

Digitalisierung und Programmierung

Prof. Dr. Nicolas Meseth

12. Mai 2025

Inhaltsverzeichnis

Vorwort	8
Warum dieses Buch?	8
Wen möchte ich wie ansprechen?	8
Wie ist das Buch aufgebaut?	9
Wie sollte man dieses Buch lesen?	10
Das LiFi-Projekt	11
Worum geht es im LiFi-Projekt?	11
Was sind die Ziele?	11
Grundlegende Konzepte	12
Technische Umsetzung	12
Leistungsmerkmale	12
Wie gehen wir vor?	12
Was benötigen wir für das LiFi-Projekt?	13
Hardware	13
Software	15
Übungsaufgaben	15
I. Probleme	17
1. Problemlösung	19
Zusammenfassung	19
1.1. Wie lösen wir Probleme systematisch?	19
1.2. Warum sind Computer beim Lösen von Problemen nützlich?	20
1.3. Wie stellen wir Probleme für Computer dar?	21
1.3.1. Das Eingabe-Verarbeitung-Ausgabe-Modell	22
1.3.2. Die Lösung des Problems	31
1.4. Welche Strategien helfen bei der Lösung von Problemen?	33
1.4.1. Problemzerlegung (<i>Problem Decomposition</i>)	33
1.4.2. Teile und Herrsche (<i>Divide and Conquer</i>)	35
1.4.3. Verteile und Parallelisiere (<i>Distribute and Parallelize</i>)	36
Übungsaufgaben	39
2. Algorithmen	41
Zusammenfassung	41

2.1.	Was ist ein Algorithmus?	41
2.1.1.	Herkunft des Begriffs	41
2.1.2.	Algorithmen und Programme	42
2.2.	Wie stellen wir Algorithmen dar?	43
2.2.1.	Natürliche Sprache	43
2.2.2.	Pseudocode	44
2.2.3.	Flussdiagramme	44
2.2.4.	Programmcode	46
2.3.	Welche Arten von Algorithmen gibt es?	47
2.3.1.	Mathematische Algorithmen	47
2.3.2.	Suchalgorithmen	54
2.3.3.	Sortieralgorithmen	55
2.3.4.	Optimierungsalgorithmen	56
2.3.5.	Graphenalgorithmen	57
2.3.6.	Stochastische Algorithmen	58
2.3.7.	Maschinelle Lernalgorithmen	58
2.4.	Welche wichtigen algorithmischen Denkmuster gibt es?	58
2.4.1.	Sequenzen	58
2.4.2.	Verzweigungen	58
2.4.3.	Iterationen	59
2.4.4.	Kapselung	59
2.4.5.	Rekursion	59
2.5.	Was sind Herausforderungen bei der Formulierung von Algorithmen?	59
2.5.1.	Haltekriterium	59
2.5.2.	Endlosschleifen	60
2.5.3.	Beurteilung des Ergebnisses	60
2.5.4.	Erklärbarkeit des Ergebnisses	60
2.6.	Gibt es bessere und schlechtere Algorithmen?	61
2.6.1.	Komplexität	61
2.6.2.	Verständlichkeit	61
	Übungsaufgaben	62

II. Repräsentation 64

3. Informationen 65

3.1.	Wie definieren wir Information?	65
3.2.	Wie messen wir Information?	67
3.2.1.	Bit für Bit	67
3.2.2.	Unsicherheit	70
3.2.3.	Information	72
3.2.4.	Wahrscheinliche und unwahrscheinliche Antworten	73
3.2.5.	Mehr als zwei Antworten	74

3.3.	Wovon hängt der Informationsgehalt einer Nachricht ab?	75
3.3.1.	Anzahl der Alternativen	75
3.3.2.	Wahrscheinlichkeiten der Alternativen	75
3.3.3.	Länge der Nachricht (Anzahl Zeichen)	75
3.4.	Was ist Redundanz und welche Rolle spielt sie?	76
3.5.	Welcher Zusammenhang besteht zwischen Information und Daten	76
	Übungsaufgaben	77
4.	Bits	79
4.1.	Was ist ein Zahlensystem?	79
4.1.1.	Stellenwertsysteme	79
4.1.2.	Das Dezimalsystem	80
4.1.3.	Zahlensysteme mit einer anderen Basis	80
4.2.	Ist das Bit die kleinste Informationseinheit?	81
4.3.	Was ist ein Byte?	81
	Übungsaufgaben	82
5.	Codesysteme	83
5.1.	ASCII-Code	83
5.2.	Verallgemeinerung von Codesystemen	83
5.3.	Unicode	83
5.4.	RGB-Code	83
5.5.	Code vs Codec	83
5.6.	Übungsaufgaben	84
6.	Datenstrukturen	86
6.1.	Wie stellen wir Daten im Computer dar?	86
6.2.	Was ist eine Datenstruktur und wozu brauchen wir sie?	86
6.3.	Welche Datenstrukturen gibt es?	86
6.3.1.	Listen	86
6.3.2.	Mengen	86
6.3.3.	Graphen	86
6.3.4.	Bäume	86
6.3.5.	Warteschlangen	86
	Übungsaufgaben	87
III.	Verarbeitung	88
7.	Analog vs. Digital	89
7.1.	Was ist der Unterschied zwischen der analogen und digitalen Welt?	89
7.1.1.	Endlich viele Möglichkeiten vs. Unendlichkeit	89
7.1.2.	Diskrete und kontinuierliche Werte	89

8. Speicher	90
8.1. Ist die Speicherung von Informationen an ein Material gebunden?	90
8.1.1. Substratunabhängigkeit	90
8.2. Welche Speicherformen gibt es in der Praxis?	90
9. Logik und Arithmetik	91
9.1. Schalter	91
9.1.1. Relais, Vakuumröhren und Transistoren	91
9.2. Logikgatter	91
9.3. Binäre Addition	91
9.4. Binäre Subtraktion	91
9.5. Substratunabhängigkeit	91
10. Computer	92
10.1. Komponenten eines Computers	92
IV. Kommunikation	93
11. Signale	94
11.1. Was ist ein Signal?	94
11.1.1. Der Kern ist die Veränderung	94
11.1.2. Häufig elektrisch	94
11.2. Wie können wir Signale erzeugen?	95
11.2.1. Von digital zu analog	95
11.2.2. Die LED	95
11.3. Wie können wir Signale empfangen?	95
11.3.1. Von analog zu digital	95
11.3.2. Der Farbsensor	95
11.4. Welche Bedeutung hat ein Signal?	95
12. Protokolle	96
12.1. Peer-To-Peer	96
12.2. Handshake	96
12.3. TCP/IP & HTTPs	96
13. Verschlüsselung	97
13.1. Verschlüsselung	97
13.1.1. Symmetrisch	97
13.1.2. Asymmetrisch	97
13.2. Digitale Signaturen	97
14. Kompression	98
14.1. Verlustfreie Kompression	98

14.2. Verlustbehaftete Kompression	98
V. Programmierung	99
15. Willkommen in Python	101
15.1. Unser erstes Programm	101
15.1.1. Nichts als Text	101
15.1.2. Zeile für Zeile	101
15.1.3. Kommentare	102
15.1.4. Nicht nur sequenziell	102
15.2. Ein Programm ausführen	102
15.3. Befehle in Python	103
15.3.1. Eingebaute Befehle	103
15.3.2. Befehle aus mitgelieferten Modulen	103
15.3.3. Befehle aus externen Modulen	104
15.3.4. Argumente eines Befehls	105
15.3.5. Rückgabewerte eines Befehls	105
15.4. Fehler	106
Übungsaufgaben	106
15.4.1. Reflektionsfragen	106
15.4.2. Programmieraufgaben	106
16. Variablen und Datentypen	108
16.1. Wozu dienen Variablen?	108
16.2. Konstanten	108
16.3. Benennung von Variablen	109
16.3.1. Regeln bei der Benennung von Variablen	109
16.3.2. Konventionen bei der Benennung von Variablen	110
16.4. Datentypen	110
16.4.1. Zeichenketten (<i>Strings</i>)	110
16.4.2. Numerische Werte (<i>Integer</i> und <i>Float</i>)	111
16.4.3. Wahrheitswerte (<i>Boolean</i>)	112
16.4.4. Komplexe Datentypen	112
16.5. Variablen im Computer	113
Übungsaufgaben	113
16.5.1. Reflektionsfragen	113
16.5.2. Programmieraufgaben	114
17. Funktionen	116
Übungsaufgaben	116
17.0.1. Reflektionsfragen	116
17.0.2. Programmieraufgaben	116

18. Collections	118
Übungsaufgaben	118
18.0.1. Reflektionsfragen	118
18.0.2. Programmieraufgaben	118
19. Kontrollstrukturen	119
Übungsaufgaben	119
19.0.1. Reflektionsfragen	119
19.0.2. Programmieraufgaben	119
20. Schleifen	120
Übungsaufgaben	120
20.0.1. Reflektionsfragen	120
20.0.2. Programmieraufgaben	120
21. Fehlersuche- und Behandlung	122
Übungsaufgaben	122
21.0.1. Reflektionsfragen	122
21.0.2. Programmieraufgaben	122
Literaturverzeichnis	123

Vorwort

Warum dieses Buch?

Dieses Buch entstand aus der Erkenntnis, dass viele klassische Lehrbücher der Informatik für Anfänger oft zu technisch und abstrakt sind. In meiner langjährigen Lehrtätigkeit habe ich beobachtet, dass Studierende besonders dann erfolgreich lernen, wenn sie die Konzepte der Informatik in einem praktischen Kontext erleben können. Deshalb habe ich mich entschieden, einen praxisorientierten Ansatz zu wählen, der theoretische Grundlagen mit einem konkreten Projekt verbindet.

Wen möchte ich wie ansprechen?

Ich richte dieses Buch an Menschen ohne Vorkenntnisse in den Bereichen Digitalisierung, Computertechnik oder Programmierung. Ich gehe nicht davon aus, dass die Leserinnen und Leser eine besondere Motivation mitbringen, sich mit diesen Themen zu beschäftigen. Wenn doch, umso besser. Diese beiden Annahmen – fehlende Vorkenntnisse und Motivation – treffen auf den Großteil meiner Studierenden zu, für die ich dieses Buch in erster Linie geschrieben habe.

Mein Ziel ist es, sowohl die Kenntnisse als auch die Begeisterung für die Informatik zu steigern – zumindest bei einigen, die dieses Buch zur Hand nehmen (müssen). Um dies zu erreichen, habe ich mich entschieden, einen anderen Weg einzuschlagen als klassische Informatik-Lehrbücher:

- Ich verwende bewusst eine **einfache Sprache**. Das heißt nicht, dass wir keine Fachbegriffe einführen werden. Wir erklären die Sachverhalte aber zunächst in einer für alle Studierenden verständlichen Sprache und führen Fachbegriffe schrittweise ein.
- Neben der Sprache knüpfe ich bei den Beispielen gezielt an bestehendes Wissen an. Ich verwende **Beispiele und Analogien aus dem Alltag**, um Ideen und Konzepte der Informatik zu veranschaulichen. Das mag nicht immer perfekt gelingen – aber wenn es gelingt, hilft es meiner Erfahrung nach dabei, neue Themen besser zu verstehen.
- Mein Fokus liegt auf dem **Verständnis** der Konzepte statt auf technischen Details. Ich verzichte bewusst auf zu viel Tiefe zugunsten eines zugänglichen Buches, das einen guten Überblick vermittelt und echtes Verständnis ermöglicht. Wer anschließend Lust auf mehr Tiefe hat, bekommt von mir in jedem Kapitel Leseempfehlungen an die Hand.

- Ich bin überzeugt, dass konkrete Projekte das Interesse und Verständnis am besten fördern. Dieses Buch verbindet das **LiFi-Projekt** mit den theoretischen Grundlagen der Informatik und führt Schritt für Schritt an algorithmisches Denken und Programmierung heran. Das Ergebnis ist ein fertiges Produkt, für das die Leserinnen und Leser alle erlernten Kenntnisse praktisch anwenden mussten – ganz nach dem Prinzip „Learning by Doing“.
- Mir ist bewusst, dass viele der jüngeren Generation das Lesen eines Buches als Herausforderung empfinden. Dennoch halte ich Bücher für unverzichtbar, um komplexe Themengebiete zu erschließen. Um den Leseprozess zu erleichtern, stelle ich ergänzende **Videos und Audioaufnahmen** bereit, die in den Kapiteln verlinkt und über QR-Codes zugänglich sind.

Wie ist das Buch aufgebaut?

Dieses Buch befasst sich mit der Frage, wie wir Computer zum Lösen von Problemen einsetzen können. Das Schaubild in Abbildung 1 visualisiert die wichtigsten Themenblöcke.

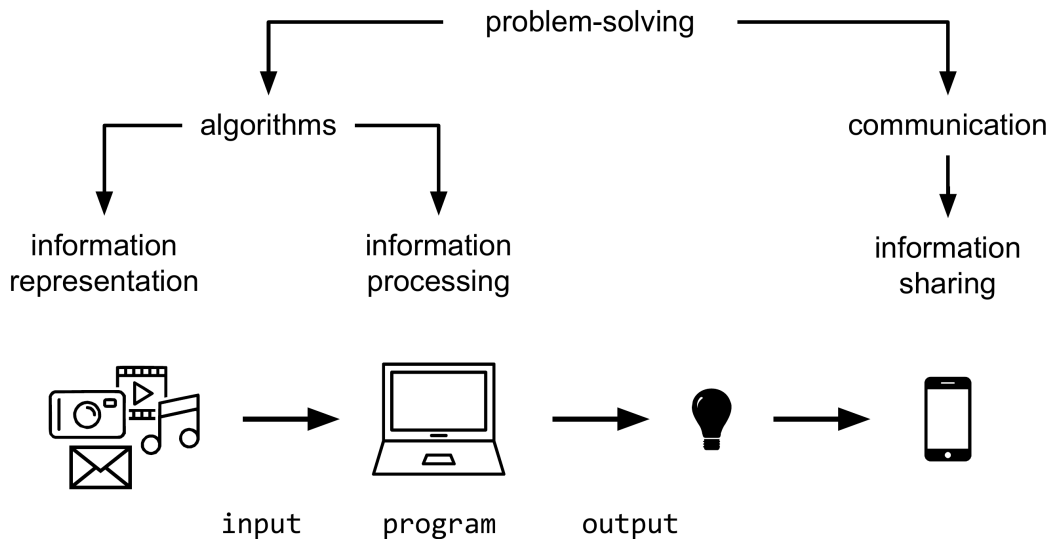


Abbildung 1.: Überblick über die Themenblöcke dieses Buches.

Ich orientiere mich übergeordnet an dem Thema des Problemlösens (*problem-solving*), das sich in zwei Bereiche gliedert:

1. **Algorithmen** (*algorithms*) bilden den Kern des Problemlösens und der Informatik. Sie beschreiben die notwendigen Schritte zur Lösung eines Problems.
2. **Kommunikation** (*communication*) umfasst das Teilen von Informationen in Computernetzwerken. Mit der weiten Verbreitung des Internets ist dieser Aspekt zentral für die moderne Computernutzung geworden. Lösungen, die ein Computer erzeugt, können heute unmittelbar über Netzwerke weltweit geteilt werden.

Im Bereich der Algorithmen beschäftigen wir uns zunächst damit, wie wir Probleme für Computer verständlich und lösbar beschreiben können. Eine zentrale Rolle spielt dabei die **Informationsrepräsentation** (*information representation*): Wie können wir Zahlen, Texte, Bilder, Videos, Audioaufnahmen und andere wichtige Inhalte so darstellen, dass ein Computer damit arbeiten kann?

Nachdem wir das verstanden haben, widmen wir uns der **Informationsverarbeitung** (*information processing*) – also der Frage, wie Eingabeinformationen so verarbeitet werden können, dass eine Lösung entsteht. Dies ist die Kernaufgabe der Algorithmen, und wir untersuchen, wie ein Algorithmus sowohl für Menschen als auch für Computer verständlich dargestellt werden kann. Dabei lernen wir die Programmiersprache Python kennen, die als moderne Programmiersprache Algorithmen in ausführbare **Programme** (*programs*) übersetzt. Anhand einfacher Beispiele wie der Addition zweier Zahlen verstehen wir zudem, wie die Ausführung von Programmierbefehlen im Rechner auf der Ebene der Bits funktioniert.

