ADITYA Y. BHARGAVA

ALGORITHMEN KAPIEREN

VISUELL LERNEN UND VERSTEHEN

mit Illustrationen, Alltagsbeispielen und Python-Code

2. Auflage

mitp

Inhaltsverzeichnis

Vorwort

Immer mehr Menschen müssen das Programmieren lernen. Einige verdienen ihr täglich Brot damit, darunter Softwareentwickler oder Webentwickler. Für viele weitere Berufe, die traditionell nichts mit dem Programmieren am Hut hatten, spielt diese Fertigkeit jetzt oder in der Zukunft eine Rolle. Wer weiß, wie man programmiert, versteht zudem die Technologie um uns herum besser.

Leider ist das Wissen um die Vorteile der Programmierung ungleich verteilt. So ist der Anteil der Frauen und bestimmter Ethnien in den Informatikstudiengängen in Nordamerika sehr gering. Wir müssen unbedingt dafür sorgen, dass viel mehr Menschen aus allen Schichten und Ethnien viel mehr Wissen über Programmierung und Informatik erhalten. Dazu müssen wir in verschiedenen Bereichen Fortschritte machen und beispielsweise Vorurteile überwinden, mehr Lehrkräfte ausbilden und diversifiziertere Lernpfade bieten. Wir müssen es den Menschen leicht machen, in das Thema einzusteigen.

Bhargavas Buch begeistert mich, weil es einen neuen Ansatz dafür bietet, mehr über Algorithmen zu erfahren, denn Algorithmen sind ein zentraler Baustein für effektives Programmieren. Einige behaupten, man könne nur etwas über Algorithmen lernen, indem man sich einen dicken und staubtrockenen mathematischen Wälzer zu diesem Thema besorgt, ihn von der ersten bis zur letzten Seite durchackert und jede Information darauf verinnerlicht. Doch solche Bücher nutzen nur Menschen, die auf genau diese Weise lernen können, die über die notwendige Zeit verfügen, auf diese Weise zu lernen, und die überhaupt erst gelernt haben, auf diese Weise zu lernen. Diese Leute gehen auch davon aus, dass wir wissen, *wieso* jemand sich mit Algorithmen beschäftigen will. Ganz ehrlich: Das ist eine ziemlich gewagte Annahme.

Versteh mich nicht falsch: Ich habe einige Lieblingsbücher zum Thema Informatik, die das Wissen über Algorithmen streng mathematisch vermitteln. Ich komme gut damit zurecht. Die meisten Informatikprofessoren kommen gut damit zurecht. Aber genau darin liegt das Problem: Wir halten es viel zu schnell für einen Fakt, dass andere auf dieselbe Art und Weise wie wir selbst lernen. Dabei benötigen wir möglichst viele unterschiedliche Ressourcen, die Wissen über die verschiedensten Informatikthemen vermitteln und sich jeweils an eine bestimmte Zielgruppe richten.

Bhargavas Buch richtet sich ausdrücklich an Menschen, die eine Einführung in Algorithmen ohne mathematische Vorkenntnisse suchen. Am meisten beeindruckt mich, dass Bhargava bestimmte Dinge weggelassen hat. Ein Buch wie dieses kann nicht jedes Detail berücksichtigen. Das würde die Leserschaft überfordern und am Ziel vorbeischießen.

Dank seiner Lehrkompetenz schafft Bhargava es, auf wenigen Seiten viel Stoff verständlich zu vermitteln. Im Kapitel »Dynamische Programmierung« zeigt sich das auf großartige Weise: Bhargava nimmt Fragen vorweg, die in vielen Köpfen auftreten, denen aber andere Bücher über Algorithmen keine Zeile widmen würden.

Ich hoffe sehr, dass dieses Buch auch dir hilft, Ihre Kenntnisse zu erweitern. Dabei spielt es gar keine Rolle, ob du dich erstmals mit Algorithmen befasst oder ob du bisher einfach nicht die für dich richtigen Erklärungen gefunden hast. Viel Vergnügen beim Lesen!

– Daniel Zingaro, Universität Toronto

Geleitwort

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 machte die Sache ganz gut, 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.

