ADITYA Y. BHARGAVA

ALGORITHMEN KAPIEREN

VISUELL LERNEN UND VERSTEHEN

mit Illustrationen, Alltagsbeispielen und Python-Code

mitp

Für meine Eltern Sangeeta und Yogesh

Aditya Y. Bhargava

Algorithmen kapieren

Visuell lernen und verstehen

mit Illustrationen, Alltagsbeispielen und Python-Code

Übersetzung aus dem Amerikanischen
von Knut Lorenzen

mitp

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.

ISBN 978-3-95845-814-7
1. Auflage 2019

www.mitp.de
E-Mail: mitp-verlag@sigloch.de
Telefon: +49 7953 / 7189 - 079
Telefax: +49 7953 / 7189 - 082

Übersetzung der amerikanischen Originalausgabe
Aditya Y. Bhargava: Grokking Algorithms
ISBN 978-1617292231

Lektorat: Sabine Schulz
Sprachkorrektorat: Simone Fischer
Coverbild und Grafiken im Buch: Leslie Haimes
Satz: III-satz, Husby, www.drei-satz.de

Inhaltsverzeichnis

Vorwort 11

Einleitung 13
Überblick 14
Verwendung dieses Buchs 15
Wer sollte dieses Buch lesen? 15
Konventionen und Downloads 16

Über den Autor 16

Danksagungen 17

1 **Einführung in Algorithmen** 19
1.1 Einführung 19
1.1.1 Performance 20
1.1.2 Problemlösungen 20
1.2 Binäre Suche 21
1.2.1 Eine bessere Suchmethode 23
✎ Übungen 27
1.2.2 Laufzeit 28
1.3 Landau-Notation 29
1.3.1 Die Laufzeiten von Algorithmen nehmen unterschiedlich schnell zu 29
1.3.2 Visualisierung verschiedener Laufzeiten 32
1.3.3 Die Landau-Notation beschreibt die Laufzeit im Worst Case 33

1.3.4 Typische Laufzeiten gebräuchlicher Algorithmen 34
✎ Übungen 35
1.3.5 Das Problem des Handlungsreisenden 36
1.4 Zusammenfassung 38

2 Selectionsort 39
2.1 Die Funktionsweise des Arbeitsspeichers 40
2.2 Arrays und verkettete Listen 42
2.2.1 Verkettete Listen 43
2.2.2 Arrays 44
2.2.3 Terminologie 45
✎ Übung 46
2.2.4 Einfügen in der Mitte einer Liste 47
2.2.5 Löschen 48
✎ Übungen 49
2.3 Selectionsort 51
Beispielcode 55
2.4 Zusammenfassung 56

3 Rekursion 57
3.1 Rekursion 58
3.2 Basisfall und Rekursionsfall 61
3.3 Der Stack 62
3.3.1 Der Aufruf-Stack 63
✎ Übung 66
3.3.2 Der Aufruf-Stack mit Rekursion 66
✎ Übung 70
3.4 Zusammenfassung 70

4 Quicksort 71
4.1 Teile und herrsche 72
✎ Übungen 79
4.2 Quicksort 80
4.3 Landau-Notation im Detail 85
4.3.1 Mergesort und Quicksort im Vergleich 86
4.3.2 Average Case und Worst Case im Vergleich 88
✎ Übungen 92
4.4 Zusammenfassung 92

5 Hashtabellen 93
5.1 Hashfunktionen 96
✎ Übungen 99

5.2 Anwendungsfälle . . . 100
5.2.1 Hashtabellen zum Nachschlagen verwenden . . . 100
5.2.2 Doppelte Einträge verhindern . . . 102
5.2.3 Hashtabellen als Cache verwenden . . . 104
5.2.4 Zusammenfassung . . . 107
5.2.5 Kollisionen . . . 107
5.3 Performance . . . 110
5.3.1 Der Auslastungsfaktor . . . 112
5.3.2 Eine gute Hashfunktion . . . 114
✎ Übungen . . . 114
5.4 Zusammenfassung . . . 115

6 Breitensuche . . . 117
6.1 Einführung in Graphen . . . 118
6.2 Was ist ein Graph? . . . 120
6.3 Breitensuche . . . 121
6.3.1 Den kürzesten Pfad finden . . . 124
6.3.2 Warteschlangen . . . 126
✎ Übungen . . . 127
6.4 Implementierung des Graphen . . . 127
6.5 Implementierung des Algorithmus . . . 130
6.5.1 Laufzeit . . . 135
✎ Übung . . . 135
6.6 Zusammenfassung . . . 138

7 Der Dijkstra-Algorithmus . . . 139
7.1 Anwendung des Dijkstra-Algorithmus . . . 140
7.2 Terminologie . . . 145
7.3 Eintauschen gegen ein Klavier . . . 147
7.4 Negativ gewichtete Kanten . . . 154
7.5 Implementierung . . . 157
✎ Übung . . . 167
7.6 Zusammenfassung . . . 168

8 Greedy-Algorithmen . . . 169
8.1 Das Stundenplanproblem . . . 169
8.2 Das Rucksackproblem . . . 172
✎ Übungen . . . 174
8.3 Das Mengenüberdeckungsproblem . . . 174
8.3.1 Approximationsalgorithmen . . . 175
✎ Übungen . . . 181