Bei der Kommunikation stellen wir uns die Frage, wie **das Teilen von Informationen** (*information sharing*) über unterschiedliche Medien wie Kabel, Luft oder Licht funktioniert. Wir lernen dabei etwas über das Senden und Empfangen von Signalen, über Protokolle – also Vereinbarungen zur Informationsübermittlung – sowie über die Frage, wie wir unsere Kommunikation effizient und gleichzeitig sicher gestalten können.

Wie sollte man dieses Buch lesen?

Das Buch ist für eine lineare Lektüre von vorne nach hinten konzipiert. An der Hochschule Osnabrück behandeln wir in der zugehörigen Veranstaltung pro Woche ein Kapitel – gelegentlich auch zwei, abhängig von der Verteilung der Feiertage im Semester. Die Kapitel hängen zusammen und bauen teilweise aufeinander auf.

Wie beschrieben orientiert sich dieses Buch an der Durchführung eines Projekts – dem LiFi-Projekt. Das Projekt bildet den Ausgangspunkt jedes Kapitels, und für jedes Thema wird der praktische Bezug zum Projekt hergestellt. Was genau das LiFi-Projekt beinhaltet, schauen wir uns im nächsten Kapitel an.

Das LiFi-Projekt

Anhand einer praktischen Problemstellung gestaltet uns das LiFi-Projekt die Einführung in die digitale Welt interessanter, greifbarer und praxisnäher. Als klassisches Ingenieursproblem kann es die grundlegenden Prinzipien der Digitalisierung besonders anschaulich verdeutlichen.

Worum geht es im LiFi-Projekt?

Das LiFi-Projekt konzentriert sich auf die Verbindung zwischen analoger und digitaler Welt. Im Zentrum steht die Nutzung von Licht zur Datenübertragung: Digitale Informationen werden in Lichtsignale umgewandelt und beim Empfang wieder zurück in digitale Form konvertiert.

LiFi (Light Fidelity) ist eine innovative Technologie zur optischen Datenübertragung. Sie verwendet das sichtbare Lichtspektrum und einen Fotodetektor, der LED-Signale erfasst. Durch präzise Steuerung der Lichteigenschaften - wie Wellenlänge und Helligkeit - werden Daten kodiert und übertragen. Diese Technologie eignet sich besonders für Umgebungen, in denen herkömmliche Funkwellen problematisch sind, etwa in der Nähe empfindlicher Elektronik oder bei Unterwasser-Anwendungen.

Die Kombination von Hardware und Software spielt dabei eine zentrale Rolle. Dies ermöglicht uns, die Grundlagen der computergestützten Informationsverarbeitung praktisch zu verstehen und erste Programmiererfahrungen in einem realen Anwendungskontext zu sammeln.

In diesem Projekt arbeitest du als Mitglied eines interdisziplinären F&E-Teams an der Entwicklung eines LiFi-Kommunikationsgeräts für Roboter im Agrar- und Lebensmittelsektor. Das System basiert auf drei Kernkomponenten: eine **RGB-LED** mit über 16 Millionen Farbkombinationen als Sender, ein präziser **Farbsensor** als Empfänger und ein **Master Brick** als zentrale Steuerungseinheit für die Kommunikation zwischen Roboter und Peripheriegeräten.

Was sind die Ziele?

Das LiFi-Projekt bietet eine praktische Einführung in ein klassisches Ingenieursproblem: die Integration von Hardware und Software zur Lösung realer Herausforderungen. Um die Projektziele systematisch zu erreichen, befassen wir uns mit folgenden Kernfragen:

Grundlegende Konzepte

- Wie erfolgt die Umwandlung physikalischer Größen (z.B. Licht, Temperatur) in digitale Daten?
- Wie können Algorithmen diese digitalen Daten verarbeiten und nutzen?

Technische Umsetzung

- Wie funktioniert die Informationsübertragung mittels Lichtsignalen?
- Wie lässt sich ein effektives Kommunikationssystem mit LED und Farbsensor realisieren?
- Welches Übertragungsprotokoll eignet sich optimal für die LED-basierte Kommunikation?

Leistungsmerkmale

- Wie zuverlässig ist die Datenübertragung?
- Welche maximale Übertragungsdistanz können wir erreichen?
- Welche Umgebungsbedingungen sind erforderlich?
- Wie kann die Übertragungssicherheit gewährleistet werden?
- Welche Datenübertragungsrate ist möglich?
- Wie lässt sich die Kommunikationseffizienz optimieren?

Dabei arbeiten wir ausschließlich mit der bereitgestellten Hardware - einer LED und einem Farbsensor - um diese Fragen zu beantworten.

Nach Abschluss des Projekts wirst du nicht nur praktische Lösungen für diese technischen Herausforderungen entwickelt haben, sondern auch ein fundiertes Verständnis der digitalen Welt und grundlegende Programmierfähigkeiten erworben haben.

Wie gehen wir vor?

Ein solch großes und komplexes Ingenieursproblem wie das LiFi-Projekt erfordert ein durchdachtes Vorgehen. Da es sich vornehmlich um ein Projekt zur Einführung in die digitale Welt handelt, gehen wir schrittweise vor und lernen bei jedem Schritt wichtige Grundlagen, die uns bei der Umsetzung helfen.

Zuerst widmen wir uns dem Basteln: Nach Anleitung setzen wir aus den bereitgestellten Hardware-Bauteilen den LiFi-Prototypen zusammen. Dieser bildet die Grundlage für unsere praktische Arbeit im Projekt. Während die Hardware nur einmalig zu Beginn aufgebaut werden muss, entwickeln wir die Software kontinuierlich über das gesamte Projekt hinweg weiter.

Um das Problem besser zu verstehen und in kleinere Teilprobleme zu zerlegen, betrachten wir in Kapitel 1 zunächst geeignete Techniken zur Problemlösung. Parallel dazu beginnen wir mit der Programmierung in Python – der Sprache, die wir für die Entwicklung der LiFi-Software nutzen werden. Diese Kenntnisse erweitern wir in jedem Kapitel, führen wichtige Konzepte der Programmierung ein und lernen, wie wir mit Python die Hardware des LiFi-Prototypen steuern können.

Das Buch ist in vier Teile gegliedert. Nach jedem Teil stellen wir eine neue Version unseres LiFi-Prototypen fertig. Die finale Version enthält dann alle notwendigen Funktionen, die zur Beantwortung der Fragestellungen erforderlich sind.

Was benötigen wir für das LiFi-Projekt?

Hardware

Komponenten

Für den LiFi-Hardware-Prototyp benötigen wir folgende Komponenten:

- 1 x [Master Brick 3.1](#)
- 1 x [RGB LED Bricklet 2.0](#)
- 1 x [Color Bricklet 2.0](#)
- 1 x [OLED 128x64 Bricklet 2.0](#)
- 4 x [Bricklet Cable 15 cm \(7p-7p\)](#)
- 1 x [USB-A to USB-C Cable 100 cm](#)
- 2 x [Mounting Plate 22x10](#)
- 4 x [Mounting Kit 12 mm](#)

Bitte überprüfe vor dem Fortfahren mit den folgenden Anweisungen, ob dein LiFi-Kit alle Komponenten in den angegebenen Mengen enthält.



Aufbauanleitung

1. Schutzfolie auf den Befestigungsplatten entfernen
2. Abstandshalter an beide Befestigungsplatten anbringen

Befestigungsplatte 1

Befestigungsplatte 2

3. Master Brick befestigen
4. Verbindungskabel einstecken
5. Befestigungsplatten miteinander verbinden
6. OLED-Anzeige montieren
7. LED und Farbsensor montieren
8. Peripheriegeräte mit Master Brick verbinden

Software

Neben der Hardware benötigen wir für die Entwicklung des LiFi-Prototyps verschiedene Software-Komponenten. Die Software ist komplett Open-Source und dadurch kostenlos nutzbar. Alle Programme sind für Windows, Mac OS und Linux verfügbar. Hier zunächst die Übersicht, bevor wir jede Software im Detail vorstellen:

- [Brick Daemon](#) und [Brick Viewer](#)
- [Visual Studio Code](#)
- [Python](#)
- [Git](#)

Brick Daemon

Brick Viewer

Visual Studio Code

Python

- Virtuelle Python-Umgebungen
- Externe Module installieren

Git

- GitHub
- Ein Repository klonen
- Den lokalen Code aktualisieren

Übungsaufgaben

1. Verwendet den Brick Viewer und verbindet euch mit eurem Master Brick. Stellt sicher, dass die Verbindung funktioniert:
 - a. Öffnet den Tab für die RGB LED und erkundet die verfügbaren Funktionen.
 - b. Testet den Farbsensor, indem ihr seine Reaktionen unter verschiedenen Lichtverhältnissen beobachtet.
 - c. Ermittelt die eindeutige Identifikationsnummer (UID) für jeden eurer Sensoren im Brick Viewer.
2. Startet das Visual Studio Code und öffnet dort ein neues Terminalfenster:

- a. Wechselt über die Kommandozeile in ein Verzeichnis eurer Wahl und klonst das GitHub-Repository mit dem Beispielcode für dieses Buch mittels des `git`-Befehls.
- b. Erstellt eine neue virtuelle Python-Umgebung im Ordner `.env` und aktiviert diese. Installiert die für den Beispielcode benötigten Python-Module `tinkerforge` und `pyyaml`.

Teil I.

Probleme

Da das LiFi-Projekt eine komplexe Herausforderung darstellt, verschaffen wir uns im ersten Teil des Buches einen Überblick darüber, wie wir computerbasierte Probleme strukturiert angehen und lösen können.

In Kapitel 1 klären wir zunächst wichtige Fragen zum Thema **Problemlösung**:

- Wie lösen wir Probleme systematisch?
- Warum sind Computer beim Lösen von Problemen nützlich?
- Wie stellen wir Probleme für Computer dar?
- Welche Strategien helfen bei der Lösung von Problemen?

Wir werden sehen, dass das LiFi-Projekt kein einzelnes Problem darstellt, sondern sich in viele kleinere Teilprobleme zerlegen lässt, die sich leichter lösen lassen. Die gefundenen Lösungen formulieren wir als Algorithmen, die wir später mithilfe einer Programmiersprache in ausführbare Computerbefehle umsetzen. In Kapitel 2 klären wir wichtige Fragen rund um **Algorithmen**:

- Was ist ein Algorithmus?
- Wie stellen wir Algorithmen dar?
- Welche Arten von Algorithmen gibt es?
- Welche wichtigen algorithmischen Denkmuster gibt es?
- Was sind Herausforderungen bei der Formulierung von Algorithmen?
- Gibt es bessere und schlechtere Algorithmen?

Nach Abschluss dieses ersten Buchteils werden wir nicht nur ein fundiertes Verständnis der verschiedenen Teilprobleme des LiFi-Projekts entwickelt haben, sondern auch konkrete Ansatzpunkte identifiziert haben, wo und wie algorithmische Lösungen entwickelt werden müssen. Diese systematische Herangehensweise bildet die Grundlage für die praktische Umsetzung in den folgenden Kapiteln des Buches.

1. Problemlösung

Zusammenfassung

Wir beginnen das Buch im ersten Kapitel, indem wir den wichtigsten Aspekt für die Nutzung von Computern genauer beleuchten – das Lösen von Problemen. Konkret suchen wir Antworten auf die folgenden Fragen:

- Welches Vorgehen eignet sich für die systematische Lösung von Problemen?
- Warum sind Computer besonders gut geeignet, um bestimmte Probleme zu lösen?
- Was ist das EVA-Modell und wie hilft es uns, Probleme für Computer darzustellen?
- Welche Problemlösungsstrategien können wir anwenden?

1.1. Wie lösen wir Probleme systematisch?

Pólya und Conway (2004) beschreiben in ihrem Aufsatz *How to Solve It* einen weit verbreiteten Ansatz zur Problemlösung. Dieser basiert auf vier Schritten, die als wiederkehrender Kreislauf angewendet werden können und sollten. Obwohl Pólya und Conway (2004) diesen Ansatz ursprünglich für mathematische Probleme entwickelten, lässt er sich auch auf andere Bereiche – insbesondere die Informatik – übertragen. Folgende Schritte beschreiben beide in ihrem Aufsatz:

1. Das Problem verstehen
2. Einen Plan zur Lösung erstellen
3. Den Plan umsetzen
4. Die Lösung reflektieren und verbessern

Während alle wichtig sind, konzentrieren wir uns in diesem Kapitel auf den ersten Schritt: das Problem zu verstehen. Bei der Lösung eines Problems mithilfe eines Computers müssen wir diesen Schritt in zwei Stufen unterteilen. Die erste Stufe beinhaltet eine gründliche Auseinandersetzung mit dem Problem in der realen Welt, um dessen Kern zu erfassen. Diese Grundvoraussetzung wird oft vernachlässigt. Erst wenn ein klares Verständnis des Problems vorhanden ist, können wir zur zweiten Stufe übergehen: der Übersetzung des Problems in ein Modell, das die computerrelevanten Aspekte des Problems erfasst und beschreibt. Wir führen dafür gleich das EVA-Modell ein Kapitel [1.3](#).

Der zweite Schritt im Ansatz von Pólya und Conway (2004) ist die Erstellung eines Plans für die Lösung des Problems. In der Informatik besteht dieser Plan aus einer Reihe von Anweisungen, die in einer bestimmten Reihenfolge ausgeführt werden müssen. Dies lässt sich gut mit einer Wegbeschreibung vergleichen: “Fahre 500 Meter geradeaus, biege an der Kreuzung links ab, fahre bis zum zweiten Kreisverkehr und nimm dort die zweite Ausfahrt. Nach 200 Metern findest du dein Ziel auf der linken Seite.” Eine solche schrittweise Anleitung nennen wir in der Informatik Algorithmus – wir lernen dazu mehr in Kapitel 2.

Nachdem wir einen Plan zur Lösung des Problems erarbeitet und in geeigneter Form aufgeschrieben haben, müssen wir ihn im dritten Schritt umsetzen. In der Informatik bedeutet dies, den Lösungsweg einem Computer beizubringen. Dies erreichen wir durch das Schreiben eines Programms in einer Programmiersprache wie etwa Python. Das Programmieren zu erlernen ist eines der Ziele dieses Buches, und wir kommen darauf in vielen Kapiteln zurück.

Es wäre überraschend, wenn wir an unserer Lösung keine Verbesserungsmöglichkeiten finden würden. Dies gilt besonders für die Informatik und unsere entwickelten Programme. Häufig entdecken wir Fehler, die wir beheben müssen, oder stoßen auf Sonderfälle des Problems, die wir im ersten Entwurf nicht bedacht haben. Da Informatiker meist Programme für andere Menschen entwickeln, erhalten wir von den Nutzern wertvolles Feedback mit konkreten Verbesserungsvorschlägen. Die Reflexion und Verbesserung unserer Lösung ist der letzte Schritt im Ansatz von Pólya und Conway (2004). Wie die Pfeile in Abbildung 1.1 zeigen, führt dieser letzte Schritt uns oft zurück zum ersten Schritt. Dadurch gewinnen wir ein tieferes Verständnis des Problems, was wiederum Änderungen am Plan und der Umsetzung erfordert. Gerade in der Softwareentwicklung ist man eigentlich nie wirklich fertig.

Bevor wir uns mit dem oben genannten EVA-Modell beschäftigen, das uns hilft, Probleme adäquat für Computer zu beschreiben, wollen wir einen kurzen Blick auf eine andere Frage werfen: Warum überhaupt Computer zur Problemlösung?

1.2. Warum sind Computer beim Lösen von Problemen nützlich?

Der wichtigste Grund für die Nutzung von Computern ist das Lösen von Problemen. Ob wir eine Route mit Google Maps planen, Online-Bestellungen bei DHL verfolgen oder eine KI wie ChatGPT um eine Empfehlung bitten – überall lösen Computer Probleme. Warum? Weil Computer zwei Eigenschaften besitzen, die für viele Probleme und deren Lösung vorteilhaft sind:

1. Computer machen keine Fehler. Wenn wir einem Computer einen Lösungsweg beibringen, wendet er ihn fehlerfrei auf neue Probleme an.
2. Computer sind unglaublich schnell. Ob einfache Schritte, komplexe Berechnungen oder die Verarbeitung großer Datenmengen – Computer lösen Probleme in einem Bruchteil der Zeit, die wir Menschen benötigen würden.