Die erste Auflage dieses Buchs wurde 2016 veröffentlicht. Seitdem wurde es von über 100.000 Menschen gelesen. Ich freue mich sehr, dass der visuelle Stil so gut angekommen ist.

Auch für die zweite Auflage bleibt mein Ziel unverändert. Ich nutze Abbildungen und einprägsame Beispiele, damit die Konzepte in den Köpfen hängen bleiben. Das Buch richtet sich an Personen, die wissen, wie man programmiert, und die ohne mathematisches Kauderwelsch mehr über Algorithmen erfahren möchten.

Diese zweite Auflage füllt ein paar Lücken der Erstauflage. Ich habe häufig den Wunsch erhalten, das Konzept der Bäume ausführlich zu erläutern. Mit zwei neuen Kapiteln zu diesem Thema komme ich dem Wunsch gern nach. Auch der Abschnitt zum Stichwort NP-vollständig fällt in dieser Auflage länger aus. NP-Vollständigkeit ist ein sehr abstraktes Konzept. Ich habe versucht, es mit meiner Erklärung zu konkretisieren, und hoffe sehr, dass ich mein Ziel erreicht habe.

Meine Fähigkeit zu schreiben, hat sich seit meinem ersten Blogbeitrag weiterentwickelt und ich hoffe, dass dieses Buch eine leicht verständliche und informative Lektüre sein wird.

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 Vermutungen 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 vereinfachen es, sich zu erinnern. 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!

Verwendung dieses Buchs

Die Reihenfolge des Inhalts dieses Buchs 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 nachdrücklich, 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 davon haben.

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

Aufbau dieses Buchs: 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 hat zwei fundamentale Datenstrukturen zum Thema: 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, Kapitel 10 und Kapitel 11 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 11) ausprobieren. Möglicherweise stellst du jedoch fest, dass es keine effiziente Lösung gibt. In diesem Fall kannst du einen Greedy-Algorithmus (siehe Kapitel 10) verwenden, um eine Näherungslösung zu berechnen.
- *Hashtabellen:* Kapitel 5 hat Hashtabellen zum Thema. Eine Hashtabelle ist eine äußerst nützliche Datenstruktur, die Schlüssel-und-Wert-Paare enthält, wie beispielsweise den Namen einer Person und deren E-Mail-Adresse oder einen Benutzernamen und das dazugehörige Passwort. Die Bedeutung von Hashtabellen kann kaum überbewertet werden. 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?«.
- *Graphen- und Baumalgorithmen:* Die Kapitel 6, Kapitel 7, Kapitel 8 und Kapitel 9 behandeln diese beiden Themen. 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 9) 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. Bäume sind eine Art von Graphen. Sie werden in Datenbanken (häufig in Form von B-Bäumen), in deinem Browser (als DOM-Baum) und im Dateisystem deines Computers verwendet.
- *k nächste Nachbarn (KNN):* In Kapitel 12 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 13 werden weitere Algorithmen vorgestellt, die gut geeignet sind, um das Thema Algorithmen weiter zu vertiefen.

Konventionen und Downloads

Die Codebeispiele in diesem Buch sind in Python 3 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 Bhargava ist als Softwareentwickler tätig. Er hat einen Master in Computer Science von der Universität Chicago. Außerdem betreibt er unter `http://adit.io` ein beliebtes, reich bebildertes Tech-Blog.

Über den Fachkorrektor

David Eisenstat ist als Softwareentwickler in der Forschung tätig. Er hat an der Brown University in Computer Science promoviert.

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.

An all meine Testleser und Lektoren: Abhishek Koserwal, Alex Lucas, Andres Sacco, Arun Saha, Becky Huett, Cesar Augusto Orozco Manotas, Christian Sutton, Diógines Goldoni, Dirk Gómez, Ed Bacher, Eder Andres Avila Niño, Frans Oilinki, Ganesh Swaminathan, Giampiero Granatella, Glen Yu, Greg Kreiter, Javid Asgarov, João Ferreira, Jobinesh Purushothaman, Joe Cuevas, Josh McAdams, Krishna Anipindi, Krzysztof Kamyczek, Kyrylo Kalinichenko, Lakshminarayanan AS, Laud Bentil, Matteo Battista, Mikael Byström, Nick Rakochy, Ninoslav Cerkez, Oliver Korten, Ooi Kuan San, Pablo Varela, Patrick Regan, Patrick Wanjau, Philipp Konrad, Piotr Pindel, Rajesh Mohanan, Ranjit Sahai, Rohini Uppuluri, Roman Levchenko, Sambaran Hazra, Seth MacPherson, Shankar Swamy, Srihari Sridharan, Tobias Kopf, Vivek Veerappan, William Jamir Silva, and Xiangbo Mao – eure Vorschläge haben dazu beigetragen, das Buch zu verbessern.