8.4 NP-vollständige Probleme . . . 181
8.5 Das Problem des Handlungsreisenden – Schritt für Schritt . . . 183
8.5.1 Wie lassen sich NP-vollständige Probleme erkennen? . . . 187
✎ Übungen . . . 189
8.6 Zusammenfassung . . . 189

9 Dynamische Programmierung . . . 191
9.1 Das Rucksackproblem . . . 191
9.1.1 Die einfache Lösung . . . 192
9.1.2 Dynamische Programmierung . . . 193
9.2 Häufig gestellte Fragen zum Rucksackproblem . . . 201
9.2.1 Was geschieht beim Hinzufügen eines Gegenstands? . . . 201
✎ Übung . . . 204
9.2.2 Was geschieht, wenn die Reihenfolge der Zeilen geändert wird? . . . 204
9.2.3 Kann man das Gitter auch spaltenweise (statt zeilenweise) befüllen? . . . 205
9.2.4 Was geschieht, wenn man ein leichteres Objekt hinzufügt? . . . 205
9.2.5 Kann man Teile eines Gegenstands stehlen? . . . 206
9.2.6 Optimierung des Reiseplans . . . 206
9.2.7 Handhabung voneinander abhängiger Objekte . . . 208
9.2.8 Ist es möglich, dass die Lösung mehr als zwei Teil-Rucksäcke erfordert? . . . 209
9.2.9 Ist es möglich, dass die beste Lösung den Rucksack nicht vollständig füllt? . . . 209
✎ Übung . . . 209
9.3 Der längste gemeinsame Teilstring . . . 210
9.3.1 Erstellen des Gitters . . . 211
9.3.2 Befüllen des Gitters . . . 212
9.3.3 Die Lösung . . . 213
9.3.4 Die längste gemeinsame Teilfolge . . . 214
9.3.5 Die längste gemeinsame Teilfolge – Lösung . . . 216
✎ Übung . . . 217
9.4 Zusammenfassung . . . 217

10 k-nächste Nachbarn . . . 219
10.1 Klassifikation von Orangen und Grapefruits . . . 219
10.2 Entwicklung eines Empfehlungssystems . . . 221
10.2.1 Merkmalsextraktion . . . 223
✎ Übungen . . . 227
10.2.2 Regression . . . 227

10.2.3 Auswahl geeigneter Merkmale 230
✎ Übung 230
10.3 Einführung in Machine Learning. 231
10.3.1 OCR 231
10.3.2 Entwicklung eines Spamfilters 232
10.3.3 Vorhersage der Entwicklung des Aktienmarkts 233
10.4 Zusammenfassung 233

11 Die nächsten Schritte 235
11.1 Bäume 235
11.2 Invertierte Indizes 238
11.3 Die Fourier-Transformation 239
11.4 Nebenläufige Algorithmen 240
11.5 MapReduce 241
11.5.1 Warum sind verteilte Algorithmen nützlich? 241
11.5.2 Die map-Funktion 242
11.5.3 Die reduce-Funktion 242
11.6 Bloom-Filter und HyperLogLog 243
11.6.1 Bloom-Filter 245
11.6.2 HyperLogLog 245
11.7 Die SHA-Algorithmen. 246
11.7.1 Dateien vergleichen 246
11.7.2 Passwörter überprüfen. 247
11.8 Locality-Sensitive Hashing 248
11.9 Diffie-Hellman-Schlüsselaustausch 249
11.10 Lineare Programmierung 250
11.11 Epilog 251

Lösungen zu den Übungen 253
Kapitel 1 253
Kapitel 2 254
Kapitel 3 257
Kapitel 4 258
Kapitel 5 259
Kapitel 6 260
Kapitel 7 262
Kapitel 8 263
Kapitel 9 264
Kapitel 10 265

Stichwortverzeichnis 267

Vorwort

Anfangs war Programmieren für mich einfach nur ein Hobby. Die Grundlagen erlernte ich mit dem Buch *Visual Basic 6 für Dummies* und ich las weitere Bücher, um mehr zu erfahren. Das Thema Algorithmen war für mich allerdings undurchschaubar. Ich kann mich noch gut daran erinnern, dass ich mir das Inhaltsverzeichnis meines ersten Lehrbuchs zu diesem Thema zu Gemüte führte und dachte: »Endlich werde ich das Ganze mal verstehen!« Der Inhalt war jedoch so kompakt und informationsreich, dass ich die Lektüre nach einigen wenigen Wochen aufgab. Erst als ich auf einen wirklich guten Professor für Algorithmen traf, wurde mir klar, wie einfach und elegant die zugrunde liegenden Ideen tatsächlich sind.

Vor einigen Jahren verfasste ich meinen ersten bebilderten Blogbeitrag. Ich kann am besten lernen, wenn mir der Stoff visuell präsentiert wird, daher gefielen mir die illustrierten Beiträge besonders gut. Seitdem habe ich selbst einige bebilderte Beiträge über funktionale Programmierung, Git, Machine Learning und Nebenläufigkeit geschrieben. Ich war anfangs übrigens nur ein mittelmäßiger Autor. Technische Begriffe zu erklären ist schwierig. Es erfordert viel Zeit, gute Beispiele zu finden und komplizierte Konzepte zu erklären, daher ist es am einfachsten, die schwierigen Dinge unter den Teppich zu kehren. Ich dachte eigentlich, ich würde die Sache ganz gut machen, nachdem sich einer meiner Artikel ziemlich weit verbreitete. Bis ein Kollege zu mir kam und sagte: »Ich habe deinen Artikel gelesen, aber das Thema noch immer nicht verstanden.« Ich musste noch viel über das Schreiben lernen.