Polya's approach to problem-solving



Abbildung 1.1.: Der Ansatz von Polya zum Lösen von Problemen.

Diese beiden Eigenschaften ermöglichen es uns, mit Computern besonders solche Probleme effizient zu lösen, die wiederkehrend und in großer Zahl auftreten. Wir sprechen dann von **Automatisierung**.

In diesem Kapitel lernen wir, wie Computer Probleme strukturieren und lösen. Um den Begriff des Problems besser zu verstehen und seine Bedeutung im Kontext von Computern einzugrenzen, führen wir zunächst ein einfaches Modell ein.

1.3. Wie stellen wir Probleme für Computer dar?

Das LiFi-Projekt stellt eine komplexe Herausforderung dar. Beim Lesen der Ziele und Fragestellungen des [LiFi-Projekts](#) kann man sich überfordert fühlen. Die zentrale Frage lautet: Wie nähern wir uns dieser Aufgabe systematisch an?

Ein bewährter Ansatz für komplexe Situationen ist die Vereinfachung. Auch wenn wir das Problem selbst nicht vereinfachen können, können wir es durch eine strukturierte Herangehensweise besser verstehen und handhabbarer machen. Die Verwendung von Modellen ist dafür ein geeigneter Weg.

Modelle zielen darauf ab, die wesentlichen Aspekte der realen Welt hervorzuheben und unwichtige Details auszublenden. Da dies zunächst abstrakt klingen mag, werden wir es anhand

eines Modells veranschaulichen, das uns durch das gesamte Buch begleiten wird: das Eingabe-Verarbeitung-Ausgabe-Modell, kurz EVA-Modell.

1.3.1. Das Eingabe-Verarbeitung-Ausgabe-Modell

Das Eingabe-Verarbeitung-Ausgabe-Modell (EVA-Modell, s. Abbildung 1.2) ist ein wichtiges Modell in der Informatik. Es erklärt die Arbeitsweise von Computern auf vereinfachte Weise und beinhaltet nur die nötigsten Elemente. Konkret zeigt das Modell, wie Computer Probleme lösen und welche drei Elemente wir dabei betrachten müssen: Computer benötigen **(1) Eingabedaten**, die sie durch einen definierten **(2) Verarbeitungsprozess** in gewünschte **(3) Ausgabedaten** umwandeln.

Das EVA-Modell beschreibt ein Problem und dessen Lösung durch drei Komponenten: die Eingabe (*Input*), die der Computer erhält, die Verarbeitung (*Computation*), die er mit diesen Daten durchführt, und die Ausgabe (*Output*), die er als Ergebnis liefert. Wenn wir diese drei Komponenten beschreiben können, haben wir die für den Computer relevanten Aspekte des Problems erfasst – alles andere ist unwichtig.



Abbildung 1.2.: Das EVA-Modell besteht aus der Eingabe, der Berechnung und der Ausgabe.

Wenden wir das EVA-Modell auf das LiFi-Projekt an. Das LiFi-Gerät soll verschiedene Daten erfassen: Lichtsignale von einem anderen LiFi-Gerät und die Temperatur des eigenen Sensors. Basierend auf diesen Daten trifft es eine Entscheidung. Anschließend kommuniziert es diese

Entscheidung zusammen mit den erfassten Temperaturdaten über Lichtsignale an das nächste Gerät. Zwar ist jeder dieser Teilschritte für sich genommen komplex und erfordert eine eigene Lösung. Dennoch können wir das EVA-Modell nutzen, um das LiFi-Projekt als Ganzes darzustellen, indem wir von den Details der einzelnen Schritte abstrahieren:



Was haben wir nun dadurch gewonnen, dass wir das EVA-Modell angewendet haben? Wir können uns nun auf die einzelnen Elemente konzentrieren und diese getrennt voneinander betrachten. Damit zerlegen wir das große, überfordernde Problem in kleinere Teile und machen es dadurch besser beherrschbar.

Bei den Eingaben müssen wir uns fragen, wie diese konkret aussehen und erfasst werden. Dabei geht es vor allem darum, in welcher Form die Eingaben dem Computer vorliegen müssen, damit er sie verarbeiten kann. Wie werden Lichtsignale im Computer dargestellt? Wie wird die physikalische Größe der Umgebungstemperatur in eine computerverständliche Form umgewandelt, und wie sieht diese aus? Es geht also um die **Repräsentation von Informationen**.

Sobald wir die Darstellung der Eingaben geklärt haben, können wir diese als Grundlage für die Verarbeitung nutzen. Wie muss ein Programm aussehen, das auf Basis der Eingabedaten die richtige Entscheidung trifft? Welche Schritte sind notwendig? Welche Prüfungen muss das Programm durchführen? Bei diesem Schritt geht es folglich um die **Verarbeitung von Informationen**.

Schließlich müssen wir die Form der Ausgabe festlegen. Wie soll das Verarbeitungsergebnis konkret aussehen? Da wir für die Kommunikation wieder Lichtsignale verwenden, geht es auch bei der Ausgabe um die **Repräsentation von Informationen**.

Wir können die Perspektive auf eine geräteübergreifende LiFi-Sicht erweitern. Dann kommt neben der Repräsentation von Informationen ein weiterer Aspekt hinzu: Die **Kommunikation von Informationen**. Wie übertragen wir die Informationen vom ersten LiFi-Gerät zum nächsten?



Vielleicht ist dir aufgefallen, dass die Struktur des gesamten Buches an den gerade identifizierten, übergeordneten Problemstellungen ausgerichtet ist. Wir befinden uns gerade im **ersten Teil** und sprechen darüber, wie wir Probleme im Computer darstellen. Im **zweiten Teil des Buches** lernen wir, wie Computer ganz unterschiedliche Informationen repräsentieren können, damit sie für die Lösung des Problems verwendet werden können. Im **dritten Teil** beschäftigen wir uns damit, wie Computer Informationen verarbeiten. Der **vierte und letzte Teil** führt uns in wichtige Konzepte der Kommunikation mittels Computern und Netzwerken ein.

1.3.1.1. Beispiel: Taschenrechner

Am Beispiel eines Taschenrechner lässt sich das EVA-Modell gut darstellen. Wir können uns bildlich vorstellen, wie ein Mensch die Eingabe tätigt und danach das Ergebnis abliest. Es

ist wichtig zu verstehen, dass Eingabe und Ausgabe sehr unterschiedliche Formen annehmen können und keinesfalls nur über eine Tastatur erfolgen müssen.

Das Beispiel des Taschenrechners wird in Abbildung 1.3 anhand einer einfachen Addition zweier Zahlen konkreter verdeutlicht. Als Eingabe werden zwei Zahlen benötigt, die Berechnung erfolgt durch eine Addition, dargestellt durch das Plusymbol. Die Ausgabe ist das Ergebnis – die Summe.

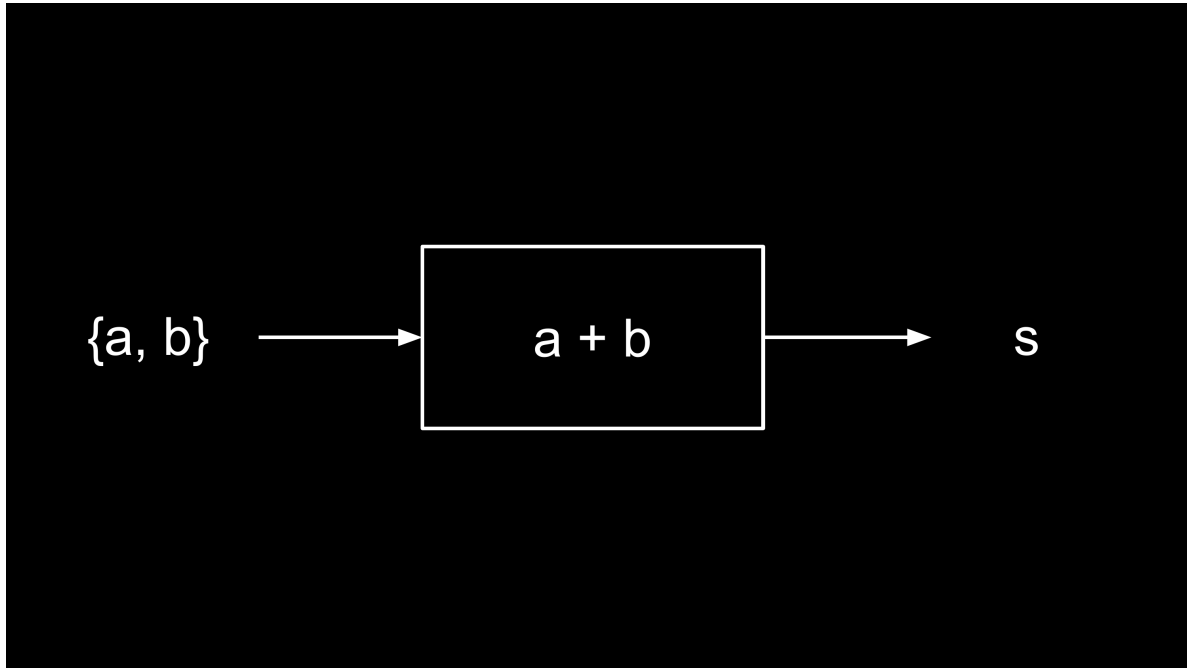


Abbildung 1.3.: Das EVA-Modell für die Addition zweier Zahlen.

Dieses einfache Beispiel zeigt, dass wir verstehen müssen, wie Computer die drei Bestandteile des EVA-Modells umsetzen. Beim Taschenrechner sind Ein- und Ausgabe jeweils Zahlen. Diese Daten speichert der Computer in seinem Arbeitsspeicher. Dabei ist wichtig zu wissen, dass Computer auf der untersten Ebene ausschließlich Nullen und Einsen speichern. Wir müssen also verstehen, wie Computer Zahlen mithilfe dieser Binärzahlen darstellen können.

Was passiert bei der Berechnung in der Mitte des Modells? Eine Addition mag uns einfach erscheinen, doch auch hier müssen wir beachten, dass Computer mit Binärzahlen arbeiten. Es stellt sich also die Frage: Wie funktioniert eine Addition, wenn die Zahlen als Folge von Nullen und Einsen dargestellt sind? Auf die beiden Fragen zur **Repräsentation und der Verarbeitung von Informationen** im binären System werden wir im Laufe des Buches Antworten bekommen.

1.3.1.2. Beispiel: Pflanzen zählen

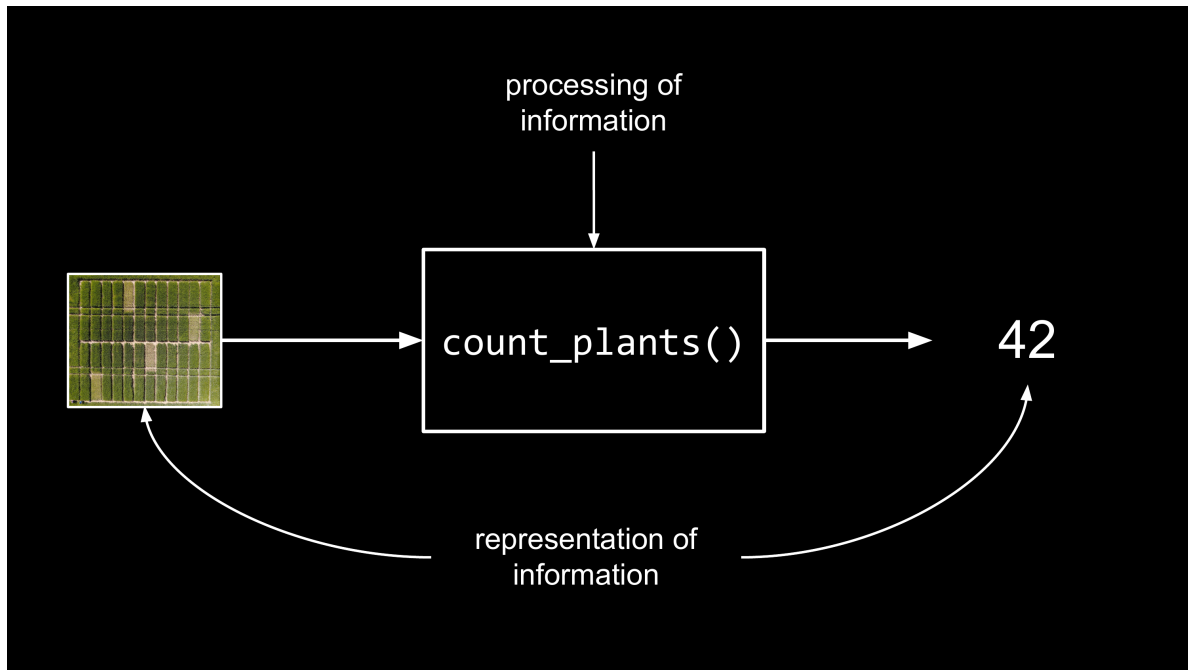
Betrachten wir ein weiteres Beispiel: Stell dir vor, du möchtest einen Computer nutzen, um Maispflanzen auf einer Drohnenaufnahme eines Ackers zu zählen. Diese Aufgabe ist für Menschen zwar einfach zu verstehen, wäre aber sehr zeitaufwändig auszuführen. Moderne Algorithmen ermöglichen es Computern, Objekte auf Bildern präzise zu lokalisieren und zu zählen. Nehmen wir an, wir haben für dieses Problem bereits eine Lösung entwickelt und ein Programm namens `count_plants` erstellt. Nun stellt sich die Frage: Wie sehen die Eingabe und die Ausgabe für dieses Problem aus? Was benötigt das Programm von uns, und was liefert es als Ergebnis?



Die erwartete **Ausgabe** lässt sich einfach beschreiben: Das Ergebnis der Zählung ist eine ganze Zahl. Die **Eingabe** für dieses Problem ist - anders als beim Taschenrechner - kein Tastendruck, sondern ein Bild. Damit der Computer das Bild verarbeiten kann, muss es dem Computer in digitaler Form bereitgestellt werden. Was das genau bedeutet, lernen wir in einem späteren Kapitel. Hier genügt es uns zu verstehen, *dass* wir das Bild digital abbilden müssen.



Wie gelangt das Bild in den Computer? Dies ist im Modell nicht näher definiert und für die Problembeschreibung auch nicht wesentlich. Das Bild muss lediglich irgendwie in den Arbeitsspeicher des Programms `count_plants` gelangen. Dies kann auf verschiedene Arten geschehen: Es kann von der Festplatte gelesen werden, über eine drahtlose Verbindung wie Bluetooth direkt übertragen und verarbeitet werden, oder das Programm `count_plants` läuft direkt auf der Drohne und greift unmittelbar auf deren Kamera zu. Die technische Umsetzung ist für unser Modell zunächst irrelevant. In einem späteren Kapitel werden wir uns damit befassen, wie Informationen genau übertragen und gespeichert werden. Genauso werden wir lernen, wie die benötigten Informationen für die Ein- und Ausgabe eines Programms in digitaler Form dargestellt werden können.



1.3.1.3. Beispiel: Schach spielen

Ein weiteres Beispiel zur Verdeutlichung des EVA-Modells ist das Schachspiel. Das Problem lässt sich einfach beschreiben: Der Computer soll auf Grundlage einer bestehenden Spielsituation den bestmöglichen nächsten Zug vorschlagen. Dieser Zug soll die Gewinnchancen maximieren.



Betrachten wir zunächst die Eingabe für dieses Problem: Wir können dem Computer nicht einfach ein physisches Schachbrett zeigen, sondern müssen überlegen, wie sich ein Schachbrett und die Position der Figuren in digitaler Form darstellen lassen. Dabei kann es durchaus mehrere Möglichkeiten geben, die uns ans Ziel führen.

Ein Schachbrett lässt sich etwa als Liste von 64 Feldern darstellen, die von oben links nach unten rechts durchnummeriert sind. Für jedes Feld speichern wir, ob es leer ist oder welche Figur darauf steht. Die Figuren werden durch Buchstaben dargestellt – zum Beispiel “R” für den Turm (Englisch: Rook) oder “N” für den Springer (Englisch: Knight). Für die Farben Schwarz und Weiß verwenden wir einfach 0 und 1. Diese Darstellungsform reduziert unser Problem auf Listen, Zahlen und Buchstaben in digitaler Form. Für Computer ist das eine leicht zu verarbeitende Struktur, wie wir später noch sehen werden. Ein Beispiel für eine solche Kodierung zeigt [Abbildung 1.4](#).