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:

- Die Grundlagen für das Verständnis dieses Buchs.
- Du wirst einen ersten Suchalgorithmus programmieren (eine binäre Suche).
- Du wirst erfahren, wie man die Laufzeit eines Algorithmus mit der Landau-Notation beschreibt.

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 9 und Kapitel 10 erörtert werden), um die kürzeste Route zu einem Ziel zu berechnen.
- Du kannst die dynamische Programmierung (siehe Kapitel 11) 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.

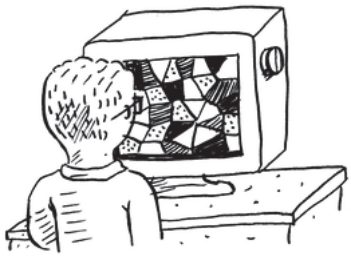
Darüber hinaus ist dieses Kapitel (wie das ganze Buch) leichter verständlich, wenn dir eine Programmiersprache vertraut ist. Die Beispiele in diesem Buch sind in Python geschrieben. Falls du noch keine Programmiersprache kennst und eine erlernen möchtest, solltest du Python wählen – die Sprache ist hervorragend 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.

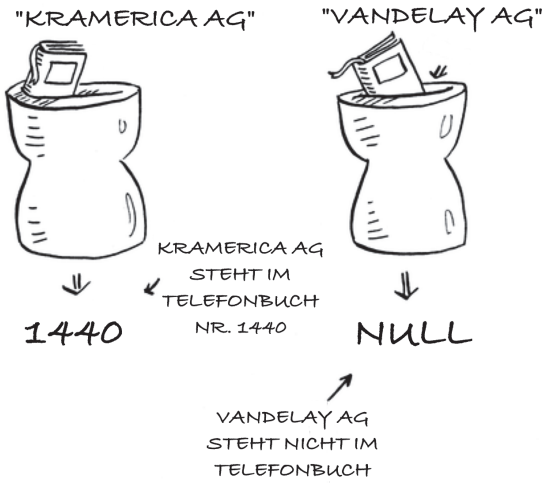
Und nun stelle dir vor, dass du dich bei Facebook anmeldest. Facebook muss dann überprüfen, ob du ein Konto besitzt und dementsprechend in einer Datenbank nach deinem Namen suchen. Nehmen wir an, dein Username lautet *karlmageddon*. Facebook könnte nun beim Buchstaben *A* mit der Suche anfangen – allerdings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



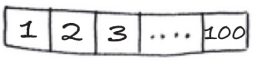
Bei dieser Aufgabe handelt es sich um eine Suche. Zur Lösung solcher Aufgaben kommt stets der gleiche Algorithmus zum Einsatz: die *binäre Suche*.

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.



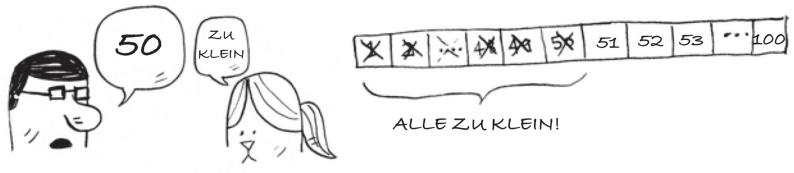
Du musst nun meine Zahl mit möglichst wenigen Versuchen erraten. Ich sage dir jeweils, ob die geratene Zahl zu groß, zu klein oder richtig ist.

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

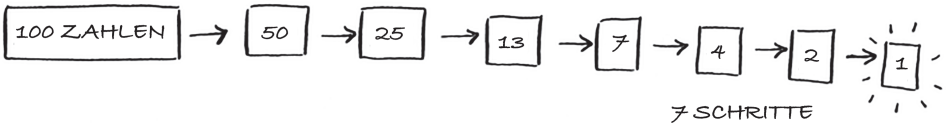
Ein besseres Verfahren ist das folgende: Fang mit der Zahl 50 an.



