maschine sich die aktuelle Position des Lesekopfs merken kann, um später zu dieser Position zurückkehren zu können. Mit der erweiterten Turing-Maschine ist die Multiplikation zweier Zahlen nun relativ einfach realisierbar. Schauen Sie sich dazu das Ausgangsbeispiel noch einmal an. Am Anfang enthält das Band folgende Symbole:

11111011111110

Die Maschine befindet sich am Anfang in Zustand 0. In diesem Zustand liest sie zunächst das aktuelle Symbol unter dem Lesekopf, das am Anfang eine Eins ist. Wenn das Symbol unter dem Lesekopf eine Eins ist, wird diese durch eine Null ersetzt und der Lesekopf bewegt sich um eine Position nach rechts. Die Maschine merkt sich danach die aktuelle Position P und wechselt danach zu Zustand 1. In Zustand 1 wird der Lesekopf nun so lange nach rechts bewegt, wie sich unter diesem eine Eins befindet. Eine Null dagegen bewegt den Lesekopf um eine Position nach rechts, die Maschine leert den Zeichenpuffer und wechselt anschließend zu Zustand 2. In Zustand 2 liest die Maschine nun ebenfalls so lange Einsen vom Band, bis eine Null gefunden wurde, jedoch speichert sie nun jede Eins im Zeichenpuffer. Wurde in Zustand 2 eine Null gefunden, gibt die Maschine daraufhin den Zeichenpuffer aus, bewegt den Lesekopf zurück zu der Position, die sich die Maschine vorher gemerkt hat, und wechselt daraufhin wieder zu Zustand 0. Nun kann der Kopiervorgang erneut stattfinden, bis irgendwann alle Einsen am Anfang *aufgebraucht* sind und sich eine Null unter dem Lesekopf befindet. Dies ist die Null direkt vor dem Multiplikanden.

Wenn Sie jedoch aufgepasst haben, dann haben Sie gemerkt, dass die Maschine ein falsches Ergebnis liefert. Das liegt daran, dass wir ein Problem noch nicht gelöst haben: Immer, wenn an das Ende etwas angehängt wird, ändert sich automatisch der Multiplikand, wodurch beim nächsten Kopiervorgang die doppelte Anzahl Zeichen kopiert wird. Genau deshalb muss die ursprüngliche Zustandstabelle durch Kommandos erweitert werden, die z. B. explizit Zeichen in den Puffer kopieren oder diesen bei Bedarf leeren können. Ferner reichen die Symbole „0“ und „1“ nicht mehr aus, um z. B. die Eingabe von dem Ergebnis trennen zu können. Deshalb habe ich eine Zwei an der Position eingesetzt, ab der das Ergebnis stehen soll. Die Eingabe lautet also jetzt wie folgt:

11111011111112

Schauen wir, ob die Maschine nun korrekt arbeitet. In Zustand 0 liest die Maschine die Zahl 1 ein, ersetzt diese durch eine Null, bewegt den Lesekopf um eine Position nach rechts, merkt sich die aktuelle Position P und wechselt zu Zustand 1. In diesem Zustand werden jetzt so lange Einsen gelesen, bis unter dem Lesekopf eine Null steht. Ist dies der Fall, rückt der Lesekopf um eine Stelle nach rechts, der Zeichenpuffer wird geleert und die Maschine wechselt zu Zustand 2. In diesem Zustand werden nun so lange Einsen gelesen und im Zeichenpuffer gespeichert, bis eine Zwei gefunden wird. Wird eine Zwei gefunden, rückt der Lesekopf eine Position nach rechts und die Maschine wechselt zu Zustand 3. In diesem Zustand werden so lange Einsen gelesen, bis sich unter dem Lesekopf eine Null oder ein

leeres Feld befindet. Ist dies der Fall, wird der Zeichenpuffer ausgegeben. Nach dem ersten Kopiervorgang steht auf dem Band Folgendes:

011110111111121111111[leer]

Nachdem der Zeichenpuffer ausgegeben wurde, wechselt die Maschine zu Zustand 4. Zustand 4 ist nur dazu da, das Kommando „bewege den Lesekopf zurück zu der vorher gemerkten Position P“ auszuführen und anschließend zu Zustand 0 zu wechseln. Dies ist hier die zweite Position auf dem Band, das jetzt folgende Symbole enthält:

0[1]11101111111121111111

Da sich unter dem Band eine Eins befindet, wird der Kopiervorgang wiederholt: Zuerst ersetzt die Maschine die 1 durch eine 0, anschließend sucht die Maschine die Zahl 0. Danach leert sie den Zeichenpuffer, liest sämtliche Einsen in den Puffer ein, bis sich die Zwei unter dem Lesekopf befindet, und sucht anschließend das erste leere Feld hinter der letzten Eins. Durch die Trennung von Eingabebereich und Ergebnisbereich werden die Einsen nun korrekt zusammengefügt. Am Ende enthält das Band folgende Ausgabe:

000000111111111111111111111111111111111112

Leider kann auch die erweiterte Turing-Maschine unter Umständen unvorhergesehenes Verhalten zeigen, wie z. B. bei der folgenden Eingabe:

11111211111112

Für Zustand 0 wurde nicht definiert, wie bei dem Symbol „2“ verfahren werden soll, deshalb friert die Maschine an dieser Stelle ein. Wenn Sie sich wieder dafür entscheiden, vorher die implizite Annahme zu machen, dass die Maschine stehen bleiben soll, falls für ein Symbol keine Regel existiert, wird am Ende das Ergebnis falsch sein. Es folgt die erweiterte Zustandstabelle für das letzte Beispiel.

Tabelle 1.3 Zustandstabelle für die Turing-Maschine aus Beispiel 1.2.3

Zustand	Symbol	Schreibe	Verschiebung Lesekopf	Kommando	Nächster Zustand
0	1	0	1 rechts	Merke aktuelle Position P	1
0	0	–	–	–	Halt
1	1	–	1 rechts	–	1
1	0	–	1 rechts	Leere Zeichenpuffer	2
2	1	–	1 rechts	Übertrage Symbol in den Zeichenpuffer	2
2	2	–	1 rechts	–	3
3	1	–	1 rechts	–	3
3	leer	–	–	Ausgabe Zeichenpuffer	4
3	0	–	–	Ausgabe Zeichenpuffer	4
4	–	–	–	Zurück zu Position P	0

1.2.4 Von der Turing-Maschine zum Prozessor

Das letzte Beispiel funktioniert sehr gut für kleine Zahlen bis 100, danach ist entweder der Zeichenpuffer voll oder es dauert (bei einem beliebig erweiterbaren Puffer) sehr lange, um z. B. eine Zahl 1000-mal in den Ergebnisbereich zu übertragen. Kurz: Wir benötigen für komplexere Algorithmen eine alternative Lösung. Diese Lösung ist, die Zahlen anders darzustellen. Statt viele Einsen aneinanderzuhängen, wird mit den Nullen und Einsen ein Stellenwertsystem aufgebaut, das dem gewohnten Zahlensystem mit zehn Ziffern ähnelt. Anstatt jedoch nach der Zahl 9 eine neue Stelle zu benutzen und danach „10“ zu schreiben, muss in dem Binärsystem, das Computer benutzen, schon nach der Ziffer 1 (also für die Zahl 2) „10“ geschrieben werden. Der Vorteil des Binärsystems ist, dass dies sehr gut in elektronischen Schaltungen realisiert werden kann und die Ziffernfolgen so lang auch wieder nicht sind. Zumindest müssen die Zahlen im Binärsystem nicht so umständlich codiert werden wie die Zahlen in den letzten Beispielen. Eine weitere Überlegung der Ingenieure ab den 50er-Jahren war, mehrere binäre Symbole gleichzeitig zu verarbeiten und auf diese Weise zu Blöcken zusammenzufassen. Mit diesen Blöcken konnten dann auch (wieder durch entsprechende elektronische Schaltungen) grundlegende Operationen wie z. B. Addition, Subtraktion, Multiplikation und Division durchgeführt werden. Diese Blöcke von binären Zeichen (dies waren am Anfang oft vier oder acht Binärsymbole) nennt man Register, die auf diese Weise erweiterten Turing-Maschinen, die auch von sich aus grundlegende Rechenoperationen durchführen können, nennt man Prozessoren. Ein Prozessor ist der wichtigste Bestandteil moderner Computer und das elektronische Bauteil, das die Programme ausführt. Ein Prozessor ist quasi der real gewordene Traum Alan Turings, eine universelle Variante seiner Maschine zu bauen.

Eine zusätzliche Maßnahme bei Prozessoren ist, dass die auszuführenden Algorithmen nicht mehr in einer Tabelle fest verdrahtet sind. Was fest verdrahtet ist, sind grundlegende Operationen, wie z. B. einen separaten Speicher anzusprechen oder zwei Zahlen zu addieren. Jede Operation wird durch ein bestimmtes Muster von binären Symbolen bestimmt, die binären Symbole heißen seit den 50er-Jahren Bits (binary digits). Ich will im Folgenden annehmen, dass immer 8 Bits zu einer Einheit zusammengefasst werden (man spricht dann seit den 60er-Jahren von einem Byte) und dass auch ein Register 8 Bit breit ist. Auch die auszuführenden Operationen sollen 8 Bit breit sein und eine spezielle Nummer zwischen 0 (binär 00000000) und 255 (binär 11111111) zugeordnet bekommen. Der Prozessor liest nun erst einmal 8 Bits aus dem Speicher (es gibt also kein Band mehr), bestimmt die zugehörige Zahl, die dahintersteckt, und führt anhand einer internen Tabelle den dazugehörigen Befehl aus (gegebenenfalls werden dazu auch weitere Bytes eingelesen). Die Nummer, die zu einem bestimmten Befehl gehört, nennt man OP-Code.

An dieser Stelle ist Turings Traum einer universellen Rechenmaschine Wirklichkeit geworden, anstatt des Bands werden nun Speicherbausteine mit wahlfreiem Zugriff benutzt. An dieser Stelle taucht dann zum ersten Mal der Begriff *random access memory,* kurz RAM, auf. Der wahlfreie Zugriff auf das RAM macht einen Prozessor sehr leistungsfähig, weil die Zeit, die benötigt wird, um auf eine bestimmte Adresse zuzugreifen, kaum von der Größe des Speichers abhängt. Statt der Position eines Lesekopfs auf einem Band benutzt man also heute Speicheradressen. Manchmal wird die Abkürzung RAM in diesem Zusammenhang auch für *random access machine* (Maschine mit wahlfreiem Zugriff) benutzt. Diese Abkürzung ist schlicht falsch und Sie sollten diese überlesen, wenn sie z. B. in Internetforen auf-

taucht. Kommen wir aber zurück zu den Prozessoren. Moderne Prozessoren können nämlich noch viel mehr, sie können z. B. Speicheradressen direkt anspringen, um die Programmausführung an genau dieser Stelle fortzusetzen. Dies leisten spezielle Register, wie z. B. der Programmzähler PC (program counter). Das Zählregister PC zeigt immer auf die Speicheradresse des nächsten Befehls und wird auch stets automatisch aktualisiert, nachdem ein Befehl ausgeführt wurde.