Die Ausgabe, also der nächste Zug, lässt sich ebenfalls durch Zahlen und Buchstaben darstellen. Eine weit verbreitete Notation gibt zunächst die Koordinate des Ausgangsfelds an, von dem eine Figur gezogen werden soll, gefolgt von der Koordinate des Zielfelds. Ein Beispiel wäre der Zug von “E2 nach E4”. Statt “E2” und “E4” könnten wir ebenso die entsprechende Zahl zwischen 1 und 64 aus unserer Liste verwenden, um mit dem obigen Schema konsistent zu bleiben. Der Zug hieße dann “53 nach 37”.

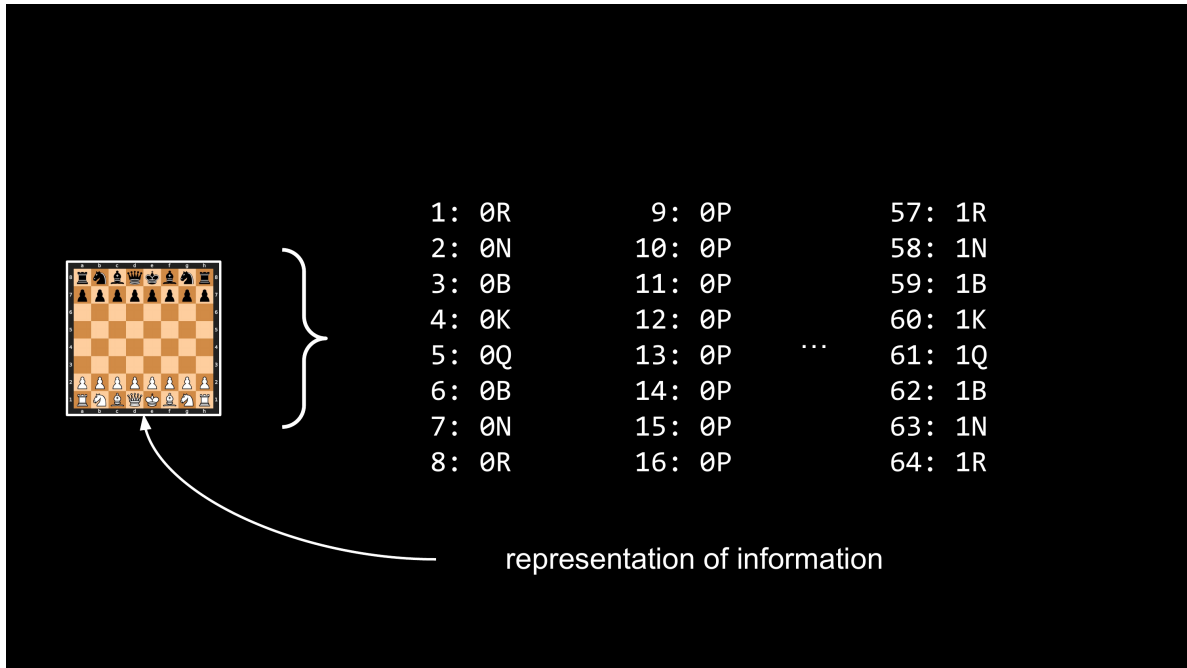


Abbildung 1.4.: Beispiel für die Darstellung von Schachfiguren als Zahlen und Buchstaben.

1.3.1.4. Beispiel: Mit Computern chatten

Als drittes Beispiel betrachten wir die Verwendung von Chatprogrammen wie ChatGPT. Seit seiner Veröffentlichung im November 2022 hat es die Welt stark verändert und einen regelrechten KI-Hype ausgelöst. Ein großes Sprachmodell wie GPT-4, das hinter dem heutigen ChatGPT steckt, ist eine komplexe Software, die wir in diesem Buch nicht vollständig ergründen können. Das Schöne an Modellen wie dem EVA-Modell ist jedoch, dass sie komplexe Sachverhalte vereinfachen können – so auch bei Sprachmodellen. Das Problem, das Sprachmodelle lösen, lässt sich wie alle Probleme in unserem EVA-Modell einfach darstellen.

Dabei betrachten wir das Sprachmodell – oder ChatGPT – als Blackbox, ohne die internen Prozesse genauer zu definieren. Für unser Modell genügt es zu verstehen, dass wir eine Eingabe in Form einer Nachricht an ChatGPT benötigen und als Ausgabe eine Antwort erhalten. Auch hier stellt sich die Frage, wie wir beides digital repräsentieren können, damit ChatGPT damit arbeiten kann.

Klassischerweise bestehen sowohl Eingabe als auch Ausgabe einfach aus Texten – allerdings beherrschen moderne Sprachmodelle auch andere Eingabeformen wie Bilder oder gesprochene Sprache über ein Mikrofon. Wir sprechen dann von multimodalen KI-Modellen. Bei Bildern stehen wir vor demselben Repräsentationsproblem wie bei unserer Drohnenaufnahme. Bei der Sprache stellt sich neben der Repräsentation von Audioinhalten die Frage, wie wir gesprochene

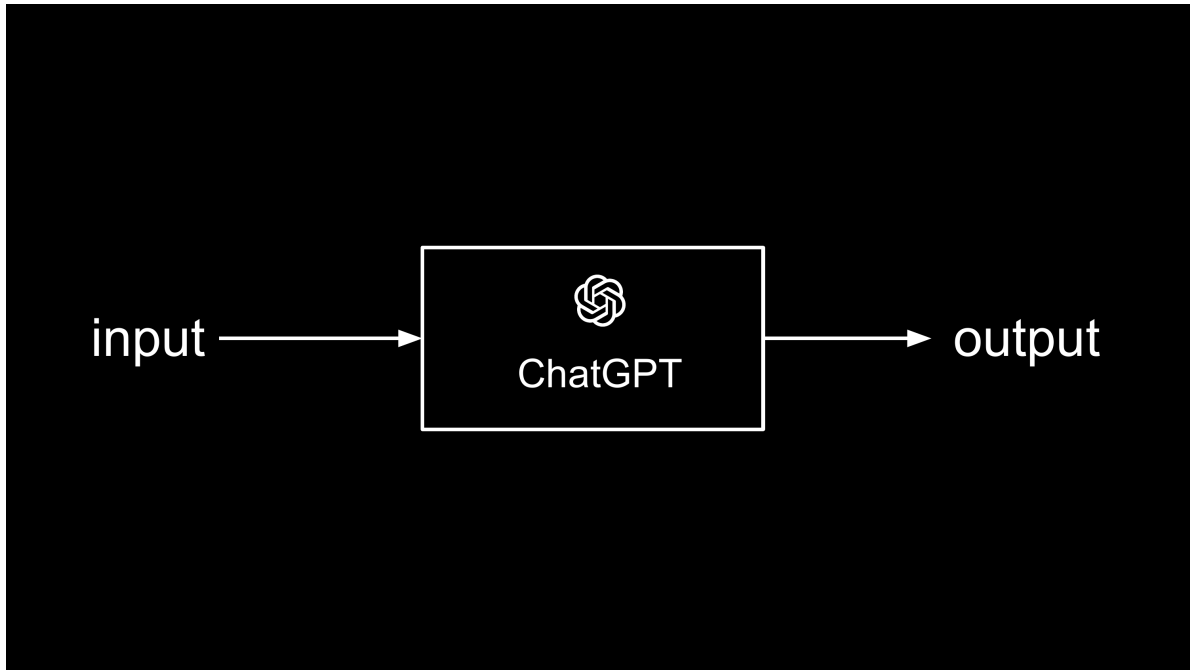


Abbildung 1.5.: ChatGPT im EVA-Modell.

Worte überhaupt in eine digitale Form überführen können. Auch dazu erfahren wir im späteren Verlauf des Buches mehr.

1.3.2. Die Lösung des Problems

Anhand des EVA-Modells wird deutlich, dass wir dem Computer Informationen in Form von digitalen Daten bereitstellen müssen, mit denen er arbeiten kann. Was aber genau soll er damit machen? Hier kommt der mittlere Kasten des Modells ins Spiel – die eigentliche Lösung des Problems.

a model to represent problems



In der Informatik nennen wir die Beschreibung zur Lösung eines Problems einen **Algorithmus**. Ein Algorithmus ist eine Schritt-für-Schritt-Anleitung zur Problemlösung und ist zunächst unabhängig von Computern. Das bedeutet, wir können die Lösung eines Problems ohne Bezug zu einem Computer beschreiben und nennen das einen Algorithmus.

Stellt euch dazu zum Beispiel eine IKEA-Aufbauanleitung vor. Sie beschreibt in sequenziellen Schritten, was zu tun ist, um das fertige Möbelstück zu bekommen. Die Eingabe besteht aus den mitgelieferten Teilen, Schrauben und dem benötigten Werkzeug für den Zusammenbau. Die Ausgabe ist das fertige Regal (oder ein anderes Möbelstück) – und das alles ganz ohne Computer.

Oder nehmt das Kochrezept eure Lieblingsessens. Auch ein Kochrezept ist ein Algorithmus: Die Eingabe besteht aus den Zutaten und Küchenutensilien, die Verarbeitung erfolgt durch die Schritt-für-Schritt-Anleitung, und die Ausgabe ist das fertige Gericht. Wie bei der IKEA-Anleitung ist der Algorithmus unabhängig von einem Computer – er beschreibt lediglich die Lösung des Problems “Wie koche ich dieses Gericht?”.

Zu Beginn des Kapitels haben wir festgestellt, dass Computer aufgrund ihrer Geschwindigkeit und Fehlerfreiheit besonders gut zur Problemlösung geeignet sind – insbesondere bei häufig wiederkehrenden Problemen. Die beiden genannten Beispiele, das IKEA-Regal und das Kochrezept, eignen sich allerdings nicht für eine direkte Umsetzung durch Computer. Der Grund liegt in den analogen Eingaben (Baumaterial, Kochzutaten) und Ausgaben (Möbelstück, fertige Mahlzeit). Diese kann ein Computer nicht unmittelbar verarbeiten. Dafür wären Roboter nötig, die mit der physischen Welt interagieren können. Eine solche Automatisierung lohnt sich

heute nur bei Aufgaben, die sehr häufig auftreten und ansonsten mit hohen Kosten verbunden sind – wie etwa in der Automobilindustrie, wo computergesteuerte Roboter in der Produktion zum Einsatz kommen.

Nehmen wir an, dass Haushaltsroboter in Zukunft erschwinglich werden und uns beim Kochen unseres Lieblingsgerichts helfen können. Wie vermitteln wir dann dem Roboter – im Grunde ein Computer mit Armen und Beinen – den Algorithmus für unser Rezept? Die Lösung liegt in der Programmierung: Wir erstellen ein **Programm** in einer computerverständlichen Sprache. Dieses Programm wandelt die Anweisungen aus unserem Kochbuch in Befehle um, die der Computer verstehen und ausführen kann. Genau genommen besteht ein Programm ebenfalls nur aus Informationen, und wir müssen herausfinden, wie wir diese Informationen digital darstellen können. Die Lösung besteht in der Verwendung einer **Programmiersprache** wie Python, die wir später kennenlernen werden.

1.4. Welche Strategien helfen bei der Lösung von Problemen?

Die konkrete Lösung und der zugehörige Algorithmus sehen für jedes Problem unterschiedlich aus. Das Erkennen von Pflanzen folgt einer anderen Logik als die Entscheidung, welche Schachfigur als nächstes gezogen werden soll. Dennoch gibt es universelle Lösungsstrategien, die auf viele Probleme anwendbar sind, um sie möglichst effizient zu lösen. Im Folgenden betrachten wir drei dieser Strategien.

1.4.1. Problemzerlegung (*Problem Decomposition*)

Eine universelle Strategie zur Lösung komplexer Probleme ist die Zerlegung in kleinere Schritte oder Teilprobleme. Jedes dieser Teilprobleme ist unterschiedlich und erfordert einen spezifischen Lösungsansatz. Nehmen wir als Beispiel das Zählen von Pflanzen auf einer Drohnenaufnahme. Dieses komplexe Problem lässt sich, ausgehend von der Eingabe – dem digitalen Bild – in drei Teilprobleme zerlegen:

1. Pflanzen auf dem Bild lokalisieren
2. Lokalisierte Pflanzen klassifizieren: Maispflanze oder nicht?
3. Identifizierte Maispflanzen zählen



Jedes dieser Teilprobleme erfordert einen eigenen Algorithmus. Dabei ist es möglich, die Teilprobleme noch weiter zu zerlegen, um sie besser bearbeiten zu können.

even smaller problem of type A	even smaller problem of type A
even smaller problem of type A	even smaller problem of type A
even smaller problem of type A	even smaller problem of type A
even smaller problem of type A	even smaller problem of type A

Die Identifizierung sinnvoller Teilprobleme erfordert ein ausgeprägtes analytisches Denkvermögen. Dies ist besonders wichtig im Umgang mit Computern. Wie wir beim Erlernen der Programmierung sehen werden, ist die Zerlegung eines Problems in kleine, lösbare Schritte der Schlüssel zur Beherrschung seiner Komplexität.

1.4.2. Teile und Herrsche (*Divide and Conquer*)

Die “Teile und Herrsche”-Strategie ist ein Ansatz zur Lösung komplexer Probleme, bei dem wir das Hauptproblem schrittweise in immer kleinere Teilprobleme zerlegen, bis diese einfach zu lösen sind. Dabei gehen wir rekursiv vor: Wir teilen das ursprüngliche Problem in kleinere Teile, diese kleineren Probleme wiederum in noch kleinere Teile und so weiter. Die Rekursion endet, wenn die Probleme so klein sind, dass sie sich nicht weiter aufteilen lassen und die Lösung direkt ersichtlich ist.

Anders als bei der Problemzerlegung sind die Teilprobleme beim Divide and Conquer-Ansatz dadurch gleichartig und stellen nur kleinere Instanzen des ursprünglichen Problems dar. Die einzelnen Lösungen für jedes Teilproblem werden dann schrittweise wieder zusammengeführt, um die Gesamtlösung zu erhalten. Ein klassisches Beispiel ist die Sortierung einer langen Liste von Zahlen: Wir teilen die Liste immer wieder in der Mitte, bis nur noch einzelne Zahlen übrig sind, und fügen diese dann in sortierter Reihenfolge wieder zusammen.

Ein anderes Beispiel ist die binäre Suche in einer sortierten Liste. Hier betrachten wir das Element in der Mitte der Liste und vergleichen es mit dem gesuchten Element. Da die Liste sortiert ist, können wir entscheiden, in welchem Teil der Liste wir weitersuchen müssen. Im zweiten Schritt suchen wir nur in diesem Teil weiter und haben damit das Problem halbiert. Die Natur des Problems bleibt dabei gleich, und wir können erneut genauso verfahren – so lange, bis wir nur noch ein Element übrig haben, das entweder das gesuchte Element ist oder nicht.

binary search

67 != 41



2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97

1.4.3. Verteile und Parallelisiere (*Distribute and Parallelize*)

Manche Probleme lassen sich effizienter lösen, wenn mehrere Personen gleichzeitig daran arbeiten. Anstatt eine einzelne Person mit der gesamten Aufgabe zu betrauen, verteilen wir die Arbeit auf mehrere Schultern und arbeiten parallel an der Lösung. Allerdings eignet sich nicht jedes Problem für diesen Ansatz.

Ein gutes Beispiel ist das Aufräumen eines Zimmers: Eine einzelne Person müsste nacheinander verschiedene Bereiche aufräumen, während mehrere Personen gleichzeitig unterschiedliche Ecken des Raums in Angriff nehmen können. Je größer das Zimmer, desto mehr Personen werden benötigt, um es in der gleichen Zeit aufzuräumen. Ein Gegenbeispiel, bei dem diese Strategie nicht funktioniert, ist das Lösen einer mathematischen Gleichung. Hier müssen die einzelnen Rechenschritte aufeinander aufbauen, weshalb das Problem nicht gleichzeitig an mehrere Personen übergeben werden kann, die unabhängig daran arbeiten.