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.

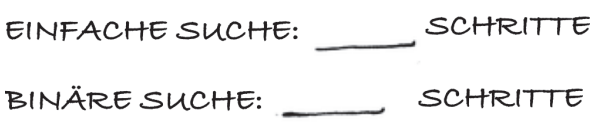
Zu groß, aber du hast wieder die Hälfte der verbliebenen Zahlen ausgeschlossen! *Bei einer binären Suche nennst du die Zahl in der Mitte und schließt dadurch jeweils die Hälfte der noch vorhandenen Zahlen aus.* Nun ist 63 an der Reihe (die Mitte zwischen 50 und 75).

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

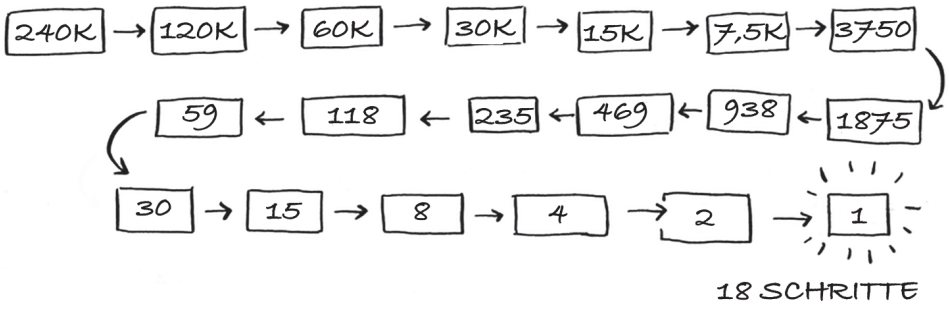


An welche Zahl ich denke, spielt keine Rolle: Du benötigst höchstens 7 Versuche, um sie zu erraten, weil bei jedem Rateversuch so viele Zahlen ausgeschlossen werden.

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?

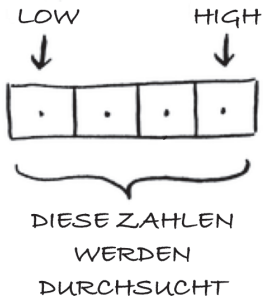
Sehen wir uns doch einmal an, wie man eine binäre Suche in Python programmiert. Der Beispielcode verwendet Arrays. Falls du nicht weißt, wie Arrays funktionieren, keine Sorge, sie werden im nächsten Kapitel erklärt. Du brauchst nur zu wissen, dass sie eine Sequenz von Elementen in einer Reihe aufeinanderfolgender 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.

Hinweis

Ich verwende die Begriffe *list* und *array* im Code synonym. Das liegt daran, dass Arrays in Python als Listen bezeichnet werden.

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
```



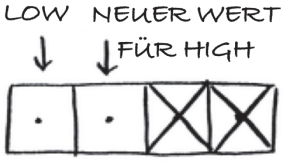
Du überprüfst jeweils das mittlere Element:

```
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)  // 2 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. Das zweite Element hat den Index 1

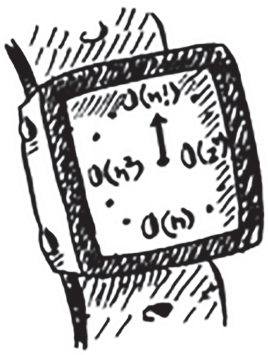
❿ »None« bedeutet in Python nil. Es zeigt an, dass ein Objekt nicht gefunden wurde.