Nehmen wir nun einen einfachen Prozessor, der zusätzlich zum Programmzähler ein Register besitzt, das 8 Bit breit ist. In dieses Register soll nun mit dem Befehl Nr. 1 eine 8-Bit-Zahl eingelesen werden. Schreiben wir ab jetzt Byte-Werte als Zahlen zwischen 0 und 255 und ersetzen die Bandposition durch eine Speicheradresse, die die Position des Bytes im Speicher angibt. Die Position 0 soll das erste Byte im Speicher darstellen. Nun müssen Sie für die Multiplikation von 5 mit 7 aus unserem letzten Beispiel zuerst die Zahl 5 in das Zahlenregister schreiben, das ich hier mit A abkürzen möchte. Dazu müssen Sie dem Prozessor folgende Bytes übergeben:

1,5 (Lade den Wert 5 in Register A)

Nun benötigen Sie noch zusätzliche Befehle für die Grundrechenarten. Seien die OP-Codes 2, 3, 4 und 5 für die Grundrechenarten +, –, * und / vorgesehen. Das Byte, das dem OP-Code folgt, sei der Operand des Befehls. Ein Operand ist der Teil einer mathematischen Operation, mit dem gerechnet wird, also der eigentliche Zahlenwert. Bei 5+7 wären die Operanden 5 und 7. Unser Prozessor in diesem Beispiel speichert die Operanden in Register A. Um die Multiplikation aus Abschnitt 1.2.3 auszuführen, müssen Sie nun vorher die folgenden Bytes benutzen:

1,5 (Lade den Wert 5 in Register A)

5,7 (Multipliziere A mit 7 und speichere das Ergebnis wieder in Register A)

Sie sehen an dieser Stelle schon, dass moderne Prozessoren sehr viel effizienter arbeiten als die relativ langsamen Turing-Maschinen. Liefern aber die Algorithmen, die auf modernen Prozessoren laufen, nun endlich immer die richtigen Ergebnisse? Werden nun sämtliche mathematischen Probleme im Prinzip berechenbar? Die Antwort ist leider „nein“, denn die grundlegenden Probleme in Bezug auf die Berechenbarkeit werden auch durch moderne, schnelle Prozessoren nicht gelöst. Mehr noch: Im Prinzip kann ein moderner Prozessor durch eine einfache Turing-Maschine nachgebildet werden, wenn auch die Ausführungsgeschwindigkeit der Programme eher bescheiden ist. In dem nächsten Abschnitt werde ich nun eine Methode vorstellen, mit der Sie die Laufzeit von Algorithmen abschätzen können, die in C programmiert wurden. Auch dieses Thema ist Bestandteil der Vorlesungen in Allgemeiner Informatik.

1.3 Laufzeitanalyse von Algorithmen

Die Grundlage der Komplexitäts- und Laufzeitanalyse von Algorithmen und Computerprogrammen im Allgemeinen ist das Entscheidungsproblem: Wie kann entschieden werden, ob eine bestimmte mathematisch-logische Aussage eindeutig wahr oder falsch ist und wie kann man in einer sinnvollen Zeit zu einem eindeutigen Ergebnis kommen? Und wenn z. B.

eine Berechnung mit einem plausiblen Ergebnis endet, was heißt dann in einer *sinnvollen Zeit?* Einen möglichen Ansatz haben Sie bereits kennengelernt, nämlich die Turing-Maschine. Beginnen wir nun wieder mit einem einfachen Beispiel für eine einfache Turing-Maschine. Es seien auf dem Band vor dem Start folgende Zeichen enthalten:

aaaaaaaaaaaaaabbbbbbbbbbbbbbaaaaaaaaaaaaaaa

Angenommen, es sollen alle „a" durch „z" ersetzt werden und die Maschine startet wieder in Zustand 0. Nun genügt im Endeffekt folgende Regel: Falls ein „a" gelesen wurde, schreibe ein „z" und rücke den Lesekopf um eine Position nach rechts. Das Programm verläuft nach diesen Überlegungen in folgender Schleife ab:

1. Falls ein „a" gelesen wurde, schreibe ein „z"
2. Rücke den Lesekopf um eine Position nach rechts
3. Zurück zu Schritt 1

An dieser Stelle sehen Sie schon, dass der Algorithmus tut, was er soll – nur stoppt er nicht. Es wurde nämlich nicht definiert, was mit den anderen Zeichen geschehen soll. Da das Eingabeband unendlich lang ist, wird die Turing-Maschine bis in alle Ewigkeit um eine Position weiterrücken. Auch in diesem einfachen Beispiel wird die Turing-Maschine also erst stoppen, wenn ein Endzustand definiert wird. Dies kann z. B. die Regel sein, dass die Maschine immer dann anhält, wenn sich ein leeres Feld unter dem Lesekopf befindet. Der veränderte Algorithmus stoppt in diesem Fall, wenn das erste Leerfeld auftritt. Wenn das erste Eingabezeichen unter dem Lesekopf kein Leerfeld ist, stoppt der Algorithmus nun in einer endlichen Zeit, die sich einfach berechnen lässt:

Laufzeit=K*Anzahl der Zeichen bis zum ersten Leerfeld

Die Konstante K gibt hier an, wie lange die Turing-Maschine benötigt, um einen einzelnen Schritt inklusive möglicher Zustandswechsel auszuführen. Wenn es also einen definierten Startzustand und einen definierten erreichbaren Endzustand gibt, lässt sich die Laufzeit des Algorithmus relativ einfach berechnen. Ferner gilt: Jeder Algorithmus, der auf einer Turing-Maschine für sämtliche möglichen Eingaben zu einem Ende kommt, ist auch vollständig berechenbar. Leider liegt das grundlegende Problem genau hier, nämlich bei der Aussage „für sämtliche Eingaben". Da das Eingabeband bei einer Turing-Maschine unendlich lang ist, bedeutet dies, dass für den im letzten Beispiel vorgestellten Algorithmus das Entscheidungsproblem nicht gelöst werden kann – man müsste bei beliebig langen Eingaben auch beliebig lang warten, um zu entscheiden, ob die Turing-Maschine jemals stoppt und ob sie dann eine logisch wahre, plausible und reproduzierbare Antwort liefert. Es gibt in der Mathematik viele Probleme, die sich nicht in einer *vernünftigen* Zeit berechnen lassen. Hier sind einige Beispiele:

- Das Produkt a*b sehr langer Primzahlen (ab etwa 100 Stellen) lässt sich nur sehr schwer faktorisieren, wenn man a und b nicht kennt.
- Die Entscheidung, ob eine zufällig gewählte Zahl eine Primzahl ist, wird mit wachsender Stellenanzahl schwieriger. Der Schwierigkeitsgrad wächst exponentiell mit der Stellenanzahl.
- Es gibt Funktionen, die Einwegfunktionen sind: Eine Berechnung in die eine Richtung ist sehr einfach, während die andere Richtung schwer bis unmöglich zu berechnen ist. Ein Beispiel ist das Produkt langer Primzahlen, ein anderes Beispiel ist z. B. der diskrete Logarithmus.

Sämtliche als „schwer bis unmöglich zu lösen“ deklarierten mathematischen Probleme haben die Eigenschaft, dass der Zeitaufwand der Berechnung exponentiell mit der Länge der Eingabe (z. B. der Stellen- oder Bitanzahl) wächst und Sie die Laufzeit nicht mehr durch ein wie immer geartetes Polynom der Art

$$f(x) = a_n x^n + a_{n-1} x_{n-1} + \ldots + a_0 x^0$$

ausdrücken können. Stellen Sie sich an dieser Stelle z. B. eine dreifach verschachtelte Schleife vor, die die Variablen i, j und k als Zähler benutzt. Wenn i, j und k nun jeweils den Wert 10 haben, dann wird die äußerste Schleife zehnmal, und die innerste Schleife 1000-mal durchlaufen. Die Laufzeit ergibt sich hier also aus dem Produkt von i, j und k und lässt sich allgemein als Polynom der Form

$$f(x) = ax^3$$

darstellen. Solche Probleme, bei denen sich die Laufzeit durch ein wie immer geartetes Polynom darstellen lässt, sind mit einem modernen Computer noch in einer vernünftigen Zeit berechenbar. Mathematische Algorithmen jedoch, bei denen die Laufzeit wie folgt wächst, sind kaum noch in einer vernünftigen Zeit berechenbar:

$$f(x) = a * e^{bx}$$

Was diese Aussage bedeutet, erfahren Sie in den nächsten Abschnitten.

1.3.1 Das P-NP-Problem

Bis jetzt gilt also folgende Aussage: Für mathematische Probleme, bei denen der Berechnungsaufwand exponentiell mit der Eingabelänge (gemessen in Ziffern bzw. Bits) wächst, gibt es zumindest in der nahen Zukunft trotz Einbeziehung moderner Technologien keine effiziente Lösung. Auf dieser Feststellung basieren einige wichtige Verschlüsselungsverfahren, wie z. B. RSA oder der sichere Schlüsselaustausch mit dem Diffie-Hellmann-Verfahren. In der Komplexitätstheorie gibt es drei für die Informatik wichtige Komplexitätsklassen. Diese nennt man P, EXP und NP. P umfasst dabei sämtliche Probleme, bei denen der Berechnungsaufwand polynomial mit der Länge der übergebenen Daten steigt. Der Aufwand lässt sich hier durch eine Funktion der Form

$$t(k) = n * k^b$$

darstellen. t(k) ist der Zeitaufwand in Abhängigkeit von der Eingabelänge k in Bit, b und n sind Konstanten. Bei den Problemen, die zur Komplexitätsklasse EXP gehören, wächst der Zeitaufwand exponentiell mit der Größe der zu berechnenden Zahlen. Der Aufwand lässt sich also hier durch eine Funktion der Form

$$t(k) = n * b^k$$

darstellen. Auch hier sind n und b wieder Konstanten. Die Klasse NP (nicht polynomiale Algorithmen) liegt innerhalb von EXP und umfasst P. Die Klasse NP lässt im Gegensatz zu P und EXP zusätzlich zu den ursprünglichen Turing-Maschinen auch nicht-deterministische Turingmaschinen zu. Diese Art Turingmaschinen enthält an einigen Stellen Zufallselemente, die eine genaue Vorhersage unmöglich machen. Diese Eigenschaft nennt man nicht-deterministisch. Man könnte z. B. die zuletzt genannte Ersetzung von a durch z wie folgt abändern, um eine nicht-deterministische Turing-Maschine zu erzeugen:

Falls ein a gelesen wurde, schreibe ein z mit einer Wahrscheinlichkeit von 0,8 und schreibe sonst ein y. Rücke danach den Lesekopf nach rechts.