Die Strategie des Verteilens und Parallelisierens – im Englischen *Distribute and Parallelize* – funktioniert nach einem klaren Prinzip: Wir zerlegen ein großes Problem in Teile, die unabhängig voneinander gelöst werden können. Diese Teile weisen wir dann verschiedenen Ressourcen zu – zum Beispiel mehreren Personen oder Computern. Jede Ressource arbeitet an ihrem Teilproblem und erzeugt ein eigenes Ergebnis. Dabei gehen wir davon aus, dass sich alle Teilergebnisse am Ende zu einer Gesamtlösung zusammenfügen lassen. Ähnlich wie beim Divide and Conquer-Ansatz sind die Teilprobleme meist gleichartig.

Um dieses Konzept greifbar zu machen, schauen wir uns ein konkretes Beispiel an, das wir mit dem EVA-Modell analysieren: das Zählen aller Wörter in einem Buch.



Abbildung 1.6.: Das Wörterzählen-Problem im EVA-Modell

Je nach Umfang des Buches kann dies eine mühsame Aufgabe sein, besonders für Menschen. Ein Computer bewältigt ein einzelnes Buch dank seiner hohen Verarbeitungsgeschwindigkeit problemlos. Allerdings lässt sich das Problem beliebig erweitern – etwa wenn wir statt eines Buches alle Texte im Internet oder sämtliche Wikipedia-Artikel analysieren möchten. In solchen Fällen wird die Aufgabe auch für Computer aufwendig und zeitintensiv. Eine Lösung besteht darin, mehrere Computer parallel einzusetzen.

In Abbildung 1.7 sehen wir beispielhaft die Verteilung der Buchseiten auf vier Studenten. Jeder erhält einen gleichen Anteil, wodurch sich die Bearbeitungszeit im Optimalfall auf ein Viertel reduziert. Bei Computern können wir analog vorgehen und mehrere Rechner gleichzeitig mit Teilen der Seiten betrauen. Diese Rechner werden in einem Netzwerk verbunden und von einem zentralen Computer gesteuert, der die Teilergebnisse am Ende zusammenführt. Ein solches System nennen wir Rechencluster, bestehend aus Arbeitern – den *Worker Nodes* – sowie einer Steuereinheit, die in solchen Systemen als *Driver* oder *Name Node* bezeichnet wird.

Durch die Verteilung und parallele Ausführung kann das EVA-Modell wie in Abbildung 1.8 angepasst und detaillierter dargestellt werden. Statt eines einzelnen Prozesses `count_words` laufen nun n parallele Prozesse, die jeweils einen Teil des Buches durchsuchen. Die Aufteilung



Abbildung 1.7.: Verteiltes Wörterzählen

erfolgt zu Beginn durch den **split**-Prozess, während das Zusammenführen der Teilergebnisse – in diesem Fall das Addieren der Teilsummen – durch den **merge**-Schritt erfolgt.

In diesem Kapitel haben wir uns mit dem EVA-Modell auseinandergesetzt - einem fundamentalen Konzept für die computergestützte Problemlösung. Dieses Modell bietet uns einen strukturierten Rahmen, der aus drei wesentlichen Komponenten besteht:

- **Eingabe (E)**: Die zu verarbeitenden Daten oder Informationen
- **Verarbeitung (V)**: Der Kern der Problemlösung durch Algorithmen
- **Ausgabe (A)**: Das Ergebnis der Verarbeitung in nutzbarer Form

Die Verarbeitung als zentrales Element des Modells ist dabei der Ort, an dem die eigentliche Problemlösung stattfindet. Hier kommen Algorithmen zum Einsatz - präzise Handlungsanweisungen, die Schritt für Schritt zur Lösung führen. Die genaue Natur dieser Algorithmen, ihre charakteristischen Eigenschaften und wie wir sie entwickeln können, werden wir im nächsten Kapitel detailliert betrachten.



Abbildung 1.8.: Das parallelisierte Wörterzählen im EVA-Modell

Übungsaufgaben

1. Formuliert drei weitere Probleme nach dem vorgestellten EVA-Modell zur Problemlösung mit digitalen Computern. Beschreibt für jedes Problem die Eingabe, die Lösung und die Ausgabe.
2. Beschreibe einen einfachen Getränkeautomaten anhand des EVA-Modells. Überlege dir dabei, welche Eingaben er erhält, welche Entscheidungen während der Verarbeitung getroffen werden müssen und welche Ausgaben daraus resultieren.
3. Welche Herausforderungen entstehen, wenn wir ein Problem aus der analogen Welt mit einem Computer lösen möchten?
4. Überlegt für jedes der folgenden Probleme, wie ihr es in sinnvolle Teilprobleme im Sinne der Problemzerlegungsstrategie aufteilen könntet:
 - a. Die Prüfung im Fach Wirtschaftsinformatik erfolgreich bestehen
 - b. Ein Haus bauen
 - c. Einen Marathon laufen
 - d. Eine Weltreise planen
 - e. Eine Party organisieren

5. Analysiert die folgenden Probleme und untersucht, welche sich davon in Teilprobleme zerlegen lassen, die parallel gelöst werden können:
 - a. Literaturrecherche für eure Abschlussarbeit
 - b. Repräsentative Befragung zur Bundestagswahl
 - c. Ein Haus bauen
 - d. Mais ernten
 - e. Eine Hochzeit planen
6. Erläutere die Vor- und Nachteile der linearen Suche in einer beliebigen Liste. Wie schätzt du die Effizienz dieser Methode ein?
7. Erkläre den *Divide and Conquer*-Ansatz mit deinen eigenen Worten! Welche Voraussetzung muss ein Problem erfüllen, damit es mit diesem Ansatz lösbar ist?

2. Algorithmen

Zusammenfassung

Algorithmen sind ein fundamentales Konzept der Informatik und der digitalen Problemlösung. In diesem Kapitel befassen wir uns mit den grundlegenden Aspekten von Algorithmen und beantworten dabei zentrale Fragen:

- Was ist ein Algorithmus und wie grenzt er sich von einem Computerprogramm ab?
- Wie könne wir Algorithmen darstellen?
- Welche verschiedenen Arten von Algorithmen existieren und durch welche Beispiele lassen sie sich veranschaulichen?
- Wie können wir Algorithmen systematisch und präzise formulieren?
- Welche Herausforderungen begegnen uns beim Entwurf von Algorithmen?
- Nach welchen Kriterien bewerten wir die Effizienz und Eignung verschiedener Algorithmen für spezifische Problemstellungen?

2.1. Was ist ein Algorithmus?

In Abbildung 2.1 sehen wir das aus dem vorherigen Kapitel bekannte EVA-Modell. Algorithmen stehen im Zentrum des Modells, da sie für die Lösung von Problemen – mit und ohne Computer – entscheidend sind. Sie beschreiben, wie ein Problem auf Basis der Eingabe gelöst werden soll und erzeugen die Ausgabe, also die Lösung selbst.

2.1.1. Herkunft des Begriffs

Der Begriff *Algorithmus* hat eine [interessante geschichtliche Herkunft](#). Er leitet sich vom Namen eines persischen Mathematikers des 9. Jahrhunderts ab: *Abu Dschaʿfar Muhammad ibn Mūsā al-Chwārizmī*. Dieser Gelehrte schrieb ein Buch über Rechenverfahren mit den damals neuen indisch-arabischen Ziffern. In lateinischer Übersetzung des Werks wurde aus seinem Namen “Algoritmi”, woraus sich das Wort *Algorismus* und schließlich *Algorithmus* entwickelte. Ursprünglich bezeichnete *Algorismus* tatsächlich die schriftlichen Rechenverfahren (Addition, Subtraktion etc.) mit arabischen Ziffern. Heutzutage verstehen wir unter einem Algorithmus



Abbildung 2.1.: Algorithmen im EVA-Modell.

aber jedes Verfahren, das nach festen Regeln abläuft – nicht nur Rechenverfahren. Die historische Anekdote zeigt jedoch, dass Algorithmen seit jeher mit systematischem **Problemlösen** und **Rechenvorschriften** verbunden sind.

Heute bezeichnet ein **Algorithmus** eine präzise, endliche Abfolge von Anweisungen, die ein bestimmtes Problem lösen oder eine Aufgabe erfüllen sollen. Im Alltag begegnen uns Algorithmen ständig, oft, ohne dass wir es merken: beim Kochen, bei der Wegbeschreibung oder beim Aufbau eines IKEA-Regals.

2.1.2. Algorithmen und Programme

Ein wichtiger Aspekt von Algorithmen ist ihre Universalität: Sie sind nicht an Computer gebunden. Ein Algorithmus ist im Kern eine strukturierte Anleitung zur Problemlösung, unabhängig davon, wer oder was diese Anleitung ausführt. Diese Flexibilität zeigt sich besonders deutlich in unserem Alltag, wo wir ständig algorithmische Anleitungen befolgen - sei es beim Aufbau eines Möbelstücks oder beim Kochen nach einem Rezept. Bei diesen Tätigkeiten führen wir Menschen die algorithmischen Schritte aus, ganz ohne Beteiligung eines Computers:

- Kochen: Ein Rezept ist ein Algorithmus für die Zubereitung eines Gerichts.
- Wegbeschreibung: Eine Schritt-für-Schritt-Anleitung, um von Punkt A nach Punkt B zu gelangen.

- **Bastelanleitung:** Die Anweisungen, um ein Modellflugzeug zusammenzubauen.

Viele Algorithmen können von Computern ausgeführt werden. Dafür ist jedoch eine Übersetzung in eine maschinenverständliche Form notwendig. Diese Übersetzung erfolgt durch das Programmieren, wobei wir den Algorithmus in einer Programmiersprache formulieren. Um die Beziehung zwischen Algorithmen und Computerprogrammen besser zu verstehen, ist es hilfreich, drei zentrale Begriffe zu unterscheiden:

- **Algorithmus:** Die abstrakte Beschreibung einer Lösungsmethode in Form einer präzisen, endlichen Sequenz von individuellen Anweisungen. Ein Algorithmus ist unabhängig von der konkreten Umsetzung und kann sowohl von Menschen als auch von Maschinen ausgeführt werden.
- **Programm:** Die konkrete Implementation eines oder mehrerer Algorithmen in einer Programmiersprache. Das Programm übersetzt die abstrakten Anweisungen des Algorithmus in eine Form, die ein Computer verstehen und ausführen kann.
- **Prozess:** Die tatsächliche Ausführung eines Programms durch einen Computer. Dabei werden die programmierten Anweisungen Schritt für Schritt abgearbeitet, um das gewünschte Ergebnis zu erzielen.

Im Folgenden konzentrieren wir uns zunächst auf das Konzept des Algorithmus an sich. Die praktische Implementierung in Form von Programmen werden wir später am Beispiel der Programmiersprache Python kennenlernen. Um Algorithmen jedoch bereits jetzt systematisch beschreiben und analysieren zu können, benötigen wir geeignete Darstellungsformen und Notationen.

2.2. Wie stellen wir Algorithmen dar?

2.2.1. Natürliche Sprache

Die natürliche Sprache bietet eine intuitive Möglichkeit, Algorithmen zu beschreiben. Ein klassisches Beispiel hierfür sind Kochrezepte, die als informelle algorithmische Beschreibungen verstanden werden können. Diese Art der Darstellung folgt keinen festgelegten Regeln - jeder Autor kann die Anweisungen nach eigenem Ermessen formulieren. Entscheidend ist dabei nur, dass andere Menschen die Beschreibung *lesen* und *verstehen* können.

Für die professionelle Informatik ist diese informelle Darstellungsform jedoch nur bedingt geeignet. Während handschriftliche oder natürlichsprachliche Notizen für erste Entwürfe und Skizzen durchaus nützlich sein können, erfordern die präzise Dokumentation und der fachliche Austausch über Algorithmen eine formellere Notation.

Die natürliche Sprache weist für die präzise Beschreibung von Algorithmen drei wesentliche Einschränkungen auf:

Erstens ist sie inhärent mehrdeutig: Ein und derselbe Satz kann je nach Kontext und Interpretation unterschiedliche Bedeutungen haben. Dies steht im Widerspruch zu der Eindeutigkeit, die Algorithmen erfordern. Das Wort “Bank” ist ein klassisches Beispiel für Mehrdeutigkeit im Deutschen. Es kann sich auf eine Sitzgelegenheit zum Ausruhen oder auf ein Finanzinstitut beziehen. Der Satz “Gehe zur Bank” könnte also zwei völlig unterschiedliche Bedeutungen haben - soll Geld abgehoben oder sich hingesetzt werden?

Zweitens setzt die natürliche Sprache oft implizites Wissen und Erfahrung voraus. Ein Kochrezept kann beispielsweise die Anweisung “Backe den Kuchen, bis er goldbraun ist” enthalten. Für einen Algorithmus ist diese Anweisung problematisch, da sie keine messbaren Kriterien enthält und stattdessen menschliches Urteilsvermögen voraussetzt. Ein Algorithmus benötigt exakte Angaben, wie zum Beispiel eine Zeitdauer und eine Temperatur. Eine bessere Version im Sinne eines Algorithmus wäre demnach: “Backe den Kuchen bei 180° C im vorgeheizten Ofen für 80 Minuten.”.

Drittens neigt die natürliche Sprache zu Weitschweifigkeit und Redundanz. Dies erschwert die präzise und effiziente Kommunikation algorithmischer Abläufe. Um diese Herausforderungen zu bewältigen, haben sich in der Informatik zwei formellere und präzisere Darstellungsformen etabliert: Pseudocode und Flussdiagramme.

2.2.2. Pseudocode

Pseudocode ist eine strukturierte, programmiersprachenähnliche Notation zur Beschreibung von Algorithmen. Er ist eine Art Zwischenlösung zwischen natürlicher Sprache und Programmiersprache und kombiniert Elemente der natürlichen Sprache mit grundlegenden Programmierkonzepten wie Schleifen, Bedingungen und Funktionen. Der Vorteil des Pseudocodes liegt in seiner Präzision und Klarheit, ohne dabei an die strengen syntaktischen Regeln einer echten Programmiersprache gebunden zu sein.

Ein wichtiges Merkmal des Pseudocodes ist seine Flexibilität: Er kann je nach Bedarf formeller oder informeller gestaltet werden, solange die grundlegende Logik und Struktur des Algorithmus klar erkennbar bleiben. Dabei werden häufig standardisierte Schlüsselwörter wie “IF”, “THEN”, “WHILE” oder “REPEAT” verwendet, die die algorithmische Struktur verdeutlichen.

2.2.3. Flussdiagramme

Flussdiagramme bieten eine visuelle Darstellung von Algorithmen durch standardisierte grafische Symbole und Verbindungslinien. Diese Notation ist besonders hilfreich, um den Ablauf eines Algorithmus und die logischen Verzweigungen auf einen Blick zu erfassen. Abbildung 2.3 zeigt die wichtigsten Elemente eines Flussdiagramms: Start- und Endpunkte, Ein- und Ausgaben (Parallelogramm), Anweisungen (Rechteck), Entscheidungen (Rauten), Wiederholungen (Sechseck).

```

READ a, b

REPEAT

    IF a < b THEN
        a = b
        b = a

    a = a - b

UNTIL a = 0 OR b = 0

RETURN the variable that is not 0

```

Abbildung 2.2.: Der Euklid'sche Algorithmus als Pseudocode.

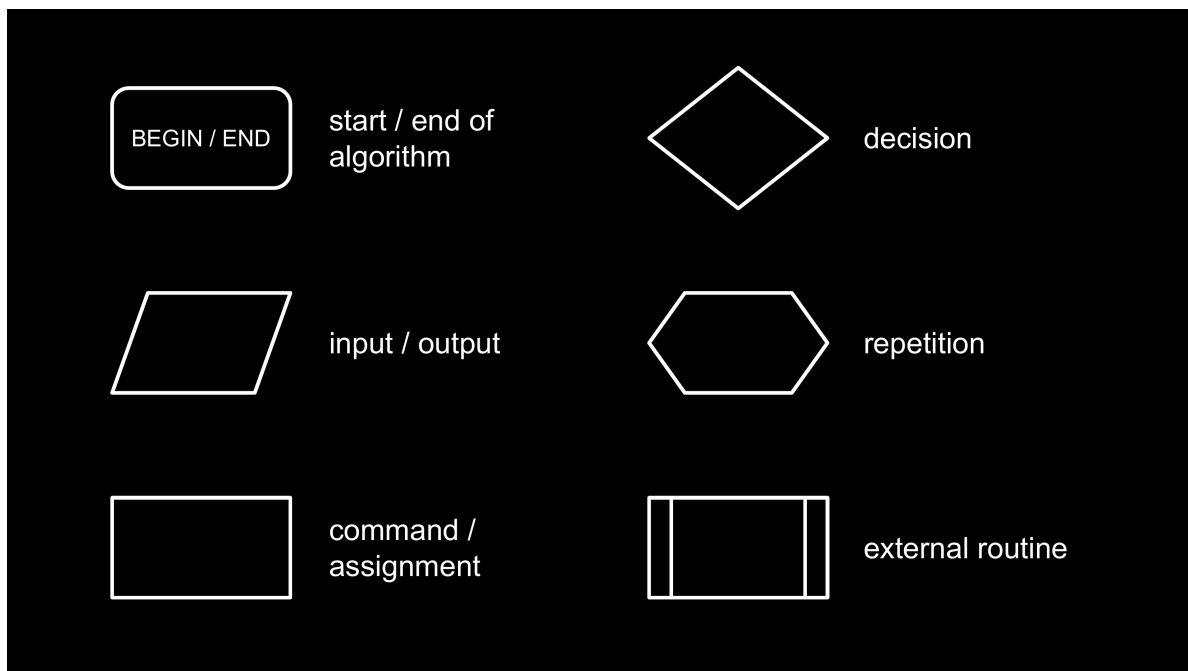


Abbildung 2.3.: Die Notationselemente des Flussdiagramms.