Übungen

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

1.2 Nehmen wir an, du verdoppelst die Größe der Liste. Wie viele Schritte sind nun maximal erforderlich?

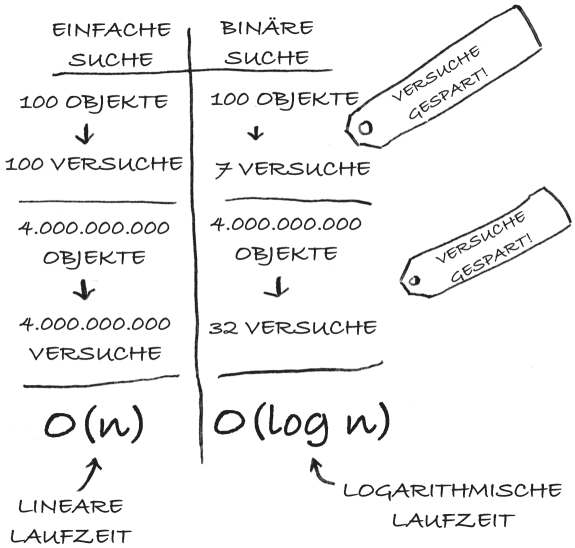
1.2.2 Laufzeit



Bei allen vorgestellten Algorithmen werde ich die Laufzeit erklären. Ob nun die Laufzeit oder aber der Speicherplatzbedarf optimiert werden soll: Im Allgemeinen möchte man den effizientesten Algorithmus auswählen.

Zurück zur binären Suche. Wie viel Zeit sparst du durch ihre Verwendung? Beim ersten Ansatz wurden alle Zahlen der Reihe nach überprüft. Wenn die Liste 100 Zahlen enthält, sind bis zu 100 Rateversuche erforderlich. Wenn die Liste 4 Milliarden Zahlen enthält, sind bis zu 4 Milliarden Rateversuche notwendig. Die maximale Anzahl der Rateversuche entspricht also der Größe der Liste. Man bezeichnet das als lineare Laufzeit.

Bei der binären Suche verhält es sich anders. Wenn die Liste 100 Objekte enthält, sind höchstens 7 Rateversuche erforderlich. Enthält die Liste 4 Milliarden Objekte, sind höchstens 32 Rateversuche notwendig. Ziemlich leistungsstark, oder? Die binäre Suche benötigt eine *logarithmische Laufzeit*. Die folgende Tabelle fasst unsere Ergebnisse zusammen.



1.3 Landau-Notation

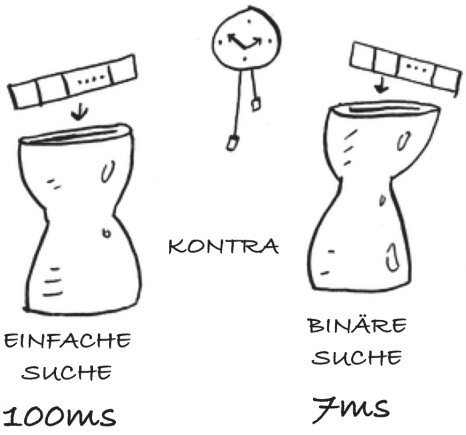
Die *Landau-Notation* (engl. *Big-O-Notation*, der Begriff Landau-Symbole ist ebenfalls gebräuchlich) ist eine spezielle Schreibweise, die angibt, wie schnell ein Algorithmus ist. Warum ist das wichtig? Du wirst feststellen, dass du häufig Algorithmen verwendest, die andere Leute entwickelt haben. In diesem Fall ist es praktisch zu wissen, wie schnell oder langsam diese Algorithmen sind. In diesem Abschnitt werde ich dir zeigen, was die Landau-Notation bedeutet und eine Liste der gängigsten Laufzeiten der Algorithmen präsentieren, die Landau-Symbole verwenden.

1.3.1 Die Laufzeiten von Algorithmen nehmen unterschiedlich schnell zu

Bob programmiert einen Suchalgorithmus für die NASA. Sein Algorithmus kommt zum Einsatz, wenn eine Rakete auf dem Mond landen soll und kann den Landeplatz berechnen.

Hierbei handelt es sich um ein Beispiel dafür, dass die Laufzeiten zweier Algorithmen auf unterschiedliche Weise zunehmen können. Bob muss sich zwischen einer einfachen Suche und einer binären Suche entscheiden. Der Algorithmus muss sowohl schnell sein als auch fehlerlos funktionieren. Einerseits ist die binäre Suche schneller. Bob hat nämlich nur *10 Sekunden* Zeit, um den Landeplatz zu ermitteln – denn sonst kommt die Rakete vom Kurs ab. Andererseits lässt sich die einfache Suche leichter programmieren. Und Bob möchte *wirklich* nicht, dass der Code zum Landen einer Rakete Bugs enthält! Vorsichtshalber will Bob die Laufzeiten beider Algorithmen messen, wenn er eine Liste verwendet, die 100 Elemente enthält.