Da sich die Probleme aus P auch nicht-deterministisch lösen lassen, ist P eine Teilmenge von NP. Zurzeit kann nicht beantwortet werden, ob P nur eine untergeordnete Teilmenge von NP ist oder ob sogar P = NP gilt. In diesem Fall (also wenn P = NP wäre) müssten sich sämtliche mathematischen Probleme in polynomialer Zeit lösen lassen, auch z. B. die Faktorisierung langer Primzahlen oder das Berechnen diskreter Logarithmen. Dies wäre ein großes Problem, weil damit lang bewährte Verfahren, wie z. B. RSA oder der sichere Schlüsselaustausch nach Diffie-Hellmann unbrauchbar würden.

1.4 Laufzeitabschätzungen von C-Programmen

Moderne Prozessoren können Algorithmen viel schneller ausführen als Turing-Maschinen und arbeiten auch sehr viel effizienter. Die Unterschiede in der Laufzeit unterscheiden sich dennoch nur um einen linearen Faktor. Auch, wenn der Geschwindigkeitszuwachs schon bei einfachen Prozessoren wie z. B. dem in den 80er-Jahren populären 6502-Mikroprozessor im Vergleich zu den Turing-Maschinen etwa dem Faktor 1000 bis 10 000 entspricht, so ist der Unterschied trotzdem nicht so gravierend, dass Sie plötzlich ganz andere mathematische Modelle verwenden müssen. Aber auch der Umstieg auf eine moderne Programmiersprache wie C ändert an der Komplexität eines Algorithmus nicht viel, deshalb können Sie statt des umständlichen Maschinencodes auch direkt die C-Programme analysieren. In den nächsten Beispielen verwende ich nun die Programmiersprache C, um Ihnen einige Grundlagen der Laufzeitanalyse vorzustellen.

Beispiel 1: Geschachtelte Schleifen

Gegeben sei das folgende C-Listing:

```
01  for (int x=0; x<n; x++)
02  {
03      for (int y=0; y<n; y++)
04      {
05          for (int z=0; z<n; z++)
06          {
07         printf("x=%d,y=%d,z=%d\n",x,y,z);
08          }
09      }
10  }
```

Sei n=10. In diesem Fall wird jede der drei geschachtelten Schleifen zehnmal durchlaufen. Die Ausgabe der Variablen x, y und z innerhalb der innersten Schleife ist für die Laufzeit des Programms relativ uninteressant, weil dieser Teil nur jeweils einmal pro Schleifendurchlauf ausgeführt wird. In diesem Beispiel ist es nicht wichtig, wie das C-Programm genau in Maschinencode aussieht. Wichtig ist nur, dass Sie sich die Schleifen genau ansehen.

Wenn Sie sich das C-Listing ansehen, sehen Sie, dass die drei Schleifen geschachtelt sind. D. h., die äußere Schleife für x wird nur zehnmal durchlaufen (denn n ist 10), die Schleife für y 100-mal und die innerste Schleife 1000-mal. In diesem Beispiel würden also 1000 Zeilen mit den jeweiligen Werten für x, y und z ausgegeben. Wenn Sie n erhöhen, so verlängern Sie natürlich auch die Laufzeit des Programms. In diesem Beispiel wächst die Laufzeit aber nicht linear mit n, sondern mit der Geschwindigkeit n*n*n. Bei n = 100 würde die innerste Schleife also schon eine millionen-mal durchlaufen. An dieser Stelle sagt man: Für die Laufzeit des Algorithmus gilt: Es ist $O(n^3)$ und die Laufzeit steigt polynomial, nämlich mit der dritten Potenz von n. Bei vier geschachtelten Schleifen gilt dann, dass $O(n^4)$ ist. Ich werde nun einige Regeln für die Laufzeitanalyse von C-Programmen angeben, die Sie sich gut einprägen sollten.

Regeln für die Laufzeitanalyse von C-Programmen:

- Allen einfachen Rechenoperationen und Funktionsaufrufen der Standardbibliothek wird O(1) zugewiesen.
- Einfache if-Ausdrücke oder case-Anweisungen erhalten O(1), wenn sie keine weiteren Schleifen enthalten.
- Einfache, nicht verschachtelte Schleifen erhalten O(n).
- Verschachtelte Schleifen erhalten $O(n^{\text{Verschachtelungstiefe}})$. D. h., wenn z. B. drei Schleifen ineinander verschachtelt werden, erhält dieser Programmteil $O(n^3)$.
- Bei Schleifen, die von einer Bedingung abhängen, wird immer der worst case betrachtet und die größtmögliche Laufzeit gewählt (es wird so getan, als ob die längst mögliche Schleife gewählt wird)
- Die geschätzte Laufzeit des gesamten Programms ist die Summe aller analysierten Funktionen der Art O(…). Bei Funktionsaufrufen werden sämtliche aufgerufenen Funktionen separat betrachtet.

Beispiel 2: Eine Worst-Case-Betrachtung

Gegeben sei das folgende C-Listing:

```
01  while (Text[i]!=0)
02  {
03      a=Text[i];
04      if (a<65)
05      {
06          for (j=0; j<20; j++)
07          {
08              for (k=0; k<20; k++)
09              {
10                  ... komplexe Berechnungen mit verschachtelter Schleife ...
11              }
12          }
13      }
14      else
15      {
16          for (j=0; j<20; j++) { ... einfache Berechnungen ... }
17      }
18      i++;
19  }
```

In diesem Beispiel wird die äußerste Schleife ausgeführt, solange der Text nicht mit einem Nullzeichen endet. Dieser Schleife wird also O(n) zugewiesen. Die erste der inneren Schleifen mit einer Tiefe von 2 wird nur ausgeführt, wenn a < 65 ist. Dies ist zwar nur für nicht-Buchstaben der Fall, kann aber trotzdem auftreten. Obwohl das Programm fast immer den else-Zweig nimmt, muss hier trotzdem der worst case betrachtet und $O(n^2)$ für die beiden inneren Schleifen angenommen werden. Es gilt also: $O(n*n^2) = O(n^3)$, wobei n die Anzahl der Zeichen im Text darstellt.

Beispiel 3: Wort-Case-Betrachtung mit Benutzereingaben

Gegeben sei das folgende C-Listing:

```
01  void ClearText(char *Text, long int Len)
02  {
03      for (long int i=0; i<Len; i++)
04      {
05          Text[i]=0;
06      }
07  }
08
09  int main (void)
10  {
11      char C=0;
12      long int i=0;
13      char Text[10000];
14      do
15      {
16          C=getch( );
17          switch (C)
18          {
19                case '$F1KEY$':
20                    ClearText(Text,i);
21                    i=0;
22                break;
23                default:
24                    Text[i]=C;
25                    printf("%c",C);
26                    i++;
27                break;
28             }
29      }
30      while (C!='$ESCAPEKEY$')
31      Text[i]=0;
32      printf("%s\n",Text);
33      return(0);
34  }
```

In dem letzten Beispiel wächst die Laufzeit des Programms mit der Textlänge, der äußeren Schleife wird O(n) zugewiesen. Der worst case ist sicherlich, dass der Benutzer mehrere Texte eingibt und am Ende immer F1 statt ESC drückt. In diesem Fall müsste die Funktion `ClearText()` immer wieder aufgerufen werden, und zwar für jeden Text, den der Benutzer eingibt und mit F1 statt mit ESC abschließt. Dies würde so lange geschehen, bis der Benutzer die Geduld verliert. In diesem Fall müsste die Laufzeit $O(n^2)$ betragen, denn die einfache Schleife, die `ClearText()` ausführt, muss separat betrachtet werden und ist deshalb als in der äußeren Schleife geschachtelt zu betrachten.

Leider stoßen Sie bei diesem Programm auf zwei Probleme. Das erste Problem ist die Funktion `getch()`, die so lange wartet, bis eine Taste gedrückt wird – und wenn es ewig dauert. Auf jeden Fall erschlägt die Verzögerung durch `getch()` die Laufzeit sämtlicher anderer Schleifen, weshalb die Laufzeit auch vor allem von der Tippgeschwindigkeit des Benutzers abhängt. Im besten Fall (ohne Fehlbedienung) wächst die Laufzeit nur linear mit der Textlänge und folgt dann O(n). Das zweite Problem ist der Benutzer selbst, der einfach das Programm missverstehen kann. Statt „Drücke ESC zum Beenden und F1 zum Löschen des Textes" liest der Benutzer „Drücke F1 zum Beenden und ESC zum Löschen des Textes". Es gibt nicht wenige Programme, die deswegen abstürzen und auf diese Weise quasi unendlich lange Laufzeiten haben.

Beispiel 4: Einfache Grafikausgabe

Gegeben sei das folgende C-Listing:

```
01  void DrawRectangle(long int y1, long int x2, long int y2, long int Color)
02  {
03      long int x,y;
04      if (x1>x2) { Swap(x1,x2); }
05      if (y1>y2) { Swap(y1,y2); }
06      for (y=y1; y<y2; y++)
07      {
08          for (x=x1; x<x2; x++)
09          {
10            SetPixel(x,y,Color);
11          }
12      }
13  }
```

Die Funktion `DrawRectangle()` zeichnet ein Rechteck von (x1,y1) nach (x2,y2). Die Vertauschung der Koordinaten wird nur aufgerufen, falls versehentlich z. B. x1 > x2 und/oder y1 > y2 ist. Die Vertauschungsfunktion `Swap()` bekommt also O(1) zugewiesen. Die Ausgabe des Rechtecks auf dem Bildschirm wird durch eine verschachtelte Schleife realisiert, die jedes Pixel einzeln setzt. Die Laufzeit erhöht sich also mit der Größe des Rechtecks und folgt $O(n^2)$.

Ich habe nun die wichtigsten Grundlagen so weit behandelt, dass ich Ihnen in den nächsten Kapiteln die ersten einfachen Algorithmen zeigen kann. An dieser Stelle muss ich noch einen Hinweis für Einsteiger einfügen: Dieses Buch ist kein C-Buch und auch kein Buch über die Grundlagen der Programmierung. Es werden hier also keine Themen, wie z. B. Variablendeklaration, Schleifen, bedingte Anweisungen, Strukturen oder Arrays besprochen. Diese Themen sind Bestandteil anderer Bücher, und natürlich Ihrer Vorlesungen in Programmierung. Wenn Sie noch nicht programmieren können, ist also dieses Buch nichts für Sie – zumindest nicht, bevor Sie sich die grundlegenden Kenntnisse angeeignet haben. Selbstverständlich können Sie dieses Buch als Studienbegleiter sehen, und dieses je nach Bedarf Kapitel für Kapitel durcharbeiten.