Dazu kommen Verbindungspfeile, die den Kontrollfluss anzeigen. Diese sind im Beispiel in Abbildung 2.4 zu sehen.



Abbildung 2.4.: Der Euklid'sche Algorithmus als Flussdiagramm.

Eine Alternative zu Flussdiagrammen sind die Struktogramme, die auch [Nassi-Shneiderman-Diagramme](#) genannt werden. Diese sind insbesondere im akademischen Bereich populär, werden hier aber nicht weiter betrachtet.

2.2.4. Programmcode

Die praktische Umsetzung eines Algorithmus erfolgt typischerweise als Computerprogramm, das in einer spezifischen Programmiersprache wie Python, Java oder C++ geschrieben wird. Der resultierende **Code** stellt eine maschinen ausführbare Repräsentation des Algorithmus dar. Dabei ist zu beachten, dass der Code strikt an die Syntax und Regeln der jeweiligen Programmiersprache gebunden ist. Derselbe Algorithmus kann in verschiedenen Programmiersprachen implementiert werden, wobei die konkreten Programme sehr unterschiedlich aussehen können. Diese Unterscheidung führt zu einer wichtigen konzeptionellen Trennung, die wir oben bereits gesehen haben: Der Algorithmus beschreibt die abstrakte Lösungsstrategie, während das Programm die spezifische technische Realisierung in einer bestimmten Programmiersprache darstellt.

2.3. Welche Arten von Algorithmen gibt es?

Algorithmen können nach ihrer grundlegenden Vorgehensweise oder ihrem Anwendungsbereich in verschiedene Kategorien eingeteilt werden. Jede Kategorie repräsentiert einen spezifischen Problemlösungsansatz:

- **Mathematische Algorithmen:** Berechnen oder approximieren Werte
- **Suchalgorithmen:** Finden bestimmte Elemente in einer Datenmenge
- **Sortieralgorithmen:** Ordnen Daten nach bestimmten Kriterien
- **Optimierungsalgorithmen:** Finden die bestmögliche Lösung für ein Problem
- **Graphenalgorithmen:** Arbeiten mit vernetzten Strukturen
- **Stochastische Algorithmen:** Verwenden Zufallselemente, um ein Problem zu lösen
- **Maschinelle Lernalgorithmen:** Erkennen Muster und treffen Vorhersagen

Diese Kategorien sind weder vollständig noch strikt voneinander getrennt. Viele Algorithmen lassen sich mehreren Kategorien zuordnen. Ein anschauliches Beispiel hierfür ist der **Dijkstra-Algorithmus**, der die kürzeste Route zwischen zwei Punkten findet. Er ist sowohl ein Graphenalgorithmus, da er auf vernetzten Strukturen arbeitet, als auch ein Optimierungsalgorithmus, da er die optimale (kürzeste) Route ermittelt.

Im folgenden beleuchten wir ein oder mehr Beispiele für jeder der genannten Klassen.

2.3.1. Mathematische Algorithmen

2.3.1.1. Größter gemeinsamer Teiler (GGT)

Der Algorithmus zur Berechnung des größten gemeinsamen Teilers (GGT) ist ein klassisches Beispiel für einen eleganten mathematischen Algorithmus. Er wurde vom griechischen Mathematiker Euklid um 300 v. Chr. in seinem Werk “Die Elemente” beschrieben und demonstriert eindrucksvoll die zeitlose Natur algorithmischen Denkens.

Das Verfahren basiert auf einem einfachen, aber genialen Prinzip: Der GGT zweier Zahlen ist identisch mit dem GGT der kleineren Zahl und der Differenz beider Zahlen (Abbildung 2.5). Zum Beispiel haben die Zahlen 48 und 18 den gleichen GGT wie 18 und 30 (48-18). Durch wiederholtes Anwenden dieser Regel wird der GGT systematisch ermittelt. Die Eleganz dieses Verfahrens liegt in seiner Einfachheit und mathematischen Präzision - Eigenschaften, die auch heute noch moderne Algorithmen auszeichnen.

Abbildung 2.6 zeigt die Schritte des Euklidschen Algorithmus für das obige Zahlenbeispiel.

Identify the larger number. If $a < b$, swap numbers so that $a > b$

Subtract b from a and replace a with the result

Repeat until one of the numbers becomes 0

Return the number that is not zero

Abbildung 2.5.: Vorgehen des Algorithmus nach Euklid.

```
Loop 1:  
a = 18, b = 48 → swap → a = 48, b = 18  
a = 48 - 18 = 30
```

```
Loop 2:  
a = 30, b = 18 → no swap  
a = 30 - 18 = 12
```

```
Loop 3:  
a = 12, b = 18 → swap → a = 18, b = 12  
a = 18 - 12 = 6
```

```
Loop 4:  
a = 6, b = 12 → swap → a = 12, b = 6  
a = 12 - 6 = 6
```

```
Loop 5:  
a = 6, b = 6 → no swap  
a = 6 - 6 = 0
```

```
return b = 6
```

Abbildung 2.6.: Beispiel für die Anwendung des Algorithmus nach Euklid.

2.3.1.2. Babylonisches Wurzelziehen

Das Babylonische Verfahren zur Berechnung der Quadratwurzel ist ein weiteres Beispiel für einen mathematischen Algorithmus. Während der Euklid'sche Algorithmus exakte Ergebnisse liefert, zeigt das Babylonische Verfahren eine andere wichtige Eigenschaft mathematischer Algorithmen: **die schrittweise Annäherung an einen Zielwert**. Der Algorithmus *approximiert* die Quadratwurzel durch wiederholte Verfeinerung der Schätzung und *konvergiert* dabei gegen den tatsächlichen Wert. Diese Methode demonstriert, wie auch ohne exakte Berechnung präzise Ergebnisse erzielt werden können.

Ein anschauliches Beispiel für das Babylonische Verfahren ist in Abbildung 2.7 dargestellt. Der Algorithmus berechnet die Quadratwurzel einer Eingabezahl durch geometrische Annäherung: Wenn wir die Eingabezahl x als Flächeninhalt eines Rechtecks interpretieren, suchen wir die Seitenlängen eines Quadrats mit der gleichen Fläche.

Das Verfahren nähert sich diesem Wert schrittweise an, indem es die Seitenlängen A und B eines Rechtecks iterativ anpasst. Nehmen wir an, wir wollen die Quadratwurzel aus $x = 16$ ziehen. Zu Beginn setzen wir $A = 1$ und berechnen B so, dass das Produkt der Seitenlängen die gesuchte Fläche ergibt: $B = \frac{x}{A} = 16$. Dieser Ausgangszustand ist in Abbildung 2.7 dargestellt.



Abbildung 2.7.: Ausgangszustand im Babylonischen Wurzelziehen.

Nach der Festlegung der Startwerte beginnt der eigentliche iterative Prozess der Annäherung. In jedem Durchlauf werden die Werte für A und B nach spezifischen Formeln neu berechnet,

wodurch sie sich schrittweise einander annähern:

$$A = \frac{A + B}{2}$$

Und damit die Gesamtfläche der gesuchten Zahl x entspricht, setzen wir B wie folgt:

$$B = \frac{x}{A}$$

Wenden wir diese Formeln auf unsere Ausgangswerte an, ergibt sich für den zweiten Iterationsschritt das in Abbildung 2.8 visualisierte Rechteck. Hier berechnen wir zunächst die neue Länge A:

$$A = \frac{1 + 16}{2} = 8.5$$

Und daraus folgt für B:

$$B = \frac{16}{8.5} \approx 1.88$$



Abbildung 2.8.: Zustand des Babylonischen Verfahrens nach dem ersten Schritt.

Verglichen mit dem Ausgangszustand haben sich die Werte A und B bereits angenähert, aber sie weichen noch deutlich voneinander ab. Um eine bessere Approximation zu erreichen, führen wir eine weitere Iteration durch und berechnen die neuen Werte für A und B nach dem etablierten Schema:

$$A = \frac{8.5 + 1.88}{2} \approx 5.19$$

Und für die andere Kante:

$$B = \frac{16}{5.19} \approx 3.08$$



Abbildung 2.9.: Zustand des Babylonischen Verfahrens nach dem zweiten Schritt.

Die Werte haben sich weiter angenähert. Da wir wissen, dass beide Kanten am Ende die Länge 4 haben sollen, ist eine weitere Iteration erforderlich. Nach der dritten Berechnung, wie in [Abbildung 2.10](#) dargestellt, nähern sich beide Kanten bereits sehr präzise dem Zielwert von 4 an.

Nach einer weiteren Iteration konvergiert der Algorithmus zum gesuchten Wert: Beide Seiten A und B erreichen durch Rundung den Wert 4. Damit haben wir das gesuchte Quadrat erfolgreich konstruiert, und der Algorithmus kann beendet werden.

$$A = (A + B) / 2 \approx 8.27 / 2 \approx 4.14$$

$$B = x / A \approx 16 / 4.14 \approx 3.86$$

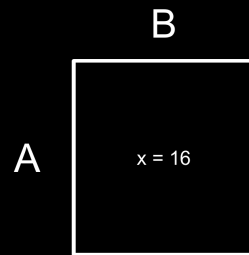
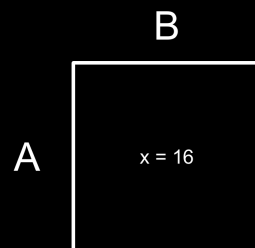


Abbildung 2.10.: Zustand des Babylonischen Verfahrens nach dem dritten Schritt.

$$A = (A + B) / 2 = 8 / 2 = 4$$

$$B = x / A = 16 / 4 = 4$$



In diesem Beispiel konvergiert der Algorithmus bereits nach wenigen Iterationen zum korrekten

Ergebnis. Die Geschwindigkeit der Konvergenz hängt jedoch von zwei wesentlichen Faktoren ab:

1. Der Eingabezahl, aus der die Wurzel gezogen werden soll
2. Der Wahl der Startwerte für A und B

Je nach Konstellation dieser Faktoren kann die Annäherung deutlich mehr Iterationen benötigen. Zudem erreichen die beiden Seiten in vielen Fällen keine exakte Gleichheit, sondern nähern sich lediglich bis auf eine beliebig kleine Differenz an. Dies wirft eine wichtige praktische Frage auf: Wann ist die Approximation der Quadratwurzel präzise genug, um den Algorithmus zu beenden? Dazu später in Kapitel 2.5.1 mehr.

Ein pragmatisches Abbruchkriterium für das Babylonische Verfahren ist das Erreichen einer vordefinierten Genauigkeit. Nach jedem Iterationsschritt wird die Differenz zwischen den Werten A und B berechnet. Ist diese Differenz kleiner als ein festgelegter Schwellenwert, wird der Algorithmus beendet. Der vollständige Ablauf des Verfahrens ist in Abbildung 2.11 dargestellt.

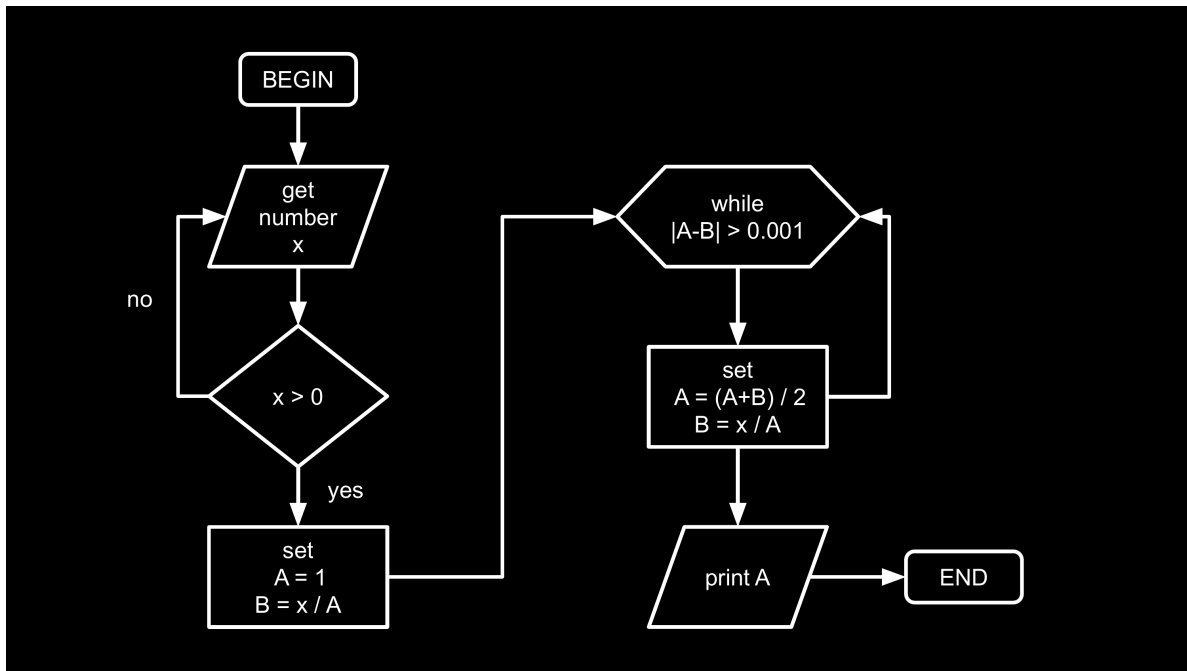


Abbildung 2.11.: Das Babylonische Verfahren als Flussdiagramm.

Das Flussdiagramm beginnt auf der linken Seite mit einer Eingabeprüfung: Die Zahl x wird eingelesen und auf Positivität getestet, da die Quadratwurzel aus negativen Zahlen nicht definiert ist. Eine Verzweigung überprüft die Bedingung $x > 0$ und leitet den Algorithmus entsprechend weiter. Bei positiver Eingabe werden die Startwerte für A und B initialisiert. Anschließend beginnt eine WHILE-Schleife, die solange durchlaufen wird, bis die Differenz zwischen A und B

einen definierten Schwellenwert unterschreitet. Nach erfolgreicher Konvergenz wird der finale Wert von A als Ergebnis ausgegeben.

Eine weitere wichtige Frage bei der Entwicklung von Algorithmen betrifft deren Terminierung: Wir müssen nicht nur festlegen, wann ein Algorithmus beendet werden soll, sondern auch sicherstellen, dass er überhaupt ein Ende erreicht. Am Beispiel des Babylonischen Verfahrens lässt sich dies gut veranschaulichen: Die entscheidende Frage ist, ob die Werte A und B tatsächlich konvergieren, also sich einander systematisch annähern.

In diesem Fall können wir die Frage positiv beantworten. Die Berechnungsformel, die in jedem Schritt den neuen Wert für A als Durchschnitt aus A und B bestimmt, garantiert eine stetige Annäherung der beiden Werte. Diese mathematische Eigenschaft sichert die Terminierung des Algorithmus. Bei anderen Algorithmen muss die Terminierung jedoch sorgfältig analysiert werden, um unendliche Ausführungen zu vermeiden. Mehr dazu in Kapitel [2.5.2](#).