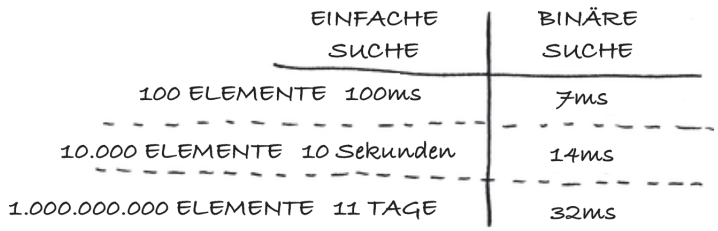
Nehmen wir an, das Überprüfen eines Elements dauert eine Millisekunde (ms). Bei der einfachen Suche muss Bob 100 Elemente überprüfen, also dauert das Ausführen der Suche 100 ms. Bei der binären Suche muss er hingegen lediglich 7 Elemente überprüfen. $\log_2 100$ ist ungefähr 7, also dauert die Suche 7 ms. Realistisch betrachtet wird die Liste tatsächlich eher eine Milliarde Elemente enthalten. Wie lange würde dann die einfache Suche dauern? Und wie lange die binäre Suche? Vergewissere dich, dass du diese beiden Fragen beantworten kannst, bevor du weiterliest.



LAUFZEITEN DER EINFACHEN UND DER BINÄREN SUCHE BEI EINER LISTE MIT 100 ELEMENTEN

Bob führt eine binäre Suche mit einer Liste aus, die eine Milliarde Elemente enthält. Sie dauert 30 ms ($\log_2$ 1.000.000.000 ist ungefähr 30). »30 ms!«, geht ihm durch Kopf. »Die binäre Suche ist rund 15 Mal schneller als die einfache Suche, denn bei 100 Elementen dauerte die einfache Suche 100 ms und die binäre Suche 7 ms. Also sollte die einfache Suche 30 × 15 = 450 ms dauern, richtig? Das ist deutlich unter den zulässigen 10 Sekunden.« Bob entschließt sich also dazu, die einfache Suche zu verwenden. Aber ist das die richtige Entscheidung?

Nein, denn wie sich zeigen wird, liegt Bob falsch. Völlig falsch. Die Laufzeit der einfachen Suche beträgt bei einer Milliarden Objekte eine Milliarde ms – das sind mehr als 11 Tage! Das Problem besteht hier darin, dass die Laufzeiten für binäre und einfache Suche *nicht auf dieselbe Weise zunehmen*.



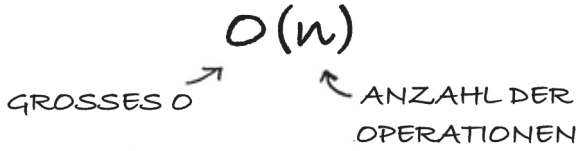
DIE LAUFZEITEN NEHMEN MIT SEHR UNTERSCHIEDLICHER GESCHWINDIGKEIT ZU!

Wenn die Anzahl der Objekte zunimmt, benötigt die binäre Suche zur Ausführung etwas mehr Zeit. Die einfache Suche jedoch benötigt *sehr viel* mehr Zeit zur

Ausführung. Deshalb wird die binäre Suche sehr viel schneller als die einfache Suche, wenn die Zahlenliste größer wird. Bob dachte, dass die binäre Suche 15 Mal schneller als die einfache Suche ist, aber das stimmt nicht. Wenn die Liste eine Milliarden Objekte enthält, ist die binäre Suche rund 33 Millionen Mal schneller. Aus diesem Grund reicht es nicht aus zu wissen, wie lange ein Algorithmus zur Ausführung benötigt – man muss auch wissen, wie die Laufzeit anwächst, wenn die Größe der Liste zunimmt. Hier kommt die Landau-Notation ins Spiel.