1.5 Übungen

Übung 1

Geben Sie die Zustandstabelle einer Turing-Maschine an, mit der Sie eine Zahl b von einer Zahl a subtrahieren können. Hierbei soll stets b<a sein und es soll die längencodierte Zahlendarstellung verwendet werden. Die Zahlen a und b sollen durch genau eine Null getrennt werden und am Ende der Eingabe soll eine 2 stehen.

Übung 2

Geben Sie die Zustandstabelle einer Turing-Maschine an, mit der Sie zwei Wörter zu einem Wort vereinigen können. Aus „Haus“ und „Tür“ wird also „Haustür“. Die zwei Wörter sollen vorher durch genau ein Leerzeichen getrennt sein und die Eingabedaten sollen durch eine 2 abgeschlossen werden. Am Ende sollen die Wörter zu einem einzigen Wort (mit nur einem Großbuchstaben am Anfang) vereinigt worden sein und durch eine 2 abgeschlossen werden.

Übung 3

Überlegen Sie, wie Sie mit einer einfachen Turingmaschine zwei Nibbels (inklusive Übertrag) addieren können. Ein Nibbel ist ein halbes Byte und Sie können damit Zahlenwerte zwischen 0 und 15 darstellen. Benutzen Sie hierfür am besten die Hexadezimalschreibweise und schreiben die Symbole 0 – 9 als Zahlen, sowie a für 10 und f für 15.

Übung 4

Überlegen Sie sich, wie Sie mit C ein kleines Simulationsprogramm für eine einfache Turing-Maschine erstellen können, mit dem Sie Ihre Ergebnisse aus den Übungen überprüfen können. Verwenden Sie für die Zustandstabelle ein globales Array und für den aktuellen Zustand eine globale Variable.

Hinweis

Sämtliche Lösungen zu den Übungsaufgaben befinden sich im Anhang.

2 Basisalgorithmen

Im letzten Kapitel wurde die Frage „Was ist ein Algorithmus?“ ausführlich beantwortet. Auf diese Frage musste ich Ihnen zuerst eine Antwort liefern, bevor Sie nun fortfahren können.

Ich möchte Ihnen in diesem Kapitel einige grundlegende Algorithmen vorstellen, die immer wieder in der Informatik auftauchen. Ich nenne diese Algorithmen „Basisalgorithmen“, weil sie die Grundlage darstellen, auf der alle komplexeren Verfahren aufbauen. Stellen Sie sich an dieser Stelle ein solides Fundament vor. Genauso, wie ein Haus ohne Fundament zusammenbricht, brechen komplexe Algorithmen in sich zusammen, wenn die Basis nicht richtig implementiert wird. Dies geschieht schneller, als Sie denken und betrifft sogar erfahrene Programmierer. Wenn Sie z. B. den Tausch zweier Zahlen nicht richtig umsetzen, handeln Sie sich unter Umständen schwer aufzufindende Bugs ein. Ein *Bug* ist in der Informatik ein Fehler, der ein Programm zu einem Verhalten veranlasst, das der Programmierer nicht vorhergesehen hat. Dies kann ein falsches Ergebnis oder ein Absturz sein. Sie können jedoch die Anzahl Bugs in Ihren Programmen minimieren, wenn Sie sich einen guten Werkzeugkoffer zulegen (in diesen gehören natürlich die Basisalgorithmen auf jeden Fall hinein). Wie jeder gute Handwerker werden Sie dann mit fortschreitender Praxiserfahrung und mit einem stetig umfangreicheren Werkzeugkoffer ein immer besserer Programmierer werden. Dies setzt natürlich (wie alles, was man erst lernen muss) stetige Übung voraus, und genau solche Übungsbeispiele für Ihren Studienalltag möchte ich Ihnen in diesem Buch vorstellen. Ich versuche also, möglichst nah an den Programmierpraktika zu bleiben und keine abgehobenen mathematischen Theorien oder ähnliche Dinge zu behandeln, die nicht zu lauffähigen Programmen führen. Für den Bereich „Theoretische Informatik“ benötigen Sie also ein anderes Buch.

Ich werde nun damit beginnen, in jedem Unterpunkt einen Basisalgorithmus behandeln und am Ende ein C-Listing für diesen Algorithmus angeben. Ich verwende an dieser Stelle C, weil auch im Studium die Basisalgorithmen meistens in der Programmiersprache C vorgestellt werden. Auch, wenn im Studium immer öfter Java eingesetzt wird, unterstützt diese Programmiersprache jedoch eine wichtige Sache nicht: die Verwendung von Zeigern, die für dieses Kapitel sehr wichtig sind. An dieser Stelle gilt wieder: Dieses Buch ist kein Programmierhandbuch, sondern setzt voraus, dass Sie an den entsprechenden Programmierpraktika teilnehmen, sich mit der Verwendung von Zeigern und indirekter Adressierung auskennen und ferner in der Lage sind, PAPs (Programmablaufpläne) zu lesen.

2.1 Der Ringtausch

Bevor ich die ersten lauffähigen Programme anführe, muss ich noch ein paar Dinge über die Formatierung des Textes sagen. Ich habe Programmierbegriffe, die in den Text eingebunden sind, besonders `hervorgehoben`, z. B. bei `printf()`. Typenangaben wie z. B. `int` sind Programmierelemente und werden so hervorgehoben wie z. B. `printf()`. Ferner habe ich Variablennamen und Zeilennummern **fett** hervorgehoben, so können Sie z. B. (wie es z. B. später bei den verketteten Listen der Fall ist) nicht das in den Text eingebundene Wort Nachfolger mit dem Variablennamen **Nachfolger** verwechseln. Ich setze an dieser Stelle wieder voraus, dass Sie sich mit C auskennen und wissen, was ein Array ist und wie man eine Variable deklariert. Ich setze auch voraus, dass Sie wissen, was ein Zeiger ist und wie Sie diesen in C verwenden. Wenn Sie gerade erst mit dem Studium begonnen haben, kann es sein, dass Sie diese Dinge noch nicht wissen. Dies macht aber nichts, denn in diesem Fall können Sie dieses Buch einfach dann weiterlesen, wenn Sie Zeiger und Arrays behandelt haben. Genauso können Sie auch mit den anderen Kapiteln verfahren und diese genau dann lesen und durcharbeiten, wenn Sie diese benötigen. Sie müssen dieses Buch also nicht von vorne nach hinten lesen wie einen Roman (vor allem hätte ich dann einfach einen Roman geschrieben, wenn ich diesen Anspruch an Sie hätte).

Kommen wir nun zurück zum Ringtausch. Vertauschungen kommen sehr oft vor, z. B. bei Sortierverfahren. Auch, wenn einige höhere Programmiersprachen, wie z. B. Java, Sortierverfahren quasi schon „drin“ haben, so kann es trotzdem hilfreich sein, den Ringtausch zu verstehen. Fangen wir nun erst einmal sehr einfach an und deklarieren zwei Variablen, nämlich **a** und **b**. Diese Variablen seien vom Typ `int`. Der hier behandelte Ringtausch funktioniert folgendermaßen: Es wird zuerst eine temporäre Variable **temp** erstellt, der der Wert von **a** zugewiesen wird. Anschließend wird `a=b` gesetzt. Die Werte von **a** und **b** sind nun identisch und haben den Wert von **b**. Genau hier benötigen Sie die temporäre Variable, denn Sie setzen nun `b=temp`. Nun ist a=b und b=a, die beiden Werte wurden vertauscht. Sehen Sie sich nun Listing 2.1 an:

Listing 2.1 ringtausch.c

```
01  #include<stdio.h>

02  void ringtausch(int *a, int *b)
03  {
04      int *temp=a;
05      a=b; b=temp;
06  }

07  int main(void)
08  {
09      int a=47, b=11;
10      ringtausch(&a,&b);
11      printf("a=%d,b=%d\n");
12      return 0;
13  }
```

In Zeile **01** binden Sie zunächst **stdio.h** ein, damit Sie Funktionen wie `printf()` benutzen können. Anschließend folgt in den Zeilen **02** – **06** die Funktion `ringtausch()`, der Sie zwei

Parameter (**a** und **b**) als Zeiger übergeben. Die Verwendung von Zeigern ist hier deshalb wichtig, weil Sie direkt mit den Speicheradressen der Variablen arbeiten müssen. Andernfalls würden nur Kopien der übergebenen Parameter benutzt und die veränderten Variablen wären außerhalb der Funktion `ringtausch()` nicht mehr sichtbar. Die Funktion `ringtausch()` führt nun den Algorithmus aus, der hier am Anfang beschrieben wurde: Erst wird **a** in der Variablen **temp** zwischengespeichert, anschließend wird `a=b` gesetzt. Zum Schluss wird der in der Variablen **temp** gesicherte Wert nach **b** übertragen.

Sie haben an dieser Stelle vielleicht schon den Verdacht gehabt, dass hier eigentlich nur die Speicheradressen von **a** und **b** vertauscht werden, nicht die Werte selbst. Dies ist richtig: Es werden nur die Zeiger vertauscht, aber genau dies ist an dieser Stelle auch gewollt. Wenn Sie nämlich für den Ringtausch von Anfang an Zeiger verwenden, können Sie sogar lange Zeichenketten vertauschen, ohne sämtliche Daten vorher in einen Puffer kopieren zu müssen. An einer Stelle müssen Sie jedoch aufpassen: Weil für die Übergabeparameter der Funktion `ringtausch()` Zeiger benutzt werden, müssen Sie im Hauptprogramm entsprechend in Zeile **10** die Variablen **a** und **b** mit dem Address-Of-Operator „&“ übergeben und so gewährleisten, dass hier wirklich die Speicheradressen und nicht die Werte selbst benutzt werden. Das Programm gibt bei der Ausgabe Folgendes in der Konsole aus:

```
a=11, b=47
```

Leider können Sie Zeichenketten nicht auf die Weise vertauschen, wie dies in Listing 2.1 geschehen ist. Der Grund hierfür ist, dass Zeichenketten `char`-Arrays sind, von denen nur die Anfangsadressen in der Variablentabelle abgelegt werden. Sie können aber den Ringtausch so abwandeln, dass Sie mit diesem auch Zeichenketten vertauschen können, ohne sämtliche Zeichen einzeln kopieren zu müssen. Hierzu müssen Sie, genau wie im ersten Beispiel, die Zeiger vertauschen. Dies erreichen Sie bei Zeichenketten durch einen doppelten Zeiger vom Typ `char**`. Sehen Sie sich dazu Listing 2.2 an:

Listing 2.2 ringtausch_strings.c

```
01  #include<stdio.h>

02  void ringtausch_st(char **a, char **b)
03  {
04      char *temp=*a;
05      *a=*b;
06      *b=temp;
07  }

08  int main(void)
09  {
10      char *a="Hallo";
11      char *b="Welt";
12      printf("%s %s\n",a,b);
13      ringtausch(&a,&b);
14      printf("%s %s\n",a,b);
15      return 0;
16  }
```

Die Funktion `ringtausch_st()` (das „st“ steht für „String“) bekommt nun zwei Parameter (**a** und **b**) vom Typ `char**` übergeben. Hier wird also ein doppelter Zeiger benutzt, nämlich ein Zeiger, der die Anfangsadresse eines Eintrags in der Variablentabelle enthält, der wie-

derum auf die Anfangsadresse eines Strings zeigt (Zeichenketten werden oft als *Strings* bezeichnet, ich werde diese Bezeichnung ab jetzt auch benutzen). Wenn Sie nun **a** und **b** vertauschen, vertauschen Sie auch in diesem Beispiel wieder die Zeigeradressen in der Variablentabelle.