Während ich eine Reihe dieser Blogbeiträge verfasste, kam der Manning-Verlag auf mich zu und fragte, ob ich Lust hätte, ein illustriertes Buch zu verfassen. Es stellte sich heraus, dass die Redakteure des Verlags sich hervorragend mit dem Erklären technischer Konzepte auskannten und sie zeigten mir, wie man Wissen

vermittelt. Ich habe dieses Buch verfasst, weil mir eine Sache besonders am Herzen lag: Ich wollte ein Buch schreiben, das schwierige technische Themen gut erklärt, ein leicht verständliches Buch über Algorithmen. Meine Fähigkeit zu schreiben hat sich seit meinem ersten Blogbeitrag weiterentwickelt und ich hoffe, dass dieses Buch eine leicht verständliche und informative Lektüre ist.

Einleitung

Dieses Buch soll leicht verständlich sein, deshalb vermeide ich große Gedankensprünge. Bei der Vorstellung eines neuen Konzepts erkläre ich es entweder sofort oder weise darauf hin, wann es erläutert wird. Kernkonzepte werden durch Übungen und mehrfache Erklärungen vertieft, sodass du deine eigenen Schlussfolgerungen überprüfen kannst und dich auf diese Weise vergewisserst, dass du dem Inhalt folgst.

Ich verwende stets Beispiele. Ich möchte keinen Zeichensalat präsentieren, sondern habe zum Ziel, dass es dir leicht fällt, die Konzepte zu visualisieren. Ich bin davon überzeugt, dass man am besten lernt, wenn man sich an bereits Bekanntes erinnern kann – und Beispiele machen es einfacher, sich etwas zu merken. Wenn du dir beispielsweise den Unterschied zwischen Arrays und verketteten Listen (die in Kapitel 2 erläutert werden) merken möchtest, brauchst du nur daran zu denken, in einem Kinosaal Platz zu nehmen. Auch wenn ich Gefahr laufe, das Offensichtliche zu verkünden: Ich bin ein visueller Lerner. Dieses Buch enthält also haufenweise Abbildungen.

Der Inhalt des Buchs wurde sorgfältig zusammengestellt. Für ein Buch, das sämtliche Sortieralgorithmen abhandelt, gibt es keinen Bedarf – zu diesem Zweck gibt es die Wikipedia und die Khan Academy. Alle vorgestellten Algorithmen sind praktisch anwendbar. Bei meiner Tätigkeit als Softwareentwickler haben sie sich als nützlich erwiesen und bilden eine gute Grundlage für komplexere Themen. Viel Vergnügen beim Lesen!

Überblick

Die ersten drei Kapitel des Buchs behandeln die Grundlagen:

- **Kapitel 1** – Du lernst den ersten praxisnahen Algorithmus kennen: die binäre Suche. Außerdem erfährst du, wie man die Geschwindigkeit eines Algorithmus mithilfe von Landau-Symbolen (engl. *Big-O-Notation*) analysiert. Diese Notation wird im gesamten Buch verwendet, um zu beschreiben, wie langsam oder wie schnell ein Algorithmus arbeitet.
- **Kapitel 2** – Dieses Kapitel behandelt zwei fundamentale Datenstrukturen: Arrays und verkettete Listen. Diese beiden Datenstrukturen werden im gesamten Buch verwendet und dienen dazu, komplexere Datenstrukturen wie Hashtabellen (siehe Kapitel 5) zu erstellen.
- **Kapitel 3** – In diesem Kapitel geht es um die Rekursion. Hierbei handelt es sich um ein praktisches Verfahren, das von vielen Algorithmen verwendet wird (wie z. B. Quicksort, das in Kapitel 4 zur Sprache kommt).

Meiner Erfahrung nach sind die Landau-Notation und Rekursion für Einsteiger ziemlich anspruchsvolle Themen. Deshalb lasse ich es langsam angehen und widme diesen beiden Abschnitten zusätzlichen Raum. Die verbleibenden Kapitel stellen Algorithmen mit einem breiten Spektrum von Anwendungsmöglichkeiten vor:

- **Problemlösungsverfahren** – Die Kapitel 4, 8 und 9 behandeln Problemlösungsverfahren. Wenn du einer Aufgabe gegenüberstehst und dir nicht sicher bist, wie sie effizient gelöst werden kann, solltest du das Teile-und-herrsche-Verfahren (siehe Kapitel 4) oder dynamische Programmierung (siehe Kapitel 9) ausprobieren. Möglicherweise stellst du jedoch fest, dass es keine effiziente Lösung gibt. In diesem Fall kannst du einen Greedy-Algorithmus (siehe Kapitel 8) verwenden, um eine Näherungslösung zu berechnen.
- **Hashtabellen** – Kapitel 5 hat Hashtabellen zum Thema. Eine Hashtabelle ist eine äußerst nützliche Datenstruktur, die Schlüssel- und Werte-Paare enthält, wie beispielsweise den Namen einer Person und deren E-Mail-Adresse oder einen Benutzernamen und das dazugehörige Passwort. Der Nutzen von Hashtabellen ist so groß, dass er kaum überbewertet werden kann. Wenn ich eine Aufgabe in Angriff nehme, stelle ich zunächst die folgenden beiden Fragen: »Kann ich eine Hashtabelle verwenden?« und »Kann ich das als Graph darstellen?«.
- **Graphenalgorithmen** – Die Kapitel 6 und 7 befassen sich mit Graphenalgorithmen. Graphen bieten die Möglichkeit, ein Netzwerk zu modellieren: ein soziales Netzwerk, ein Netzwerk aus Straßen oder Neuronen oder irgendeine andere aus Verbindungen bestehende Menge. Die Breitensuche (engl. *Breadth-First Search*, kurz BFS, siehe Kapitel 6) und der Dijkstra-Algorithmus (siehe Kapitel 7) ermöglichen es, die kürzeste Verbindung zwischen zwei Punkten eines Netzwerks zu finden. Du kannst diesen Ansatz beispielsweise dazu verwenden, um die Verschiedenartigkeit zweier Personen oder die kürzeste Verbindung zu einem Ziel zu berechnen.

- **k-nächste Nachbarn (KNN)** – In Kapitel 10 geht es um k-nächste Nachbarn, einen einfachen Machine-Learning-Algorithmus. Mit KNN kann beispielsweise ein Empfehlungssystem, eine optische Zeichenerkennung (engl. *Optical Character Recognition*, OCR) oder ein System zur Vorhersage von Aktienkursen erstellt werden – also Systeme, die irgendeine Vorhersage treffen (»Der Besucher bewertet diesen Film mit 4 Sternen«) oder Objekte klassifizieren (»Dieser Buchstabe ist ein Q«).
- **Die nächsten Schritte** – In Kapitel 11 werden 10 weitere Algorithmen vorgestellt, die gut geeignet sind, um das Thema Algorithmen weiter zu vertiefen.

Verwendung dieses Buchs

Die Reihenfolge der Inhalte und der Inhalt selbst wurden sorgfältig zusammengestellt. Wenn du an einem Thema besonders interessiert bist, steht es dir natürlich frei, einen Teil des Buchs zu überspringen. Andernfalls solltest du die Kapitel jedoch der Reihe nach lesen, da sie aufeinander aufbauen.

Ich empfehle ausdrücklich, dass du den Code der Beispiele auch tatsächlich ausführst. Ich kann es gar nicht oft genug wiederholen. Gib die Codebeispiele genau wie angegeben ein (oder lade sie unter `https://github.com/egonschiele/grokking_algorithms`) herunter und führe sie aus. Wenn du das machst, wirst du sehr viel mehr behalten.

Darüber hinaus rate ich auch dazu, die Übungen zu bearbeiten. Sie sind nicht zeitaufwändig – in der Regel dauert es nur ein oder zwei Minuten, vielleicht auch mal fünf oder zehn, sie zu bearbeiten. Die Übungen helfen dir dabei, dein Verständnis zu überprüfen, sodass du rechtzeitig bemerkst, wenn du dem Inhalt nicht mehr folgen kannst.

Wer sollte dieses Buch lesen?

Das Buch wendet sich an Leser, die über grundlegende Kenntnisse der Programmierung verfügen und Algorithmen besser verstehen möchten. Vielleicht stehst du schon vor einer ganz konkreten Aufgabe, für die du eine Lösung in Form eines Algorithmus suchst. Oder du möchtest einfach nur wissen, wofür Algorithmen gut sind. Nachfolgend eine kurze, unvollständige Liste von Lesern, für die das Buch von Nutzen sein kann:

- Hobbyprogrammierer
- Schüler, die einen Programmierkurs besuchen
- Informatiker, die ihre Kenntnisse auffrischen möchten
- Physiker, Mathematiker oder andere Akademiker, die an der Programmierung interessiert sind

Konventionen und Downloads

Die Codebeispiele in diesem Buch sind in Python 2.7 programmiert. Für den im Buch abgedruckten Code wird eine `nicht-proportionale Schrift` verwendet, um ihn vom Fließtext zu unterscheiden. Einige der Listings enthalten Anmerkungen, die wichtige Konzepte hervorheben.

Die Codebeispiele kannst du unter folgender Adresse herunterladen:
`https://github.com/egonschiele/grokking_algorithms`

Ich bin davon überzeugt, dass du am besten lernst, wenn es dir Spaß macht – also gönne dir das Vergnügen und führe die Codebeispiele aus!

Über den Autor

Aditya Y. Bhargava ist als Softwareentwickler bei Etsy tätig, einem virtuellen Marktplatz für handgemachte Produkte. Er hat einen Master in Computer Science von der University of Chicago. Außerdem betreibt er unter `http://adit.io` ein beliebtes reich bebildertes Tech-Blog.