Die Landau-Symbole geben an, wie schnell ein Algorithmus ist. Betrachte beispielsweise eine Liste der Größe *n*. Bei der einfachen Suche muss jedes einzelne Element überprüft werden, daher sind *n* Operationen erforderlich. Für diese Laufzeit wird das Landau-Symbol O(*n*) verwendet. Aber wo sind die Sekunden angegeben? Es gibt keine – die Geschwindigkeit wird nicht in Sekunden angegeben. *Die Landau-Notation ermöglicht es, die Anzahl der Operationen zu vergleichen*. Sie teilt dir mit, wie schnell die Anzahl der Operationen des Algorithmus anwächst.

Hier ist ein weiteres Beispiel: Die binäre Suche benötigt log *n* Operationen, um eine Liste der Größe *n* zu überprüfen. Wie sieht das als Landau-Notation aus? Hierfür schreibt man O(log *n*). Im Allgemeinen notiert man ein Landau-Symbol folgendermaßen:

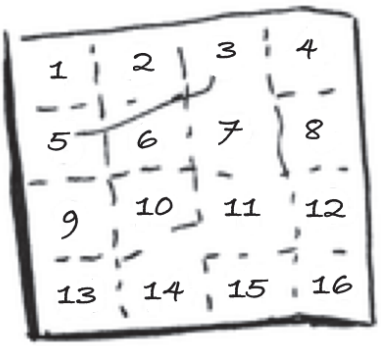


Du kannst daran ablesen, wie viele Operationen der Algorithmus ausführen wird. Im Englischen spricht man auch von der *Big-O-Notation*, weil der Anzahl der Operationen ein großes »O« vorangestellt wird. (Klingt wie ein Scherz, ist aber tatsächlich wahr!)

Betrachten wir einige Beispiele. Kannst du die Laufzeiten der Algorithmen herausfinden?

1.3.2 Visualisierung verschiedener Laufzeiten

Zunächst ein praktisches Beispiel, das du mit ein paar Blättern Papier und einem Bleistift leicht nachvollziehen kannst. Deine Aufgabe besteht darin, ein Gitter zu zeichnen, das aus 16 Kästchen besteht.



WELCHER ALGORITHMUS WÄRE ZUM ZEICHNEN DES GITTERS GEEIGNET?

Algorithmus 1

Eine Möglichkeit besteht darin, der Reihe nach 16 einzelne Kästchen zu zeichnen. Wie du weißt, gibt die Landau-Notation die Anzahl der Operationen an. In diesem Beispiel stellt das Zeichnen eines Kästchens eine Operation dar. Und du musst 16 Kästchen zeichnen. Wie viele Operationen sind also erforderlich, wenn du die Kästchen einzeln zeichnest?

DIE KÄSTCHEN DER REIHE NACH ZEICHNEN

Zum Zeichnen von 16 Kästchen sind 16 Schritte notwendig. Welche Laufzeit benötigt dieser Algorithmus demzufolge?

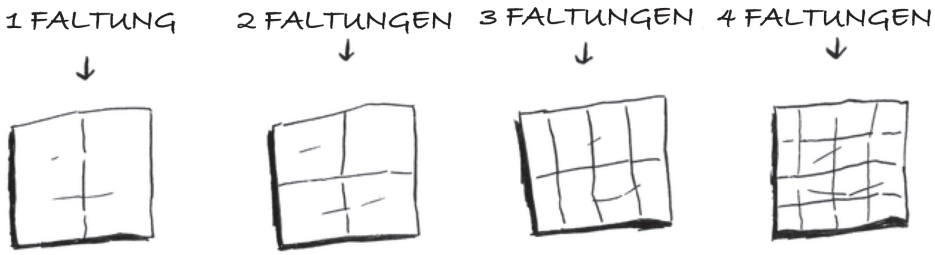
Algorithmus 2

Probiere nun den folgenden Algorithmus aus. Falte das Papierblatt.

Bei diesem Beispiel ist jedes Falten des Papierblatts eine Operation. Du hast soeben also zwei Kästchen mit nur einer Operation erstellt!

Falte das Papierblatt ein zweites, drittes und viertes Mal.

Falte das Papierblatt nach der vierten Faltung wieder auf. Und siehe da – ein tadelloses Gitter. Mit jeder Faltung verdoppelst du die Anzahl der Kästchen. Du hast also mit 4 Operationen 16 Kästchen erstellt!