Dies funktioniert aber nur, wenn Sie in Zeile **04** die Variable **temp** als String (also als `char*`) deklarieren und dieser anschließend einen Zeiger auf die Variable **a** (also `a*`) zuweisen. Mit einer Deklaration von **temp** ohne Zeigertyp und einer Zuweisung wie `temp=a` würden Sie nur auf das erste Zeichen der Zeichenkette a zugreifen, nicht aber auf die Zeigereinträge in der Variablentabelle. Deshalb wird auch in Zeile **05** `*a=*b` und nicht `a=b` gesetzt, sowie in Zeile **06** `*b=temp` und nicht `b=temp`. Aber Vorsicht: **temp** ist schon ein Zeiger, also wäre `*b=*temp` falsch und würde zu einem Compilerfehler der Art „CANNOT CONVERT CHAR* TO CHAR**“ führen.

Das Hauptprogramm funktioniert nun ähnlich wie in Listing 2.1. Zuerst werden in Zeile **10** und **11** die zwei String-Variablen **a** und **b** deklariert, **a** bekommt die Zeichenkette „Hallo“ und **b** die Zeichenkette „Welt“ zugewiesen. Bei der ersten Ausgabe mit `printf()` in Zeile **12** werden **a** und **b** durch Leerzeichen getrennt ausgegeben, also wird „Hallo Welt“ ausgegeben. In Zeile **13** werden nun **a** und **b** getauscht und in Zeile **14** wird noch einmal **a** und **b** durch ein Leerzeichen getrennt ausgegeben. Das Programm gibt bei der Ausführung also Folgendes aus:

Hallo Welt

Welt Hallo

Hinweis: Benutzung von Zeigern beim Ringtausch

Beim Ringtausch müssen Sie sehr achtsam mit den entsprechenden Zeiger-Variablen umgehen, sonst können Sie sich schwer zu findende Fehler einhandeln. Besonders, wenn Sie eine allgemeine Schutzverletzung erhalten oder Ihr Programm abstürzt, sollten Sie Ihre Zeigerdeklarationen nochmal dahingehend prüfen, ob sämtliche Zeiger auf eine gültige Adresse zeigen.

Als abschließendes Beispiel zum Ringtausch möchte ich Ihnen nun zeigen, wie Sie Strukturvariablen miteinander vertauschen können. Dieses Verfahren taucht immer wieder im Studium auf. Eine Standardanwendung ist z. B. das Sortieren von Personendaten mit einem wie immer gearteten Sortierverfahren. Ich werde auf Sortierverfahren noch sehr detailliert eingehen, eine Sache ist jedoch immer wichtig: Sie sollten beim Sortieren stets auf die Performance achten und es vermeiden, Daten hin und her zu kopieren. Deshalb benutzt auch das folgende Beispielprogramm Zeiger. Sehen Sie sich dazu Listing 2.3 an:

Listing 2.3 Ringtausch_structs.c

```
01  #include<stdio.h>
02  #include<string.h>

03  typedef struct
04  {
05      char Anrede[5];
```

```
06      char Vorname[50];
07      char Nachname[50];
08      int Alter;
09  } Person_t;

10  void ringtausch(Person_t *a, Person_t *b)
11  {
12      Person_t temp=*a;
13      *a=*b;
14      *b=temp;
15  }

16  int main(void)
17  {
18      Person_t a,b;
19      strcpy(a.Anrede,"Herr");
20      strcpy(a.Vorname,"Herbert");
21      strcpy(a.Nachname,"Meyer");
22      a.Alter=42;
23      strcpy(b.Anrede,"Frau");
24      strcpy(b.Vorname,"Erika");
25      strcpy(b.Nachname,"Mustermann");
26      b.Alter=46;
27      ringtausch(&a,&b);
28      printf("Person 1:\n");
29      printf("Name:%s %s %s\n",a.Anrede,a.Vorname,a.Nachname);
30      printf("Alter:%d\n",a.Alter);
31      printf("Person 2:\n");
32      printf("Name:%s %s %s\n",b.Anrede,b.Vorname,b.Nachname);
33      printf("Alter:%d\n",b.Alter);
34      return 0;
35  }
```

In den Zeilen **01** und **02** wird zusätzlich zu **stdio.h** noch **string.h** eingebunden. Diese Header-Datei ermöglicht es Ihnen, Funktionen wie `strcpy()` zu benutzen. In den Zeilen **03** – **09** wird nun der strukturierte Datentyp `Person_t` deklariert. Hierzu müssen Sie zusammen mit dem Schlüsselwort `struct` auch `typedef` verwenden. Nur auf diese Weise gewährleisten Sie, dass der strukturierte Datentyp `Person_t` auch wirklich als ein eigener Typ betrachtet wird, der auch stets bei der Variablendeklaration den entsprechenden Speicher zugewiesen bekommt. Wenn Sie einen C++-Compiler verwenden, ist das `typedef` nicht unbedingt nötig, da C++ `struct` immer automatisch um ein `typedef` ergänzt.

Die Funktion `ringtausch()` funktioniert fast wie die Funktion `ringtausch()` im ersten Beispiel. Die Parameter **a** und **b** werden als Zeiger übergeben und sind vom Typ `Person_t`. Sie können aber nun nicht einfach `Person_t *temp=a` statt `int *temp=a` schreiben, wie im ersten Beispiel. Strukturierte Datentypen werden nämlich ähnlich abgespeichert, wie Strings. Es wird hier also nur die Anfangsadresse des Datenblocks der entsprechenden Struktur in der Variablentabelle abgelegt, weshalb Sie in Zeile **12** durch `Person_t temp=*a` angeben müssen, dass Sie **temp** eine Zeigeradresse zuweisen wollen. Probieren Sie es ruhig aus und benutzen Sie z. B. `Person_t *temp=a` bzw. `Person_t *temp=a`. Sie erhalten in beiden Fällen Fehlermeldungen in der Art „CANNOT CONVERT PERSON_T TO PERSON_T*“.

Auch in Zeile **13** und **14** müssen Sie mit Zeigern arbeiten, deshalb schreiben Sie `*a=*b` statt `a=b` und `*b=temp`. Auch hier wäre `*b=*temp` falsch. In diesem Fall würden nämlich am Ende

des Tauschvorgangs **a** und **b** auf die gleiche Speicheradresse zeigen. Wenn Sie dann Glück haben, gibt Ihr Compiler eine entsprechende Warnmeldung aus, dass Sie versuchen, nicht kompatible Typen ineinander umzuwandeln.

Im Hauptprogramm werden nun zwei Variablen (**a** und **b**) vom Typ `Person_t` deklariert und anschließend in den Zeilen **18**–**26** mit entsprechenden Werten gefüllt. Hier benötigen Sie die Funktion `strcpy()`, um die Struktur-Variablen-Elemente **Anrede**, **Vorname** und **Nachname** entsprechend zu initialisieren. Am Ende haben Sie zwei Personen angelegt. Person **a** ist Herr Herbert Meyer (42), Person **b** ist Frau Erika Mustermann (46). Nun wird in Zeile **27** der Ringtausch von **a** und **b** ausgeführt und anschließend werden die Personendaten durch `printf()` ausgegeben. Nun ist Person **a** Erika Mustermann und Person **b** Herbert Meyer. Das Programm gibt bei der Ausführung Folgendes aus:

Person 1:

Name:Frau Erika Mustermann

Alter:46

Person 2:

Name:Herr Herbert Meyer

Alter:42

2.2 Einfache Textsuche

Es gibt zahlreiche komplexe Algorithmen, um z. B. ein bestimmtes Wort in einem Text zu finden. Auf einige dieser Algorithmen werde ich später noch detaillierter eingehen. Da komplexe und geschwindigkeitsoptimierte Suchverfahren aber oft gar nicht gebraucht werden, fange ich nun mit einem sehr einfachen Verfahren an, dass Sie in nur einer einzigen Praktikumsstunde umsetzen können. Die Grundidee einer einfachen Textsuche ist recht simpel. Es sei eine Variable vom Typ `char*` gegeben, die den zu durchsuchenden Text enthält. Der Einfachheit halber heißt die Variable **Text** und wird wie folgt initialisiert:

```
char *Text="Ich bin ein einfacher Text. Suche bitte in mir nach einem Wort.";
```

Nun benötigen Sie ein Suchmuster, das durch folgende Variable deklariert wird:

```
char *P="bitte";
```

Hinweis: Der Variablenname P bei Suchalgorithmen

Der Variablenname **P** taucht in vielen Suchalgorithmen immer wieder auf. Der Name „P“ bedeutet „pattern“ und ist englisch für „Muster“. Die Variablennamen **TextPos** (aktuelle Textposition) und **PatternPos** (aktuelle Leseposition im Muster) werden auch oft verwendet, vor allem in Internetforen.