Danksagungen

Ich danke dem amerikanischen Originalverlag Manning, der mir die Möglichkeit gab, dieses Buch zu schreiben und mir eine Menge kreativer Freiheit ließ. Ich danke dem Herausgeber Marjan Bace, Mike Stephens, der mich engagierte, Bert Bates, der mich lehrte, wie man schreibt, und der unglaublich aufgeschlossenen und zuvorkommenden Redakteurin Jennifer Stout. Dank gebührt auch dem Produktionsteam: Kevin Sullivan, Mary Piergies, Tiffany Taylor, Leslie Haimes und vielen anderen, die hinter den Kulissen tätig waren. Darüber hinaus möchte ich den vielen Menschen danken, die das Manuskript gelesen und Verbesserungsvorschläge geliefert haben: Karen Bensdon, Rob Green, Michael Hamrah, Ozren Harlovic, Colin Hastie, Christopher Haupt, Chuck Henderson, Pawel Kozlowski, Amit Lamba, Jean-François Morin, Robert Morrison, Sankar Ramanathan, Sander Rossel, Doug Sparling und Damien White.

Dank gebührt auch den Menschen, die mir dabei geholfen haben, dieses Buch zu verwirklichen: den Mitarbeitern des Flashkit Game Boards, die mich das Programmieren lehrten, den vielen Freunden, die verschiedene Kapitel überprüft haben, Ratschläge gaben und es mir ermöglichten, unterschiedliche Erklärungen auszuprobieren. Hierzu gehören Ben Vinegar, Karl Puzon, Alex Manning, Esther Chan, Anish Bhatt, Michael Glass, Nikrad Mahdi, Charles Lee, Jared Friedman, Hema Manickavasagam, Hari Raja, Murali Gudipati, Srinivas Varadan und andere. Ich danke Gerry Brady, der mir Algorithmen erklärt hat. Großer Dank gebührt auch den Algorithmen-Koryphäen CLRS (Cormen, Leiserson, Rivest, Stein), Knuth und Strang. Ich stehe wahrhaft auf den Schultern von Giganten.

Ich danke meinem Vater, meiner Mutter, Priyanka und der übrigen Familie für die beständige Unterstützung. Und ein besonderes Dankeschön an meine Frau Maggie. Uns stehen noch viele Abenteuer bevor – und damit meine ich nicht, am Freitagabend zuhause zu bleiben und Abschnitte umzuschreiben.

Und schließlich ein großes Dankeschön an alle Leser, die sich auf dieses Buch eingelassen haben und an die Leser, die im Forum zu diesem Buch Feedback gegeben haben. Ihr habt wirklich dazu beigetragen, das Buch zu verbessern.

Einführung in Algorithmen

1

In diesem Kapitel:

- Du wirst einen ersten Suchalgorithmus programmieren (eine binäre Suche).
- Du wirst erfahren, wie man die Laufzeit eines Algorithmus mit der Landau-Notation beschreibt.
- Du wirst ein bei der Entwicklung von Algorithmen gängiges Verfahren kennenlernen: die Rekursion.

1.1 Einführung

Ein *Algorithmus* ist eine Reihe von Anweisungen, die eine Aufgabe ausführen. Man könnte eigentlich jeden Codeschnipsel als Algorithmus bezeichnen, aber dieses Buch befasst sich mit den interessanteren Aspekten. Die Algorithmen in diesem Buch habe ich ausgewählt, weil sie schnell sind, interessante Aufgabenstellungen lösen oder beides. Hier sind einige der Highlights:

- Dieses Kapitel beschreibt die binäre Suche und führt vor, wie ein Algorithmus deinen Code beschleunigen kann. In einem der Beispiele wird die Anzahl der erforderlichen Schritte von 4 Milliarden auf nur 32 reduziert!
- Ein GPS-Empfänger verwendet Graphenalgorithmen (die in Kapitel 6, Kapitel 7 und Kapitel 8 erörtert werden), um die kürzeste Route zu einem Ziel zu berechnen.
- Du kannst die dynamische Programmierung (siehe Kapitel 9) dazu verwenden, einen Algorithmus zu schreiben, der Dame spielt.

Ich werde hierfür jeweils einen Algorithmus erläutern und ein Beispiel dafür zeigen. Anschließend betrachten wir die Laufzeit des Algorithmus mithilfe der Landau-Notation (Landau-Symbole). Und schließlich erfährst du, welche anderen Aufgabenstellungen mit demselben Algorithmus gelöst werden können.

1.1.1 Performance

Die gute Nachricht ist, dass Implementierungen der Algorithmen, die in diesem Buch beschrieben werden, sehr wahrscheinlich in deiner Lieblingsprogrammiersprache verfügbar sind. Du musst die Algorithmen also nicht alle selbst schreiben! Allerdings sind diese Implementierungen nutzlos, wenn du deren Vor- und Nachteile nicht verstehst. In diesem Buch lernst du, die Vor- und Nachteile verschiedener Algorithmen miteinander zu vergleichen: Solltest du für eine bestimmte Aufgabe Mergesort oder lieber Quicksort verwenden? Ist ein Array oder eine Liste besser geeignet? Die Verwendung einer anderen Datenstruktur kann einen sehr großen Unterschied ausmachen.