2.3.2. Suchalgorithmen

Nach der Betrachtung mathematischer Algorithmen wenden wir uns nun den Suchalgorithmen zu - einer fundamentalen Kategorie von Algorithmen, die ein essentielles Element der Informatikausbildung darstellt. Diese Algorithmen bilden die Grundlage für effiziente Datenzugriffe und sind damit von großer praktischer Bedeutung.

2.3.2.1. Lineare Suche

Die lineare Suche ist der einfachste Suchalgorithmus und folgt einem intuitiven Ansatz: Sie durchläuft eine Liste sequentiell von Anfang bis Ende, bis das gesuchte Element gefunden wird. Obwohl dieser Algorithmus konzeptionell sehr einfach ist, hat er den Nachteil einer relativ hohen Laufzeit, da im schlechtesten Fall die gesamte Liste durchsucht werden muss. Dennoch ist die lineare Suche für unsortierte Listen oder kleine Datenmengen eine praktikable Lösung.

2.3.2.2. Binäre Suche

Die binäre Suche ist ein hocheffizienter Algorithmus, der auf dem “Teile und Herrsche”-Prinzip (Divide & Conquer) basiert. Eine wichtige Voraussetzung für ihre Anwendung ist, dass die zu durchsuchende Liste sortiert vorliegt. Der Algorithmus arbeitet systematisch, indem er in jedem Schritt die Liste in der Mitte teilt und anhand eines Vergleichs entscheidet, ob sich das gesuchte Element im linken oder rechten Teilbereich befindet. Durch diese Halbierung des Suchbereichs in jedem Schritt erreicht die binäre Suche eine bemerkenswerte Effizienz.

Diese Strategie, die wir bereits im vorigen Kapitel 1.4.2 als *Divide-and-Conquer*-Ansatz kennengelernt haben, ermöglicht es der binären Suche, selbst in großen Datensätzen ein gesuchtes Element mit minimaler Anzahl von Vergleichsoperationen zu finden.

2.3.2.3. Suche in Bäumen

Die Suche in Baumstrukturen stellt eine weitere wichtige Variante der Suchalgorithmen dar. Binäre Suchbäume ermöglichen durch ihre hierarchische Struktur eine besonders effiziente Suche, da in jedem Knoten eine binäre Entscheidung getroffen wird, die den Suchbereich systematisch einschränkt. Diese Baumstrukturen kombinieren die Vorteile der binären Suche mit einer dynamischen Datenorganisation, die Einfüge- und Löschoperationen effizient unterstützt.

2.3.3. Sortieralgorithmen

Sortieralgorithmen bilden eine weitere fundamentale Kategorie von Algorithmen, die sich mit der systematischen Anordnung von Datenelementen in einer bestimmten Reihenfolge befassen. Die Bedeutung dieser Algorithmen liegt nicht nur in ihrer direkten Anwendung zur Sortierung von Daten, sondern auch in ihrer Rolle als Grundlage für effizientere Such- und Analyseverfahren. Im Folgenden betrachten wir drei klassische Sortieralgorithmen, die sich in ihrer Herangehensweise und Effizienz deutlich unterscheiden.

2.3.3.1. Bubblesort

Bubblesort ist ein einfacher Sortieralgorithmus, der paarweise benachbarte Elemente vergleicht und bei Bedarf vertauscht. Dieser Prozess wird solange wiederholt, bis keine Vertauschungen mehr notwendig sind und die Liste sortiert ist. Obwohl Bubblesort aufgrund seiner einfachen Implementierung und Verständlichkeit oft zu Lehrzwecken verwendet wird, ist er für große Datenmengen wegen seiner quadratischen Laufzeitkomplexität wenig effizient.

2.3.3.2. Selectionsort

Selectionsort ist ein weiterer elementarer Sortieralgorithmus, der die Liste schrittweise sortiert, indem er wiederholt das kleinste Element im unsortierten Teil der Liste findet und es an die korrekte Position im sortierten Teil verschiebt. Ähnlich wie Bubblesort ist auch dieser Algorithmus konzeptionell einfach zu verstehen, weist jedoch ebenfalls eine quadratische Laufzeitkomplexität auf. Der Vorteil gegenüber Bubblesort liegt in der geringeren Anzahl tatsächlicher Vertauschungsoperationen, da Elemente nur dann verschoben werden, wenn sie tatsächlich an ihre finale Position gebracht werden.

2.3.3.3. Mergesort

Mergesort ist ein effizienter Sortieralgorithmus, der auf dem Divide-and-Conquer-Prinzip basiert. Er teilt die zu sortierende Liste rekursiv in kleinere Teilsequenzen, bis diese nur noch ein Element enthalten, und führt diese dann schrittweise in sortierter Reihenfolge wieder zusammen. Im Gegensatz zu Bubblesort und Selectionsort erreicht Mergesort durch seine rekursive Strategie eine deutlich bessere Laufzeitkomplexität, was ihn besonders für große Datenmengen attraktiv macht.

2.3.4. Optimierungsalgorithmen

Optimierungsalgorithmen sind eine wichtige Klasse von Algorithmen, die darauf abzielen, die bestmögliche Lösung für ein gegebenes Problem zu finden. Diese Algorithmen sind besonders relevant in praktischen Anwendungen, wo optimale oder nahezu optimale Lösungen für komplexe Probleme gefunden werden müssen. Typische Beispiele finden sich in der Logistik, der Produktionsplanung oder der Ressourcenallokation. Die verschiedenen Ansätze zur Optimierung unterscheiden sich dabei hauptsächlich in ihrer Herangehensweise und Effizienz.

2.3.4.1. Brute-Force

Der Brute-Force-Ansatz ist die einfachste, aber auch rechenintensivste Methode der Optimierung. Dabei werden systematisch alle möglichen Lösungen durchprobiert, um die beste zu finden. Obwohl dieser Ansatz für kleine Probleme praktikabel sein kann, wird er bei wachsender Problemgröße schnell ineffizient, da die Anzahl der zu prüfenden Kombinationen exponentiell steigt.

2.3.4.2. Random-Sampling

Random-Sampling ist ein stochastischer Optimierungsansatz, der zufällig ausgewählte Lösungen aus dem Suchraum evaluiert. Im Gegensatz zur erschöpfenden Suche des Brute-Force-Verfahrens werden hier nur Stichproben untersucht, was den Rechenaufwand deutlich reduziert. Dieser Ansatz eignet sich besonders für große Suchräume, in denen eine vollständige Enumeration nicht praktikabel ist, kann aber nicht garantieren, das globale Optimum zu finden.

2.3.4.3. Evolutionäre Algorithmen

Evolutionäre Algorithmen sind Optimierungsverfahren, die auf den Prinzipien der biologischen Evolution basieren. Sie operieren mit einer Population von Lösungskandidaten und verbessern diese durch drei zentrale Mechanismen: Mutation (zufällige Veränderungen), Rekombination

(Kombination erfolgreicher Lösungen) und Selektion (Auswahl der besten Varianten). Der iterative Prozess aus Variation und Auswahl führt schrittweise zu besseren Lösungen.

Diese Algorithmen sind besonders effektiv bei komplexen Optimierungsproblemen, für die keine exakten Lösungsmethoden existieren oder diese zu rechenintensiv wären. Da die Mutation der Lösungskandidaten auf Zufallsprozessen basiert, gehören evolutionäre Algorithmen zur Klasse der stochastischen Verfahren.

2.3.4.4. Bayes'sche Optimierung

Die Bayes'sche Optimierung ist ein effizienter Ansatz zur Optimierung von komplexen Funktionen, der auf dem Bayes'schen Theorem basiert. Anders als klassische Optimierungsverfahren nutzt dieser Algorithmus probabilistische Modelle, um die Zielfunktion zu approximieren und vielversprechende Bereiche des Suchraums zu identifizieren. Besonders nützlich ist dieser Ansatz bei rechenintensiven Problemen, bei denen jede Auswertung der Zielfunktion zeit- oder kostenaufwendig ist.

2.3.5. Graphenalgorithmen

2.3.5.1. Dijkstra-Algorithmus

Der Dijkstra-Algorithmus, entwickelt 1956 von Edsger W. Dijkstra, ist ein fundamentaler Algorithmus zur Berechnung kürzester Pfade in gewichteten Graphen. Seine Funktionsweise basiert auf dem Prinzip der schrittweisen Optimierung: Ausgehend von einem Startknoten berechnet er systematisch die kürzesten Wege zu allen anderen Knoten im Graphen. Aufgrund seiner Effizienz und Zuverlässigkeit findet der Algorithmus heute breite praktische Anwendung, insbesondere in der Routenplanung, Netzwerkanalyse und Logistik.

2.3.5.2. Das Traveling Salesman-Problem

Das Traveling Salesman-Problem (TSP) gehört zu den bekanntesten Optimierungsproblemen der Graphentheorie. Die Aufgabenstellung ist dabei einfach zu verstehen: Ein Handelsreisender muss eine Route planen, die alle vorgegebenen Städte genau einmal besucht und am Ende zum Ausgangspunkt zurückführt. Das Ziel ist es, die kürzeste mögliche Route zu finden. Trotz dieser scheinbar einfachen Formulierung ist das TSP ein NP-schweres Problem - das bedeutet, dass bisher kein Algorithmus gefunden wurde, der für große Städt Mengen in angemessener Zeit die optimale Lösung garantiert berechnen kann.

2.3.6. Stochastische Algorithmen

2.3.6.1. Annäherung der Kreiszahl π

Die Monte-Carlo-Methode ist ein anschauliches Beispiel für stochastische Algorithmen. Sie nutzt Zufallszahlen zur Annäherung mathematischer Werte, wie etwa der Kreiszahl π . Das Verfahren platziert dabei zufällig Punkte in einem Quadrat mit einbeschriebenem Kreis. Das Verhältnis der Punkte innerhalb des Kreises zur Gesamtanzahl der Punkte ermöglicht eine Approximation von π . Ein wichtiges Merkmal dieser stochastischen Methode ist, dass die Genauigkeit der Berechnung mit der Anzahl der generierten Punkte steigt.

2.3.7. Maschinelle Lernalgorithmen

Maschinelles Lernen hat sich in den letzten Jahren zu einem der wichtigsten Bereiche der Algorithmenentwicklung entwickelt. Diese Algorithmenklasse bildet die technische Grundlage für viele praktische Anwendungen der Künstlichen Intelligenz (KI). Maschinelle Lernalgorithmen zeichnen sich dadurch aus, dass sie aus vorhandenen Daten Muster erkennen und auf dieser Basis Vorhersagen oder Entscheidungen treffen können.

2.4. Welche wichtigen algorithmischen Denkmuster gibt es?

2.4.1. Sequenzen

Die Sequenz ist das grundlegendste Muster in der Algorithmik. Sie beschreibt eine geordnete Abfolge von Anweisungen, die nacheinander ausgeführt werden. Wie bei einer Wegbeschreibung folgt dabei ein Schritt dem anderen in einer festgelegten Reihenfolge. Die korrekte und vollständige Ausführung aller Schritte ist entscheidend - das Auslassen oder Vertauschen von Anweisungen führt in der Regel nicht zum gewünschten Ergebnis.

Sowohl Menschen als auch Computer verarbeiten Anweisungen standardmäßig sequentiell - also Schritt für Schritt von oben nach unten. Diese intuitive Vorgehensweise bildet die Grundlage für das Verständnis von Algorithmen. Allerdings können Algorithmen auch komplexere Strukturen enthalten, die von diesem linearen Ablaufmuster abweichen und alternative Ausführungspfade ermöglichen.

2.4.2. Verzweigungen

Verzweigungen stellen eine grundlegende Form der nicht-linearen Ausführung eines Algorithmus dar. Sie ermöglichen es, dass der Algorithmus basierend auf bestimmten Bedingungen unterschiedliche Ausführungspfade einschlägt. Eine Verzweigung führt also je nach erfüllter oder

nicht erfüllter Bedingung zu verschiedenen Anweisungsfolgen. Ein alltägliches Beispiel findet sich in Kochrezepten: “Wenn der Teig zu flüssig ist, füge mehr Mehl hinzu”. Die Anweisung, mehr Mehl hinzuzufügen, wird nur dann ausgeführt, wenn die Bedingung “Teig ist zu flüssig” zutrifft. Ist der Teig bereits von idealer Konsistenz, wird diese Anweisung übersprungen.

2.4.3. Iterationen

Ein Beispiel für ein iteratives, schrittweise Vorgehen ist das Babylonische Wurzelziehen aus Kapitel [2.3.1.2](#).

2.4.4. Kapselung

Kapselung beschreibt das Prinzip, komplexe Algorithmen in kleinere, überschaubare Einheiten zu zerlegen. Diese Modularisierung erhöht nicht nur die Lesbarkeit und Wartbarkeit des Algorithmus, sondern ermöglicht auch die Wiederverwendung von Teilfunktionen in anderen Kontexten. Ein klassisches Beispiel ist die Auslagerung häufig benötigter Berechnungen in separate Funktionen, die dann an verschiedenen Stellen aufgerufen werden können.

2.4.5. Rekursion

Rekursion ist ein mächtiges algorithmisches Konzept, bei dem sich ein Algorithmus selbst aufruft, um ein Problem zu lösen. Dieses Prinzip eignet sich besonders gut für Probleme, die sich in kleinere, gleichartige Teilprobleme zerlegen lassen. Ein klassisches Beispiel ist die Berechnung der Fakultät einer Zahl, bei der sich die Lösung aus der Multiplikation mit der Fakultät der nächstkleineren Zahl ergibt. Aber auch die binäre Suche kann rekursiv implementiert werden, indem der Algorithmus den zu durchsuchenden Bereich in der Mitte teilt und sich selbst mit der relevanten Hälfte aufruft. Diese rekursive Struktur macht den Algorithmus nicht nur elegant, sondern auch besonders effizient.

2.5. Was sind Herausforderungen bei der Formulierung von Algorithmen?

2.5.1. Haltekriterium

Ein wichtiges Kriterium bei der Formulierung von Algorithmen ist die Bestimmung eines geeigneten Haltekriteriums. Das Haltekriterium definiert die Bedingung, unter der ein Algorithmus seine Ausführung beendet und ein Ergebnis zurückliefert. Beim Babylonischen Wurzelziehen beispielsweise ist das Haltekriterium erreicht, wenn die Differenz zwischen zwei aufeinanderfolgenden Näherungswerten einen bestimmten Schwellenwert unterschreitet.

Die Bestimmung eines geeigneten Haltekriteriums stellt für manche Algorithmen eine besondere Herausforderung dar. Dies zeigt sich besonders deutlich bei Optimierungsalgorithmen, wo die Frage nach dem optimalen Zeitpunkt für die Beendigung des Algorithmus nicht trivial ist. Da das theoretische Optimum in der Regel unbekannt ist, lässt sich schwer einschätzen, wie nah die aktuelle Lösung bereits am bestmöglichen Ergebnis liegt. Eine pragmatische Lösung für dieses Problem besteht in der Festlegung eines Optimierungsbudgets, das die maximale Anzahl der Durchläufe definiert, die der Algorithmus ausführen darf.

2.5.2. Endlosschleifen

Endlosschleifen stellen eine kritische Herausforderung bei der Entwicklung von Algorithmen dar. Sie entstehen, wenn ein Algorithmus das definierte Haltekriterium nie erreicht und stattdessen kontinuierlich dieselben Anweisungen wiederholt. Ein typisches Beispiel ist eine While-Schleife, deren Bedingung permanent wahr bleibt - der Algorithmus verbleibt dann in einem unendlichen Ausführungszyklus. Ohne geeignete Fehlerbehandlung führt dies meist zu einem Programmabsturz, da Systemressourcen wie Arbeitsspeicher oder Prozessorzeit erschöpft werden.

2.5.3. Beurteilung des Ergebnisses

Die Beurteilung der Qualität eines algorithmischen Ergebnisses ist nicht immer eindeutig und hängt stark vom jeweiligen Anwendungsfall ab. Bei numerischen Berechnungen lässt sich die Genauigkeit oft durch den Vergleich mit bekannten Referenzwerten oder theoretischen Grenzen bestimmen. Bei Optimierungsproblemen hingegen ist die Bewertung komplexer, da das theoretische Optimum häufig unbekannt ist und die Qualität der Lösung von verschiedenen, teils konkurrierenden Kriterien abhängt.