Suchen wir nun nach dem Vorkommen des ersten Zeichens des Wortes „bitte“ in unserem Text. Dies kann durch die folgende einfache Schleife realisiert werden:

```
TextPos=0;
while ((Text[TextPos]!=P[0])&&(Text[TextPos]!=0))
{
    TextPos++;
}
```

Falls nun das Wort „bitte“ oder ein anderes Wort, das mit „b“ anfängt, im Text enthalten ist, dann ist irgendwann die Bedingung `Text[TextPos]==P[0]` erfüllt (`true`). In diesem Fall trifft die Bedingung, dass die Schleife irgendwann „aussteigt“, weil sie am Textende angelangt ist, nicht zu. Sie prüfen dann in diesem Fall (also, wenn das Textende noch nicht erreicht ist) in einer zweiten `while`-Schleife, ob nun das vollständige Muster **P** folgt, und brechen die `while`-Schleife vorzeitig ab, wenn dies nicht der Fall ist. Intelligenterweise benutzen Sie hierfür auch wieder die Variable **TextPos**:

```
PatternPos=0;
while ((Text[TextPos]==P[PatternPos]) &&(P[PatternPos]!=0))
{
    TextPos++; PatternPos++;
}
```

Wenn die `while`-Schleife abbricht und `P[PatternPos]==0` ist, dann wurde das gesamte Muster **P** im Text gefunden. Dies ist so, weil sämtliche C-Strings mit einem Null-Byte enden. Falls die `while`-Schleife abbricht und `P[PatternPos]!=0` ist, traf die erste Bedingung zu, nämlich dass vorher im Text ein Zeichen aufgetreten ist, das nicht mehr mit den Zeichen im Suchmuster übereinstimmt. Genialerweise enthält dann **TextPos** genau die Position, an der das Programm weitersuchen muss, wenn das Wort „bitte“ dann doch nicht gefunden wurde. In diesem Beispiel ist dies an der sechsten Position der Fall (diese ist in eckigen Klammern angegeben):

Text="Ich bi[n] ein einfacher Text. Suche bitte in mir nach einem Wort.";

Bei einem zweiten Durchlauf der `while`-Schleife zum Suchen nach dem Muster **P** wird dann das Wort „bitte“ gefunden. An dieser Stelle können Sie dann die Suche abbrechen und z. B. von der entsprechenden Suchfunktion die Position zurückgeben lassen, an der zum ersten Mal das Wort „bitte“ auftritt. Sehen Sie sich hierzu nun Listing 2.4 an. Ich empfehle Ihnen auch, sich hierfür zusätzlich den entsprechenden Programmablaufplan anzusehen.

Listing 2.4 Mustersuche.c

```
01  #include<stdio.h>

02  int Textsuche(char *Text, char *P)
03  {
04      bool gefunden;
05      int TextPos,PatternPos,temp;
06      gefunden=false; TextPos=0; PatternPos=0;
07      while ((gefunden==false)&&(Text[TextPos]!=0))
08      {
09          while ((Text[TextPos]!=P[0])&&(Text[TextPos]!=0))
```

```
10          {
11              TextPos++;
12          }
13          PatternPos=0; temp=TextPos; // Zähler zurücksetzen
14          while ((Text[TextPos]==P[PatternPos])&&(P[PatternPos]!=0))
15          {
16              TextPos++; PatternPos++; // Immer ein Zeichen weiter
17          }
18          if (P[PatternPos]==0) // Die Suche war erfolgreich
19          {
20              gefunden=true;
21          }
22      }
23      if (gefunden==true) { return temp; }
24      else { return -1; }
25  }

26  int main(void)
27  {
28      int Pos;
29      char P[100];
30      char *Text="Ich bin ein einfacher Text. Suche bitte in mir nach einem Wort.";
31      printf("Text:%s\n",Text);
32      printf("Zu suchendes Wort:"); scanf("%s",&P);
33      Pos=Textsuche(Text,P);
34      if (Pos>=0) { printf("Das Wort '%s' wurde an Position %d gefunden.\n",P,Pos); }
35      else { printf("Das zu suchende Wort wurde nicht gefunden.\n"); }
36      return 0;
37  }
```

Für dieses Programm benötigen Sie nur die Bibliothek **stdio.h**, die Sie in Zeile **01** einbinden. Danach folgt die Funktion `Textsuche()`, der zwei Parameter vom Typ `char*` übergeben werden müssen, nämlich der zu durchsuchende Text und das Suchmuster **P**. Zuerst müssen in den Zeilen **04** – **06** einige Variablen deklariert werden. Dies ist einmal die Variable **gefunden** (Typ `bool`), die einen Wahrheitswehrt enthält, der angibt, ob das gesuchte Muster bereits gefunden wurde. Ferner werden noch die Zähler-Variablen **TextPos**, **PatternPos** und **temp** benötigt. **TextPos** ist zu Beginn der Suche 0 und zeigt auf den ersten Buchstaben des zu durchsuchenden Textes, **gefunden** ist am Anfang `false`. In Zeile **07** folgt nun die erste `while`-Schleife für den Haupt-Suchvorgang, die so lange wiederholt wird, bis entweder **gefunden** wahr wird oder das Textende-Zeichen (das Nullbyte) erreicht wird.

Hinweis: Verwendung von C++ statt C

Wenn Sie das vorige Beispiel-Listing auf einem C++-Compiler eingeben, dann können Sie den Variablentyp `bool` ohne Weiteres benutzen. Ansonsten müssen Sie vor allem bei ANSI-C-Compilern entweder **stdbool.h** einbinden oder aber folgende Typendefinition an den Anfang Ihres Programms stellen:

```
typedef enum {false,true} bool;
```

Nun folgt in den Zeilen **0** – **12** die erste Suchschleife, die Sie schon aus der Einführung kennen – nämlich die Suche nach dem ersten Zeichen des Suchmusters. Wenn dieses Zeichen gefunden wird, ohne vorher das Textende zu erreichen, wird in den Zeilen **14** – **17** die zweite Suchschleife ausgeführt. Da die zweite Suchschleife die Variable **TextPos** verändert, muss diese vorher in der Variablen **temp** gesichert werden. Die zweite Suchschleife überprüft nun, ob ab der aktuellen Position **TextPos** das vollständige Suchmuster **P** folgt. Ist dies der Fall, so bricht die zweite Suchschleife ab und `P[PatternPos]` ist 0, woraufhin gefunden auf wahr (`true`) gesetzt wird. Ist allerdings `P[PatternPos]` nicht 0, so bleibt gefunden falsch (`false`) und die Mustersuche wird ab der Position **TextPos** fortgesetzt.

Die Funktion `Textsuche()` kann nun auf zwei Arten beendet werden. Entweder gefunden behält den Wert `false`, was bedeutet, dass das Suchmuster nicht gefunden wurde. In diesem Fall wird in Zeile **24** der Wert –1 zurückgegeben. Wurde das Suchmuster gefunden, wird in Zeile **23** die Position zurückgegeben, an der das Suchmuster zum ersten Mal auftrat. Diese Position wurde zuvor in Zeile **13** in der Variablen **temp** gesichert.

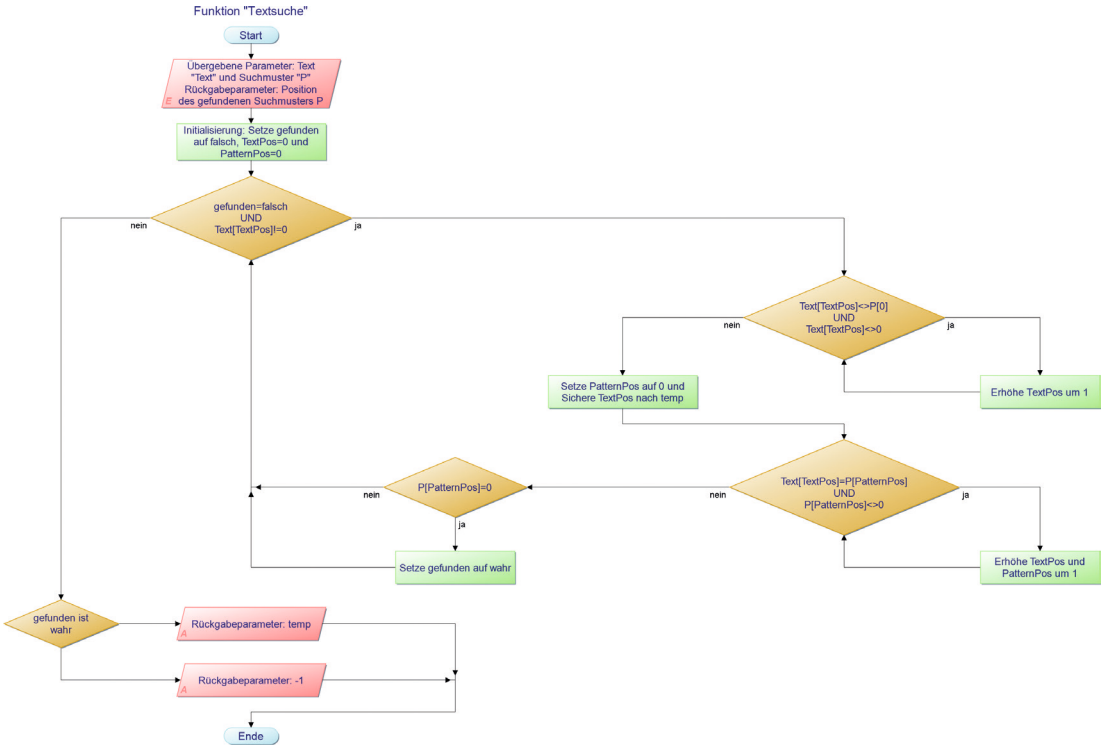

Bild 2.1 Programmablaufplan der Funktion „Textsuche“

Das Hauptprogramm habe ich sehr einfach gehalten. Zuerst werden in den Zeilen **28** – **30** die Variable **Pos** (Rückgabewert der Funktion `Textsuche()`), die Variable **P** (für bis zu 100 Zeichen eines Suchmusters) und der zu durchsuchende Text selbst initialisiert. Anschließend wird in Zeile **31** der zu durchsuchende Text ausgegeben und der Benutzer wird zur Eingabe des zu suchenden Wortes aufgefordert. Anschließend wird die Funktion `Textsuche()` aufgerufen. Gibt `Textsuche()` eine positive Zahl zurück, so wird diese Position in der Kon-

sole ausgegeben. Gibt `Textsuche()` –1 zurück, so wurde das vorher eingegebene Wort nicht gefunden und es wird ein entsprechender Hinweis ausgegeben. Das Programm gibt bei der Ausführung z. B. Folgendes aus:

```
/home/rene/Listings/Kapitel 2>./wortsuche
Text:Ich bin ein einfacher Text. Suche bitte in mir nach einem Wort.
Zu suchendes Wort:bitte
Das Wort 'bitte' wurde an Position 34 gefunden.
/home/rene/Listings/Kapitel 2>./wortsuche
Text:Ich bin ein einfacher Text. Suche bitte in mir nach einem Wort.
Zu suchendes Wort:Ich
Das Wort 'Ich' wurde an Position 0 gefunden.
/home/rene/Listings/Kapitel 2>./wortsuche
Text:Ich bin ein einfacher Text. Suche bitte in mir nach einem Wort.
Zu suchendes Wort:Haus
Das zu suchende Wort wurde nicht gefunden.
```

Das letzte Beispiel arbeitet tadellos, jedoch gibt es ein kleines Problem: Was ist, wenn ein Wort mehrmals im Text vorkommt? In diesem Fall müsste Ihre Funktion direkt mit der übergebenen Textposition (quasi als Startposition) arbeiten, aber trotzdem noch die Position zurückliefern, an der Ihr Suchmuster auftritt. Dies ist aber nicht allzu schwierig, denn hierzu müssen Sie die Variable **TextPos** nur wie folgt per Referenz (also mit dem Address-Of-Operator) übergeben:

```
int Textsuche(char *Text, char *P, int &TextPos);
```

Bei einer erfolgreichen Suche zeigt nun **TextPos** direkt hinter das gefundene Muster, sodass Sie direkt danach eine neue Suche starten können. Bei einer erfolglosen Suche zeigt **TextPos** auf das Textende und `Textsuche()` gibt den Wert -1 zurück.