1.1.2 Problemlösungen

Du wirst zudem Verfahren zur Problemlösung für Aufgaben kennenlernen, an die du dich bislang vielleicht nicht herangetraut hast. Einige Beispiele:

- Falls du Videospiele magst, kannst du ein System für die Künstliche Intelligenz (KI) programmieren, das Graphenalgorithmen verwendet und das den User im Spiel verfolgt.
- Du wirst erfahren, wie du mit dem k-Nächste-Nachbarn-Algorithmus (k-Nearest-Neighbors-Algorithmus) ein Empfehlungssystem entwickeln kannst.
- Manche Aufgaben lassen sich nicht in angemessen kurzer Zeit lösen. Der Abschnitt des Buchs über NP-vollständige Probleme beschreibt, wie du solche Probleme erkennen kannst und stellt einen Algorithmus vor, der dir eine Näherungslösung liefert.

Allgemeiner formuliert: Nach der Lektüre dieses Buchs werden dir einige der meisten verbreiteten und für sehr viele Fälle anwendbaren Algorithmen vertraut sein. Mit dem Wissen aus diesem Buch kannst du dich spezielleren Algorithmen für die KI, für Datenbanken usw. zuwenden oder dich noch größeren Herausforderungen stellen.

Erforderliche Kenntnisse

Für die weitere Lektüre des Buchs benötigst du Grundkenntnisse der Algebra. Betrachte beispielsweise die folgende Funktion: $f(x) = x \times 2$. Welchen Wert besitzt dann $f(5)$? Wenn deine Antwort 10 lautet, bist du bereit.

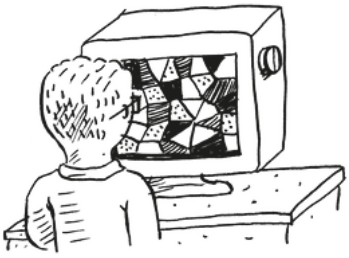
Darüber hinaus ist dieses Kapitel (wie das ganze Buch) leichter verständlich, wenn dir eine Programmiersprache vertraut ist. Die Beispiele in diesem Buch sind in Python geschrieben. Falls du noch keine Programmiersprache kennst und eine erlernen möchtest, solltest du Python wählen – die Sprache ist hervorragend für Anfänger geeignet. Wenn du eine andere dir bekannte Programmiersprache (wie z. B. Ruby) verwenden möchtest, ist das problemlos möglich.

1.2 Binäre Suche

Nehmen wir an, du suchst in einem Telefonbuch nach einer Person (wie altmodisch das klingt!). Der Name beginnt mit *K*. Du könntest nun am Anfang des Telefonbuchs loslegen und so lange blättern, bis du zum Buchstaben *K* gelangst. Wahrscheinlich wirst du mit der Suche jedoch eher in der Mitte anfangen, weil du weißt, dass sich die Einträge mit *K* ungefähr in der Mitte des Telefonbuchs befinden.

Oder du stellst dir vor, dass du einen Begriff, der mit dem Buchstaben *O* beginnt, in einem Wörterbuch suchst. Auch in diesem Fall wirst du mit der Suche in der Nähe der Mitte beginnen.

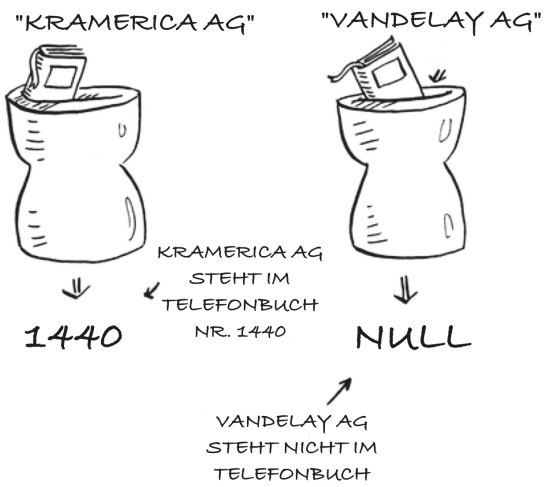
Und nun stelle dir vor, dass du dich bei Facebook anmeldest. Facebook muss dann überprüfen, ob du ein Konto besitzt und dementsprechend in einer Datenbank nach deinem Namen suchen. Nehmen wir an, dein Username lautet *karlmageddon*. Facebook könnte nun beim Buchstaben *A* mit der Suche anfangen – allerdings ist es sinnvoller, irgendwo in der Mitte anzufangen.



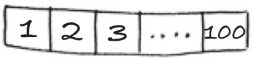
Bei dieser Aufgabe handelt es sich um eine Suche. Zur Lösung solcher Aufgaben kommt stets der gleiche Algorithmus zum Einsatz: die *binäre Suche*.

Die binäre Suche ist ein Algorithmus, dessen Eingabe aus einer sortierten Liste von Elementen besteht. (Ich erkläre später, warum die Liste sortiert sein muss.) Wenn das gesuchte Element in dieser Liste enthalten ist, liefert die binäre Suche die Position zurück, an der es sich befindet. Andernfalls gibt die binäre Suche den Wert `null` zurück.

Zum Beispiel:



Hier ist ein Beispiel für die Funktionsweise der binären Suche. Ich denke an eine Zahl zwischen 1 und 100.