Du kannst die Anzahl der Kästchen mit jeder Faltung verdoppeln, also kannst du mit 4 Schritten 16 Kästchen »zeichnen«. Welche Laufzeit benötigt dieser Algorithmus? Lege die Laufzeiten für die beiden Algorithmen fest, bevor du weitermachst.

Antworten: Algorithmus 1 benötigt die Laufzeit O(n) und Algorithmus 2 die Laufzeit O(log n).

1.3.3 Die Landau-Notation beschreibt die Laufzeit im Worst Case

Nehmen wir an, du verwendest eine einfache Suche, um in einem Telefonbuch nach einer Person zu suchen. Wie du weißt, benötigt die einfache Suche die Laufzeit O(n), was im Worst Case bedeutet, dass du sämtliche Einträge deines Telefonbuchs überprüfen musst. Im vorliegenden Fall suchst du nach meinem Namen »Adit«. Hierbei handelt es sich um den ersten Eintrag in deinem Telefonbuch. Du musst also gar nicht alle Einträge überprüfen – du hast das Gesuchte schon beim ersten Versuch gefunden. Heißt das nun, dass dieser Algorithmus die Laufzeit O(n) benötigt? Oder sogar nur O(1), weil du die Person beim ersten Versuch gefunden hast?

Die einfache Suche benötigt nach wie vor die Laufzeit O(*n*). In diesem Fall hast du zwar sofort gefunden, was du gesucht hast, aber hierbei handelt es sich um den günstigsten Fall. Die Landau-Notation beschreibt jedoch den *Worst Case*. Du kannst also sagen, dass du im *Worst Case* sämtliche Einträge im Telefonbuch einmal überprüfen musst – und das entspricht der benötigten Laufzeit O(*n*). Dabei handelt es sich also um eine Art Zusicherung (eine obere Schranke für die Laufzeit) – du kannst dir sicher sein, dass die einfache Suche niemals langsamer als O(*n*) sein wird.

Hinweis

Neben der Laufzeit im Worst Case, also im ungünstigsten Fall, spielt auch die Laufzeit im Average Case, also im durchschnittlichen Fall, eine wichtige Rolle. Diese sogenannten *Zeitkomplexitäten* (der Worst Case und der Average Case) werden einander in Kapitel 4 gegenübergestellt.

1.3.4 Typische Laufzeiten gebräuchlicher Algorithmen

Nachstehend sind fünf Laufzeiten (sortiert nach abnehmender Geschwindigkeit) aufgeführt, die dir häufig begegnen werden:

- O(log *n*), die auch als *logarithmische Laufzeit* bezeichnet wird. Beispiel: Die binäre Suche.
- O(*n*), die auch als *lineare Laufzeit* bezeichnet wird. Beispiel: Die einfache Suche.
- O(*n* * log *n*). Beispiel: Ein schneller Sortieralgorithmus wie Quicksort (das in Kapitel 4 erörtert wird).
- O(n^2). Beispiel: Ein langsamer Sortieralgorithmus wie Selectionsort (das in Kapitel 2 erörtert wird).
- O(*n*!). Beispiel: Ein richtig langsamer Algorithmus, wie der Algorithmus zur Lösung des Problems des Handlungsreisenden (mehr dazu in Kürze).

Nehmen wir an, du zeichnest wieder die 16 Kästchen und du kannst dir dafür einen von 5 verschiedenen Algorithmen aussuchen. Wenn du den ersten Algorithmus verwendest, wirst du eine Laufzeit von O(log *n*) benötigen, um die Kästchen zu zeichnen. Du kannst pro Sekunde 10 Operationen ausführen. Bei einer Laufzeit von O(log *n*) benötigst du 4 Operationen, um 16 Kästchen zu erzeugen (log 16 = 4). Du benötigst also 0,4 Sekunden zum Zeichnen des Gitters. Und wenn du 1.024 Kästchen zeichnen müsstest? Zum Zeichnen von 1.024 Kästchen sind log 1.024 = 10 Operationen bzw. 1 Sekunde Zeit erforderlich. Diese Angaben gelten für den ersten Algorithmus.

Stichwortverzeichnis

W

Z