Tipp: Die richtige Verwendung des Address-Of-Operators

Der Address-Of-Operator „&“ ist nur bei C++ identisch mit einem Zeiger. Dies liegt daran, dass C und C++ unterschiedlich mit den indirekten Adressierungsmodi Ihres Prozessors umgehen – dies detailliert zu erklären, würde allerdings zu weit führen. Wenn Sie einen reinen C-Compiler verwenden und hierbei Probleme mit der Variante `int &TextPos` bekommen, müssen Sie die Variable **TextPos** durch `int *TextPos` übergeben und Ihre Funktion notfalls entsprechend anpassen. Da es sehr viele verschiedene Compiler gibt, kann ich Ihnen an dieser Stelle leider keine genauen Anweisungen geben. Studieren Sie auf jeden Fall Ihr Compiler-Handbuch oder die entsprechenden Howtos. Am wenigsten Probleme werden Sie haben, wenn Sie einen modernen Compiler verwenden, der mindestens den C18-Standard unterstützt. Bei Microsoft Visual Studio ist dies auf jeden Fall gegeben. Unter Linux (dies gilt auch für den Pi) müssen Sie dem gcc-Compiler eventuell das Compiler-Flag -std=c18 übergeben, um C18 zu benutzen. Ein Beispiel hierzu wäre:

```
cc wortsuche.c -o wortsuche -std=c18
```

2.3 Einfaches Suchen und Ersetzen

Suchen und Ersetzen, auf Englisch „search and replace“, und Kopieren und Einfügen, auf Englisch „copy and paste“, sind die Grundlagen jedes modernen Texteditors. Es gibt einige sehr effiziente Lösungen für diese Probleme, nur sind die entsprechenden Algorithmen für einen Anfänger kaum zu bewältigen. Deshalb stelle ich Ihnen nun ein Verfahren vor, dass Sie „mal eben“ eingeben können, wenn Sie im Praktikum eine entsprechende Aufgabe bekommen und nur wenig Zeit für die Lösung haben.

Nehmen wir nun als Test folgenden Ausgangstext:

```
char *Text="Einfaches * und Ersetzen";
```

In diesem Text soll das Sternchen durch das Wort „Suchen“ ersetzt werden. Um dies zu erreichen, können Sie jetzt die folgenden, einfach umzusetzenden Schritte programmieren:

1. Entfernen des Sternchens: Kopieren sämtlicher Zeichen des Strings Text bis zu der Position, an der das Sternchen steht, nach unten (das Sternchen muss dabei korrekt überschrieben werden) und Merken der letzten Position des Sternchens.
2. Einfügen von sechs freien Stellen: Kopieren sämtlicher Zeichen des Strings Text ab der Position, an der das Sternchen zuletzt stand, um sechs Positionen nach oben.
3. Einfügen des Wortes „Suchen“: Kopieren des Wortes „Suchen“ an die Stelle, an der zuletzt das Sternchen stand.

Das hier erläuterte Verfahren ist sehr einfach und hat außerdem den Vorteil, dass Sie nach der Umsetzung noch andere grundlegende Funktionen zur Verfügung haben, wie z. B. das Löschen von Textpassagen oder das Einfügen von Leerzeichen. Zusammen mit der einfachen Textsuche aus dem letzten Abschnitt haben Sie dann grundlegende Funktionen zur Verfügung, um z. B. einen einfachen Texteditor zu erstellen. Selbstverständlich können Sie mit den hier vorgestellten Basisalgorithmen keine Geschwindigkeitsrekorde brechen, trotzdem haben Sie aber schon einmal einen guten Werkzeugkoffer zur Verfügung, mit dem Sie 75 Prozent der Praktikumsaufgaben und Übungen meistern können. Ich möchte Ihnen nun die Schritte, die für das einfache Suchen und Ersetzen notwendig sind, vorstellen. Am Ende steht dann wieder ein vollständiges C-Programm zum Suchen und Ersetzen eines Wortes durch ein anderes.

2.3.1 Entfernen eines Textes aus einer Zeichenkette

Sei unser Text nun der folgende String:

```
char *Text="Einfaches * und Ersetzen";
```

Angenommen, Sie kennen bereits die Position, an der das Sternchen steht (das ist die Position 10, die Zählung beginnt bei Arrays und Strings ab 0). Nun müssen Sie sämtliche Zeichen ab Position 10 um eine Position nach unten kopieren. Vielleicht denken Sie jetzt an folgende Schleife:

```
int i=0, StartPos=10, Anzahl=1;
int L=strlen(Text);
for (i=StartPos; i<L; i++)
{
       Text[i]=Text[Anzahl+i];
}
```

Sie haben richtig gedacht, diese Schleife ist genau das, was Sie brauchen, und in dem hier vorgestellten Verfahren wird sogar das String-Ende-Zeichen mitkopiert. Sie müssen also nichts Weiteres mehr beachten.

Tipp: Der richtige Umgang mit Zeichenketten

Beachten Sie, dass Zeichenketten von C wie Arrays behandelt werden, in denen jedes einzelne Zeichen durch einen Index angesprochen werden kann. Dieser Index fängt bei 0 und nicht bei 1 an. Achten Sie ferner darauf, auch wirklich jede Zeichenkette mit einem Nullzeichen abzuschließen und hierfür auch den entsprechenden Speicher anzulegen. Beispielsweise können in dem Array `char Text[10]` nur neun Textzeichen abgelegt werden, da hier spätestens die Position 9 das Nullzeichen enthalten muss. An dieser Stelle sage ich immer: „Strings sind fast Arrays, aber eben nur fast." Natürlich gilt diese Aussage nicht für Java, denn dort werden Strings durch eine separate Klasse mit separaten Methoden verwaltet, die viel mächtiger sind, als die hier vorgestellten Basisalgorithmen. Vor allem das Ersetzen von Teilstrings durch andere ist in Java quasi schon fest eingebaut. Trotzdem kann es hilfreich sein, zu verstehen, wie Java intern arbeitet, wenn es Strings verarbeitet.

2.3.2 Einfügen von Freiräumen in den Text

Wenn Sie Freiräume in den Text einfügen wollen, müssen Sie sämtliche Zeichen nach oben kopieren, in diesem Beispiel also ab der Position, an der zuletzt das Sternchen stand, um sechs Positionen nach oben. Vielleicht kommen Sie jetzt auf die Idee, die Schleife aus dem letzten Punkt wie folgt abzuwandeln:

```
int i=0, StartPos=10, Anzahl=6;
int L=strlen(Text);
for (i=StartPos; i<L; i++)
{
    Text[Anzahl+i]=Text[i];
}
```

Wenn Sie das vorige Programmfragment zur Ausführung gebracht haben und auf die Idee kommen, sich das Zwischenergebnis in der Konsole ausgeben zu lassen, werden Sie allerdings merken, dass hier etwas gewaltig schiefläuft. Es kommt wahrlich nicht das heraus, was Sie sich vorgestellt haben. Der Text lautet nun:

Einfaches ErsetzErsetz

Wenn Sie etwas nachdenken, werden Sie aber schnell merken, dass Sie die Zeichen von der falschen Seite aus kopieren und einige Zeichen überschreiben, noch bevor sie korrekt kopiert werden können. Sie müssen also die Schleife für das Kopieren nach oben wie folgt abwandeln:

```
for (i=L; i>=StartPos; i--)
{
    Text[Anzahl+i]=Text[i];
}
```

2.3.3 Ein vollständiges Programm zum Suchen und Ersetzen

Nun haben Sie alles zusammen, um ein bestimmtes Wort durch ein anderes zu ersetzen. Das folgende Beispielprogramm liest von der Tastatur einen Text und anschließend zwei Wörter ein. Anschließend wird in dem Text das erste Wort durch das zweite ersetzt. Wird das erste Wort nicht gefunden, wird eine Fehlermeldung ausgegeben. Um das Listing möglichst einfach zu halten, wird das gesuchte Wort nur einmal im Text ersetzt, nämlich an der Position, an der es zum ersten Mal auftritt, und der Programmablauf/Programmablaufplan der Funktion `Suchen()` ist identisch mit dem aus dem letzten Beispiel.

Listing 2.5 suchen_und_ersetzen.c

```
01  #include<stdio.h>
02  #include<stdlib.h>
03  #include<string.h>

04  long int Suchen(char *Text, char *P)
05  {
06      bool gefunden;
07      long int TextPos,PatternPos,temp;
08      gefunden=false; TextPos=0; PatternPos=0;
09      while ((gefunden==false)&&(Text[TextPos]!=0))
10      {
11          while ((Text[TextPos]!=P[0])&&(Text[TextPos]!=0))
12          {
13              TextPos++;
14          }
15          PatternPos=0; temp=TextPos; // Zähler zurücksetzen
16          while ((Text[TextPos]==P[PatternPos])&&(P[PatternPos]!=0))
17          {
18              TextPos++; PatternPos++; // Immer ein Zeichen weiter
19          }
20          if (P[PatternPos]==0) // Die Suche war erfolgreich
21          {
22              gefunden=true;
23          }
24      }
25      if (gefunden==true) { return temp; }
26      else { return -1; }
27  }

28  void Entfernen(char *Text, long int StartPos, long int Anzahl)
```

```
{
    long int i,L;
    L=strlen(Text);
    for (i=StartPos; i<L; i++)
    {
        Text[i]=Text[Anzahl+i];
    }
}

void Einfuegen(char *Text, long int StartPos, long int Anzahl)
{
    long int i=0,L=strlen(Text);
    for (i=L; i>=StartPos; i--)
    {
        Text[Anzahl+i]=Text[i];
    }
}

void Ueberschreiben(char *Text, char *P, long int StartPos, long int L)
{
    long int i=0;
    for (i=0; i<L; i++)
    {
        Text[StartPos+i]=P[i];
    }
}

void Ersetzen(char *Text, char *Wort1, char *Wort2, long int Pos)
{
    Entfernen(Text,Pos,strlen(Wort1));
    Einfuegen(Text,Pos,strlen(Wort2));
    Ueberschreiben(Text,Wort2,Pos,strlen(Wort2));
}

int main(void)
{
    long int Pos;
    char Puffer[100];
    char *Text,*Wort1,*Wort2;
    printf("Text:"); fgets(Puffer,100,stdin);
    Text=(char*)malloc(2*strlen(Puffer));
    Puffer[strlen(Puffer)-1]=0; strcpy(Text,Puffer); // newline entfernen
    printf("Wort:"); fgets(Puffer,100,stdin);
    Wort1=(char*)malloc(strlen(Puffer)+1);
    Puffer[strlen(Puffer)-1]=0; strcpy(Wort1,Puffer); // newline entfernen
    printf("Ersetzen durch:"); fgets(Puffer,100,stdin);
    Wort2=(char*)malloc(strlen(Puffer)+1);
    Puffer[strlen(Puffer)-1]=0; strcpy(Wort2,Puffer); // newline entfernen
    Pos=Suchen(Text,Wort1);
    if (Pos!=-1)
    {
        Ersetzen(Text,Wort1,Wort2,Pos);
        printf("%s\n",Text);
    }
    else
    {
        printf("Das gesuchte Wort kann nicht gefunden werden.\n");
    }
    return 0;
}
```

In Zeile **01** – **03** müssen Sie zusätzlich zu **stdio.h** noch **stdlib.h** und **string.h** einfügen, damit Sie im Hauptprogramm die Funktionen `malloc()` und `strcpy()` benutzen können. Die Zeilen **04** – **27** enthalten nichts Neues, sondern implementieren nur die Wort-Suchfunktion, die Sie schon aus dem letzten Beispiel kennen.