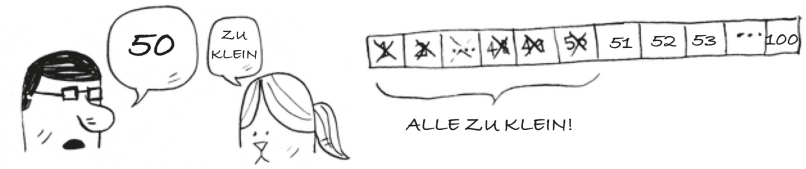
Du musst nun meine Zahl mit möglichst wenigen Versuchen erraten. Ich sage dir jeweils, ob die geratene Zahl zu groß, zu klein oder richtig ist.

Nehmen wir an, du nennst die Zahlen der Reihe nach: 1, 2, 3, 4, ... Das sähe dann folgendermaßen aus:

Hierbei handelt es sich um eine *einfache Suche* (vielleicht wäre *eintönige Suche* in diesem Fall eine passendere Bezeichnung). Mit jedem Versuch schließt du nur eine einzige Zahl aus. Wenn ich an die Zahl 99 gedacht hätte, würdest du 99 Versuche benötigen, um meine Zahl zu erraten!

1.2.1 Eine bessere Suchmethode

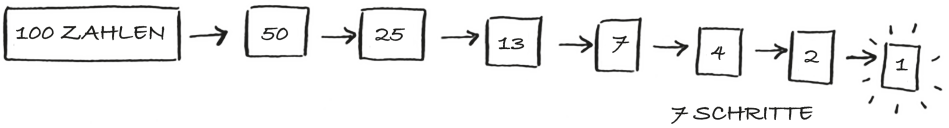
Ein besseres Verfahren ist das folgende: Fang mit der Zahl 50 an.



Diese ist zu klein, allerdings hast du soeben *die Hälfte* aller Zahlen ausgeschlossen! Du weißt nun, dass alle Zahlen von 1 bis 50 zu klein sind. Nächster Versuch: 75.

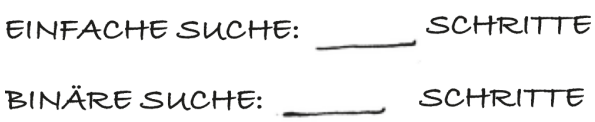
Zu groß, aber du hast wieder die Hälfte der verbliebenen Zahlen ausgeschlossen! *Bei einer binären Suche nennst du die Zahl in der Mitte und schließt dadurch jeweils die Hälfte der noch vorhandenen Zahlen aus.* Nun ist 63 an der Reihe (die Mitte zwischen 50 und 75).

So funktioniert die binäre Suche. Du hast soeben deinen ersten Algorithmus erlernt! So viele Zahlen kannst du mit jedem Rateversuch ausschließen:

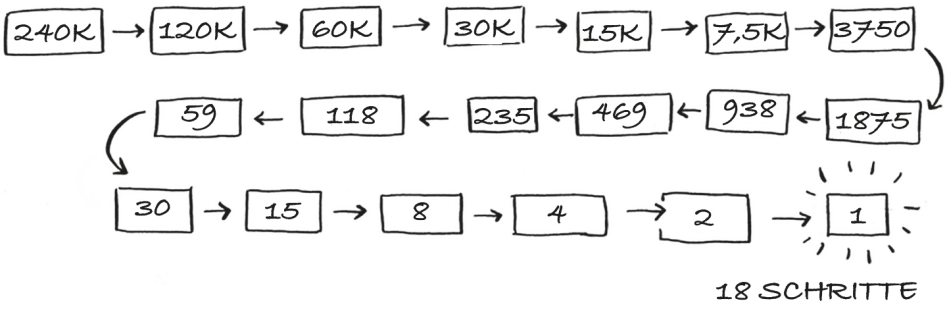


An welche Zahl ich denke, spielt keine Rolle: Du benötigst höchstens 7 Versuche, um sie zu erraten, weil bei jedem Rateversuch so viele Zahlen ausgeschlossen werden.

Nehmen wir wieder an, du suchst nach einem Begriff in einem Wörterbuch, das 240.000 Einträge enthält. Was meinst du, wie viele Schritte für eine Suche im *Worst Case*, also im ungünstigsten Fall, nötig sind?



Bei der einfachen Suche können 240.000 Schritte notwendig sein, sofern sich das gesuchte Wort ganz am Ende des Wörterbuchs befindet. Bei der binären Suche hingegen wird bei jedem Schritt die Anzahl der verbliebenen Wörter halbiert, bis schließlich nur noch ein Wort übrig ist.



Bei der binären Suche sind also 18 Schritte erforderlich – ein riesiger Unterschied! Verallgemeinert bedeutet das: Bei einer Liste der Länge n benötigt die binäre Suche im Worst Case $\log_2 n$ Schritte, bei einer einfachen Suche sind hingegen n Schritte erforderlich.

Logarithmen

Vielleicht erinnerst du dich nicht mehr daran, was Logarithmen sind, aber du weißt vermutlich noch, was Exponentialfunktionen sind. Der Ausdruck $\log_{10} 100$ entspricht der Frage: »Wie viele Zehnen muss man miteinander multiplizieren, um 100 zu erhalten?«. Die Antwort lautet 2: $10 \times 10 = 100$. Also ist $\log_{10} 100 = 2$. Logarithmen sind die Umkehrfunktionen von Exponentialfunktionen.