In Zeile **28** – **36** wird nun das Entfernen von Zeichen aus einer Zeichenkette in die Funktion `Entfernen()` ausgelagert. `Entfernen()` bekommt drei Parameter übergeben, nämlich den Ausgangstext als String (Typ `char*`), die Startposition als ganze Zahl (Typ `long int`) und die Anzahl Zeichen, die ab der Startposition entfernt werden sollen. Ich verwende hier den Typ `long int`, damit Sie später auch Texte mit einer Länge von mehr als 65535 Zeichen verarbeiten können (auf manchen (älteren) Systemen ist der Typ `int` nur 16 Bit breit). Die Schleife, die von der Funktion `Entfernen()` zum Löschen von Zeichen verwendet wird, ist identisch mit der Kopierschleife aus Abschnitt 2.3.1 .

In Zeile **37** – **44** wird nun das Einfügen von freien Positionen in eine Zeichenkette in die Funktion `Einfuegen()` ausgelagert. `Einfuegen()` bekommt drei Parameter übergeben, nämlich den Ausgangstext als String (Typ `char*`), die Startposition als ganze Zahl (Typ `long int`) und die Anzahl Zeichen, die ab der Startposition „freigeschaufelt“ werden sollen. Auch hier verwende ich den Typ `long int`, damit Sie später auch Texte mit einer Länge von mehr als 65535 Zeichen verarbeiten können. Die Schleife, die von der Funktion `Einfuegen()` zum Reservieren von neuen Zeichen verwendet wird, ist identisch mit der Kopierschleife aus Abschnitt 2.3.2 . Beachten Sie, dass die Funktion `Einfuegen()` keine neuen Zeichen in die ihr übergebene Zeichenkette schreibt, sondern lediglich sämtliche Zeichen ab einer gewissen Position nach oben verschiebt. Deswegen müssen Sie auch im Hauptprogramm mit `malloc()` vorher genug Speicher für die Variable **Text** reservieren (in diesem Fall das Doppelte der Eingabepuffer-Größe), damit der spätere Zeichen-Kopierprozess nicht an Speichermangel scheitert.

Da die Funktion `Einfuegen()` nur die entsprechenden Zeichen nach oben kopiert, benötigen Sie noch die Funktion `Ueberschreiben()`, die in Zeile **45** – **52** definiert wird. `Ueberschreiben()` macht nichts anderes, als **L** Zeichen des Musters **P** in die Zeichenkette **Text** zu kopieren, die Sie im ersten Parameter übergeben haben. Die Startposition wird durch den Parameter **StartPos** übergeben. Die Funktion `Ersetzen()`, die in Zeile **53** – **58** definiert wird, ruft dann lediglich die Funktionen `Entfernen()`, `Einfuegen()` und `Ueberschreiben()` in der richtigen Reihenfolge auf.

Tipp: Verwendung von Umlauten im Quellcode ist oft nicht möglich

In C und auch in anderen Programmiersprachen wie Java gibt es einen Ausführungszeichensatz und einen Ausgabezeichensatz für die Ausgabe von Text in der Konsole. Deshalb werden Umlaute in der Konsole oft richtig dargestellt, in Funktionsnamen dürfen sie aber nicht vorkommen, weil Umlaute im Ausführungszeichensatz nicht enthalten sind. Deswegen heißt die entsprechende Funktion in dem letzten Beispiel auch nicht `Überschreiben()`, sondern `Ueberschreiben()`.

Kommen wir nun zum Hauptprogramm. Zuerst werden die entsprechenden Variablen deklariert, die Sie benötigen: Dies ist einmal die Variable **Text** vom Typ `char*` sowie die

Variable **Wort1** (zu suchendes Wort) und **Wort2** (das Wort, wodurch **Wort1** ersetzt werden soll) vom selben Typ. Um zu erreichen, dass das letzte Beispiel beliebig lange Texte verarbeiten kann, habe ich nur für die Größe des Eingabepuffers einen festen Wert gewählt und reserviere den Speicher für die Variablen **Text**, **Wort1** und **Wort2** dynamisch mit `malloc()`. Ferner lasse ich bei der Eingabe der Texte Leerzeichen zu und verwende `fgets()` zusammen mit einem Eingabepuffer anstatt `scanf()`. Deshalb sieht das Einlesen des zu durchsuchenden Textes so aus:

```
printf("Text:"); fgets(Puffer,100,stdin);
Text=(char*)malloc(2*strlen(Puffer));
Puffer[strlen(Puffer)-1]=0; strcpy(Text,Puffer); // newline entfernen
```

Sie lesen also die Variable **Text** mit `fgets()` von der Standardeingabe ein und reservieren anschließend mit `malloc()` für die Variable **Text** neuen Speicher, dessen Größe der doppelten Länge des Eingabepuffers entspricht. Nach Ersetzen des **newline**-Zeichens am Ende des Eingabepuffers durch das Nullzeichen wird der Eingabepuffer mit `strcpy()` in den Zielstring kopiert. Mit den anderen Tastatureingaben verfahren Sie auf dieselbe Weise. Der Vorteil dieses eher umständlichen Vorgehens ist die hohe Flexibilität, die Sie damit erreichen. So können Sie in dem letzten Beispiel durchaus auch ein Wort durch mehrere Wörter ersetzen.

Tipp: Verwenden von Unicode-Zeichen (z. B. unter Windows 10)

Wenn Sie Unicode-Zeichen verwenden, dann belegt ein Zeichen zwei Bytes Speicher. Wenn Sie sich nicht sicher sind, ob Sie ein Unicode-System benutzen, wandeln Sie die `malloc()`-Anweisungen wie folgt ab:

```
printf("Text:"); fgets(Puffer,100,stdin); // Zeichen
Text=(char*)malloc(2*sizeof(char)*strlen(Puffer)); // Bytes
Puffer[strlen(Puffer)-1]=0; strcpy(Text,Puffer); // Zeichen
```

2.4 Einfaches Sortieren von Zahlen

Es gibt sehr effiziente Sortieralgorithmen, die ich Ihnen natürlich auch später noch vorstellen möchte. Nur sind diese Algorithmen alles andere als einfach gestrickt und gehören deswegen nicht zu den Basisalgorithmen, die Sie mal eben umsetzen können. Dagegen können Sie mit dem nun vorgestellten Sortierverfahren Bubble Sort ein Zahlenarray mit bis zu 1000 Zahlen relativ zügig sortieren. Die Betonung liegt hier auf relativ, denn bei mehr als etwa 1000 Zahlen wird Bubble Sort spürbar langsam. Trotzdem sollten Sie Bubble Sort in Ihrem Werkzeugkoffer haben, denn oft ist es überhaupt nicht nötig, große Zahlenkolonnen zu sortieren, und Sie benötigen dagegen eine Methode, die Sie z. B. in einem einzelnen Praktikum umsetzen können.

2.4.1 Bubble Sort[1]

Bubble Sort als der einfachste unter den Sortieralgorithmen arbeitet nach einer sehr simplen Methode: Das zu sortierende Zahlenarray, das ich hier durch die Variable **Zahlen** repräsentiere, wird wiederholt von vorne nach hinten durchlaufen. Hierzu dient ein einfacher Positionszähler, den ich hier **i** nenne. Angenommen, Sie wollen das Array aufsteigend sortieren. Dann wird das entsprechende Array-Element an der Zähler-Position **i** immer dann mit dem benachbarten Array-Element an der Position **i+1** vertauscht, wenn der Wert dieses Elements größer ist als der des Nachbarelements. Angenommen, das Array hat **n** Elemente. Dann durchlaufen Sie das Array im ersten Schritt mit folgender Schleife:

```
for (i=0; i<n-1; i++)
{
    if (Zahlen[i]>Zahlen[i+1])
    {
        tausche(Zahlen,i,i+1); // Ringtausch
    }
}
```

Weitere Durchläufe durch Ihr Zahlen-Array führen Sie so lange aus, bis keine Vertauschungen mehr vorgenommen werden müssen. Die Prüfung, ob noch Vertauschungen nötig sind, können Sie z. B. mit einem Zähler erreichen, der vor jedem Durchlaufen des Zahlen-Arrays auf 0 zurückgesetzt wird und immer dann um 1 erhöht wird, wenn eine Vertauschung nötig ist. Sie können dazu die obige Schleife wie folgt abwandeln:

```
Zaehler=0;
for (i=0; i<n-1; i++)
{
    if (Zahlen[i]>Zahlen[i+1]) { tausche(Zahlen,i,i+1); Zaehler++; }
}
```

Wahlweise können Sie auch eine Variable vom Typ `bool` verwenden, die vor jedem Durchlauf des Zahlen-Arrays auf `false` gesetzt wird und immer dann auf `true` gesetzt wird, wenn doch noch eine Vertauschung nötig ist. Wie Sie vorgehen, ist Geschmackssache, ich bevorzuge jedoch die Variante mit der Variablen vom Typ `bool`. Dies tue ich deswegen, weil Zähler gerne mal überlaufen und dadurch schwer auffindbare Fehler verursacht werden können.

Schauen Sie sich nun das folgende Zahlen-Array mit zehn Zahlen an:

```
int Zahlen[10]={13,44,12,78,42,43,78,19,54,60};
```

Im ersten Durchlauf werden dann folgende Vertauschungsschritte ausgeführt:

- Die Zahl 44 wird mit der Zahl 12 vertauscht.
- Die Zahl 78 wird mit der Zahl 42 vertauscht.
- Die Zahl 78 wird anschließend noch einmal mit der Zahl 43 vertauscht.
- Die Zahl 79 wird mit der Zahl 19 vertauscht.

[1] *https://de.wikipedia.org/wiki/Bubblesort*