Wenn es in diesem Buch um die Laufzeit und die Landau-Notation (die in Kürze erklärt wird) geht, bedeutet log stets $\log_2$. Wenn du für die Suche nach einem Element eine einfache Suche verwendest, musst du im Worst Case jedes einzelne

Element überprüfen. Bei einer Liste von 8 Zahlen musst du höchstens 8 Zahlen überprüfen. Bei einer binären Suche musst du im Worst Case log *n* Elemente überprüfen. Für eine Liste mit 8 Elementen gilt log 8 == 3, denn $2^3 == 8$. Du musst also höchstens 3 Zahlen überprüfen (und dann kannst du mit dem 4. Versuch das richtige Ergebnis nennen). Für eine Liste mit 1.024 Elementen gilt log 1.024 == 10, denn $2^{10} == 1.024$. Bei einer Liste von 1.024 Zahlen musst du also höchstens 10 Zahlen überprüfen.

$10^2 = 100 \leftrightarrow \log_{10} 100 = 2$

$10^3 = 1000 \leftrightarrow \log_{10} 1000 = 3$

$2^3 = 8 \leftrightarrow \log_2 8 = 3$

$2^4 = 16 \leftrightarrow \log_2 16 = 4$

$2^5 = 32 \leftrightarrow \log_2 32 = 5$

Hinweis

In diesem Buch geht es des Öfteren um die logarithmische Laufzeit, deshalb sollte dir das Konzept von Logarithmen vertraut sein. Sollte dies nicht der Fall sein, findest du bei der Khan Academy (`https://www.khanacademy.org`) ein anschauliches englisches Video, das dieses Konzept verdeutlicht.

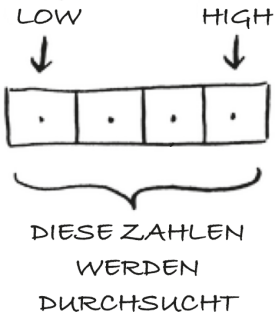
Hinweis

Die binäre Suche funktioniert nur dann, wenn die zu durchsuchende Liste sortiert ist. Die Namen in einem Telefonbuch sind beispielsweise alphabetisch sortiert. Hier kannst du also für die Suche nach einem Namen eine binäre Suche verwenden. Wie sähe es aus, wenn die Namen nicht sortiert wären?

Sehen wir uns doch einmal an, wie man eine binäre Suche in Python programmiert. Der Beispielcode verwendet Arrays. Falls du nicht weißt, wie Arrays funktionieren, keine Sorge, sie werden im nächsten Kapitel erklärt. Du brauchst nur zu wissen, dass sie eine Sequenz von Elementen in einer Reihe aufeinanderfolgender Behälter speichern können, deren Gesamtheit als Array bezeichnet wird. Die Behälter werden, angefangen bei 0, durchnummeriert: Der erste befindet sich an der Position #0, der zweite ist #1, der dritte #2 usw.

Die Funktion `binary_search` nimmt ein sortiertes Array und ein Objekt entgegen. Wenn das Objekt in diesem Array enthalten ist, liefert die Funktion dessen Position zurück. Du führst darüber Buch, welcher Teil des Arrays zu durchsuchen ist. Anfangs handelt es sich um das gesamte Array:

```
low = 0
high = len(list) - 1
```



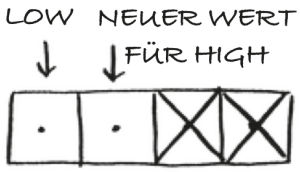
Du überprüfst jeweils das mittlere Element:

```
mid = (low + high) / 2 ❶
guess = list[mid]
```

❶ Der Wert der `mid`-Funktion wird von Python automatisch abgerundet, sofern `(low + high)` keine gerade Zahl ist.

Sollte der geratene Wert zu klein sein, aktualisierst du `low` dementsprechend:

```
if guess < item:
  low = mid + 1
```



Und sollte der geratene Wert zu groß sein, aktualisierst du `high`. Hier ist der vollständige Code:

```
def binary_search(list, item):

low = 0 ❶
  high = len(list)-1 ❶

  while low <= high: ❷
    mid = (low + high) ❸
    guess =  list[mid]
    if guess == item: ❹
      return mid
    if guess > item: ❺
      high = mid - 1
    else: ❻
      low = mid + 1
  return None ❼

my_list = [1, 3, 5, 7, 9] ❽

print binary_search(my_list, 3) # => 1 ❾
print binary_search(my_list, -1) # => None ❿
```

❶ `low` und `high` führen darüber Buch, welcher Teil der Liste durchsucht wird.

❷ Solange der Suchbereich mehr als ein Element umfasst ...

❸ ... wird das mittlere Element überprüft.

❹ Das gesuchte Objekt wurde gefunden.

❺ Der geratene Wert war zu groß.

❻ Der geratene Wert war zu klein.

❼ Das gesuchte Objekt ist in der Liste nicht enthalten.

❽ Testen der Funktion.

❾ Denk daran, dass die Nummerierung der Listenelemente bei 0 beginnt.

❿ Das zweite Element hat den Index 1.

Übungen

1.1 Eine sortierte Liste enthät 128 Namen. Du durchsuchst sie mit einer binären Suche. Wie viele Schritte sind dafür maximal erforderlich?