2.5.4. Erklärbarkeit des Ergebnisses

Die Erklärbarkeit algorithmischer Entscheidungen spielt eine zentrale Rolle in der modernen Informatik, besonders in kritischen Bereichen wie der Medizin oder Rechtsprechung. Dabei stellt die mangelnde Transparenz komplexer Algorithmen eine fundamentale Herausforderung dar: Der Prozess von der Eingabe bis zur Entscheidungsfindung ist oft nicht direkt nachvollziehbar. Dieses Problem manifestiert sich besonders deutlich bei Deep Learning Modellen, deren mehrschichtige Strukturen und komplexe Berechnungen eine intuitive Interpretation der Entscheidungswege erschweren. Die Entwicklung von Methoden zur besseren Erklärbarkeit dieser "Black Box"-Systeme ist daher ein aktives Forschungsfeld, das darauf abzielt, die Akzeptanz und Vertrauenswürdigkeit algorithmischer Entscheidungen zu erhöhen.

2.6. Gibt es bessere und schlechtere Algorithmen?

2.6.1. Komplexität

Die Komplexität eines Algorithmus beschreibt, wie sich sein Ressourcenbedarf (meist Zeit und Speicher) in Abhängigkeit von der Eingabegröße entwickelt. Diese mathematische Charakterisierung ermöglicht einen objektiven Vergleich verschiedener algorithmischer Lösungen für dasselbe Problem. Besonders wichtig ist dabei die asymptotische Komplexität, die das Verhalten des Algorithmus für große Eingabemengen beschreibt.

2.6.2. Verständlichkeit

Die Verständlichkeit eines Algorithmus ist ein weiteres wichtiges Qualitätsmerkmal. Ein gut strukturierter und dokumentierter Algorithmus erleichtert nicht nur die Wartung und Weiterentwicklung, sondern reduziert auch die Wahrscheinlichkeit von Fehlern bei der Implementierung. Dabei spielt die Wahl aussagekräftiger Bezeichner und eine klare Dokumentation der einzelnen Schritte eine zentrale Rolle.

Übungsaufgaben

1. Woher stammt der Begriff “Algorithmus”?
2. Definiere, was ein Algorithmus ist, und gib drei Beispiele für Algorithmen aus dem Alltag, die keinen direkten Bezug zu Computern haben.
3. Beobachte eine Woche lang Algorithmen in deinem Alltag (z.B. Navigationssysteme, Empfehlungssysteme, Suchmaschinen). Dokumentiere deren Eingaben, Verarbeitung und Ausgaben.
4. Welche grundlegenden Ansätze zur Klassifizierung von Algorithmen gibt es?
5. Erläutere, was mit der Komplexität eines Algorithmus gemeint ist. Warum ist die Komplexität eines Algorithmus wichtig? Wie wird sie angegeben?
6. Welche Komplexitätsklassen kennst du? Bringe sie in eine Reihenfolge von der geringsten zur höchsten Komplexität.
7. Vergleiche die Laufzeitkomplexität von linearer und binärer Suche anhand eines konkreten Beispiels mit einer Million sortierten Zahlen!
8. Berechne den größten gemeinsamen Teiler der Zahlen 56 und 98 mithilfe des euklidischen Algorithmus! Dokumentiere jeden Schritt!
9. Wir haben exemplarisch für einen Algorithmus die babylonische Methode zur Approximation einer Quadratwurzel kennengelernt. Beantworte die nachfolgenden Fragen in diesem Kontext:
 - a. Berechne die Quadratwurzel von 25 mit der babylonischen Methode und dokumentiere jeden Schritt! Wähle einen sinnvollen Startwert!
 - b. Vergleiche die Ergebnisse der babylonischen Methode nach 3, 5 und 7 Iterationen mit dem exakten Wert der Quadratwurzel.
 - c. Erkläre die Funktionsweise des babylonischen Algorithmus zur Berechnung der Quadratwurzel. Verwende dazu visuelle Hilfsmittel. Warum konvergiert der Algorithmus gegen den exakten Wert der Quadratwurzel?
10. Erläutere die Monte-Carlo-Methode zur Schätzung von π und erkläre, wie man mithilfe von Zufallszahlen eine Annäherung an π erreichen kann.
11. Erkläre den Unterschied zwischen einem stochastischen und einem deterministischen Algorithmus anhand eines selbst gewählten Beispiels.
12. Finde weitere Probleme, die sich durch Monte-Carlo-Simulationen lösen lassen. Weshalb sind manche dieser Probleme mit anderen Methoden nicht lösbar?
13. Du hast zwei Sanduhren: Eine läuft 4 Minuten, die andere 7 Minuten. Wie kannst du damit genau 9 Minuten messen? Entwickle einen präzisen Algorithmus für das Problem!

14. Du hast drei Gefäße mit 3, 5 und 8 Litern Fassungsvermögen. Das 8-Liter-Gefäß ist voll, die anderen sind leer. Entwickle einen Algorithmus, um genau 4 Liter abzumessen!
15. Entwickle einen Algorithmus, der prüft ob eine eingegebene Zahl eine Primzahl ist. Stelle den Algorithmus als Flussdiagramm dar!
16. Entwirf einen Algorithmus für den einfachen Getränkeautomaten aus dem vorigen Kapitel. Schreibe den Algorithmus als Pseudocode und als Flussdiagramm auf!
17. Recherchiere einen Fall, bei dem ein algorithmisches System zu problematischen Entscheidungen geführt hat. Analysiere die Ursachen und schlage Möglichkeiten zur Verbesserung!
18. Diskutiert die ethischen Implikationen von nicht-erklärbaren algorithmischen Entscheidungen in kritischen Bereichen wie Medizin oder Rechtsprechung! In welchen Bereichen könnte eine fehlende Erklärbarkeit noch kritisch sein?

Teil II.

Repräsentation

3. Informationen



Zu diesem Kapitel gibt es ein Video auf YouTube. Scanne einfach den QR-Code oder klicke in der digitalen Version des Buches auf den QR-Code.

Im digitalen Zeitalter sind wir von **Information** umgeben - ein ständiger Strom von Nachrichten, Daten und Fakten prägt unseren Alltag. Doch was versteht die Informatik unter dem Begriff "Information"? Eine erste Annäherung bietet unsere alltägliche Erfahrung: Information entsteht, wenn wir etwas Neues erfahren, das uns hilft, bessere Entscheidungen zu treffen. Nehmen wir eine Wettervorhersage als Beispiel: Die Aussage "es wird morgen regnen" stellt für jemanden, der das Wetter noch nicht kennt, eine wichtige Information dar - sie reduziert die Unsicherheit über das morgige Wetter. Die Informatik und Informationstheorie greifen genau dieses Konzept auf und definieren **Information** präzise als die Abnahme von Unsicherheit.

Dieses Kapitel führt systematisch in den Informationsbegriff ein und legt damit das theoretische Fundament für die nachfolgenden Themen: vom Binärsystem als Grundlage der digitalen Informationsverarbeitung über verschiedene Methoden der Informationskodierung bis hin zu praktischen Anwendungen wie Datenkompression und Verschlüsselung.

3.1. Wie definieren wir Information?

Information begegnet uns im Alltag in vielen Formen – als Nachricht, Erkenntnis oder Mitteilung. Die Informatik verwendet jedoch einen präziseren Informationsbegriff, der sich von diesem alltäglichen Verständnis unterscheidet. Basierend auf Adami (2016) werden wir in diesem Kapitel eine formale Definition von Information entwickeln und anhand konkreter Beispiele veranschaulichen.

“Information is that which allows you to make a correct prediction with accuracy better than chance.”

Adami, Christoph. "What Is Information?" *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2063, Mar. 2016, p. 20150230, <https://doi.org/10.1098/rsta.2015.0230>.

Abbildung 3.1.: Definition des Informationsbegriffs laut Adami (2016).

Um das Konzept der Information greifbar zu machen, betrachten wir ein praktisches Gedankenexperiment: ein Zahlenratespiel. Angenommen, ich denke an eine Zahl zwischen 1 und 16 (Abbildung 3.2). Deine Aufgabe ist es, diese Zahl zu erraten. Du hast zwar nur einen finalen Rateversuch, darfst aber zuvor Fragen stellen, die ich mit “ja” oder “nein” beantworte. Diese Fragen haben die Form “Ist die Zahl größer als X?” und helfen dir, die möglichen Zahlen systematisch einzugrenzen.

Bei jeder Antwort reduziert sich deine **Unsicherheit** über die gesuchte Zahl. Durch den Ausschluss bestimmter Zahlen wird der Bereich möglicher Lösungen kleiner, und dein Wissen über die gesuchte Zahl wächst entsprechend.

Wichtig: Der informationstheoretische Begriff der Information unterscheidet sich vom alltäglichen Verständnis. In der Informatik geht es nicht primär um die *inhaltliche Bedeutung* (Semantik) einer Nachricht, sondern um die *quantitative Reduktion von Unsicherheit*. Die Frage ist also nicht, was eine Information inhaltlich - oder semantisch - bedeutet, sondern wie stark sie die vorhandene Ungewissheit verringert.

Dieses Prinzip der Unsicherheitsreduktion bildet das Fundament der Informationstheorie. Mit jeder Antwort, die wir erhalten, sammeln wir Information, die die Unsicherheit reduziert. Aber wie lässt sich dieser Informationsgewinn konkret messen? Welche Menge an Information steckt in einer einzelnen Antwort?



Abbildung 3.2.: Das Zahlenratespiel zur Veranschaulichung des Informationsbegriffs.

3.2. Wie messen wir Information?

3.2.1. Bit für Bit

Die Informatik definiert Information präzise als das, was es uns ermöglicht, richtige Vorhersagen zu treffen, die genauer sind als der pure Zufall (Adami 2016). Dabei konzentriert sich diese Definition auf den messbaren Aspekt: Information reduziert systematisch die Unsicherheit über einen Sachverhalt, indem sie die Anzahl möglicher Optionen eingrenzt.

Betrachten wir nun unser Zahlenratespiel unter dem Aspekt der Informationstheorie. In diesem Beispiel reduziert jede Antwort systematisch die Anzahl der möglichen Zahlen. Diese Reduzierung der Unsicherheit lässt sich mit einer präzisen Maßeinheit quantifizieren: dem **Bit**. Ein Bit (von **binary digit**, Binärziffer) stellt die fundamentale Einheit der Information dar und entspricht einer Halbierung der Unsicherheit. Anders ausgedrückt: Wenn eine Antwort die Anzahl der möglichen Optionen exakt halbiert, gewinnen wir genau ein Bit an Information.

Allerdings werden nicht alle Fragen und deren Antworten genau ein Bit Information liefern. Wenn zum Beispiel deine erste Frage im Ratespiel lautet: “Ist deine Zahl größer als 12?” und die Antwort “nein” ist, bleiben die Zahlen 1 bis 12 übrig. Das bedeutet, du hast noch 12 Optionen von ursprünglich 16, was die Unsicherheit nicht halbiert. Es werden nur 4 statt der nötigen 8 Möglichkeiten entfernt. Lautet die Antwort dagegen “ja”, so kannst du insgesamt

12 Möglichkeiten streichen, was mehr als der Hälfte entspricht und der Informationsgehalt der Antwort wäre größer als ein Bit.

Um genau ein Bit Information zu erhalten, solltest du versuchen, mit jeder Frage genau die Hälfte der möglichen Zahlen auszuschließen. Wenn du zum Beispiel fragst “Ist deine Zahl größer als 8?”, stellst du sicher, dass du - egal ob die Antwort “ja” oder “nein” ist - in beiden Fällen 8 mögliche Zahlen übrig behältst. Ist die Antwort “ja”, bleiben die Zahlen 9 bis 16 übrig. Ist die Antwort “nein”, bleiben die Zahlen 1 bis 8. In beiden Szenarien wird deine Unsicherheit um die Hälfte reduziert, also um ein Bit.

Indem du deine Fragen sorgfältig so wählst, dass sie die verbleibenden Optionen jedes Mal halbieren, reduzierst du deine Unsicherheit Bit für Bit und du findest schneller die richtige Zahl.



Abbildung 3.3.: Deine Fragen sollten die Anzahl der Möglichkeiten halbieren.

Angenommen, die Antwort auf deine erste Frage “Ist deine Zahl größer als 8?” war “nein”. Was sollte deine nächste Frage sein, um die Unsicherheit weiterhin effektiv zu reduzieren? Die beste Strategie ist es zu fragen: “Ist sie größer als 4?”. Diese Vorgehensweise stellt sicher, dass dir immer nur vier mögliche Zahlen bleiben: entweder 1 bis 4, wenn die Antwort “nein” ist, oder 5 bis 8, wenn die Antwort “ja” ist. Erneut halbiert dies die verbleibenden Optionen und liefert genau ein Bit an Information.



Abbildung 3.4.: Nach zwei Fragen bleiben noch 4 Möglichkeiten, wenn die Fragen gut gewählt wurden.

Lass uns mit dieser Methode fortfahren. Angenommen, deine zweite Frage “Ist sie größer als 4?” erhält ein “nein” als Antwort. Deine neue Auswahl an Zahlen ist auf 1, 2, 3 und 4 begrenzt. Um die Unsicherheit weiter zu reduzieren, wäre die nächste logische Frage: “Ist sie größer als 2?” Dies lässt dir entweder die Zahlen 1 und 2 oder 3 und 4, abhängig von der Antwort.

Indem du weiterhin Fragen stellst, die systematisch die verbleibenden Optionen halbieren, kannst du sehen, wie wir schrittweise die Unsicherheit Bit für Bit reduzieren. Schließlich wirst du nach nur vier Fragen die Zahl auf genau eine eingrenzen, da keine anderen Möglichkeiten mehr übrig sind. Das bedeutet, du hast die Unsicherheit auf null reduziert.

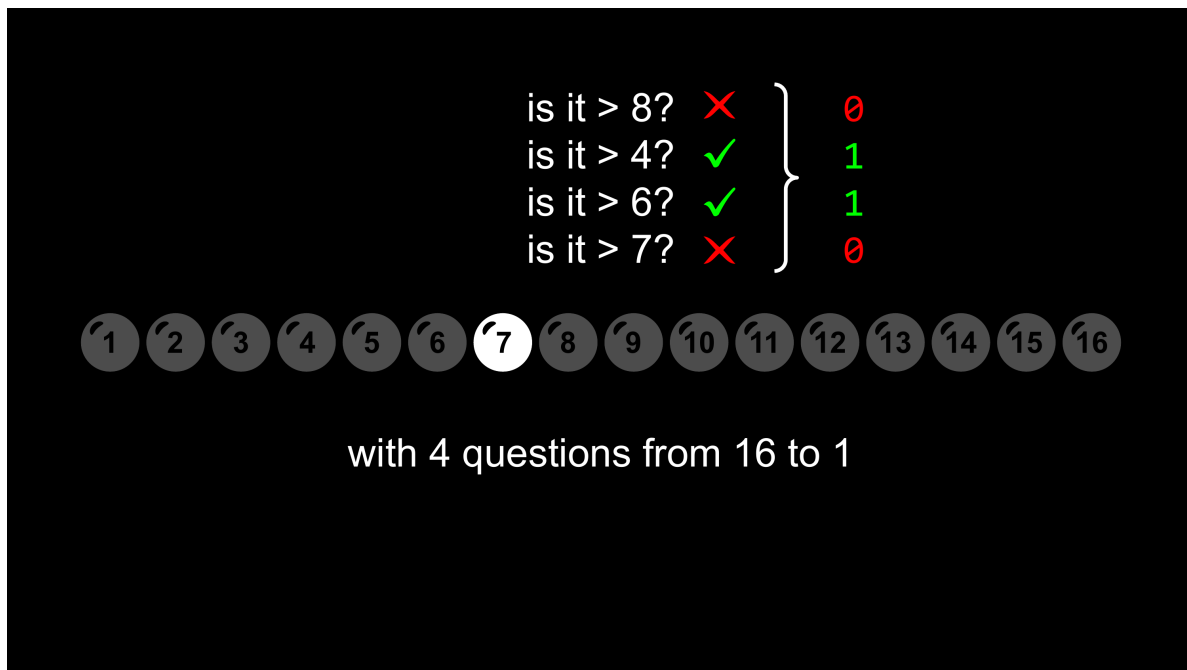


Abbildung 3.5.: Wir benötigen vier Ja/Nein-Fragen, um die Optionen von 16 auf eine einzige zu reduzieren.

3.2.2. Unsicherheit

Wir haben oben gelernt, dass wir mit jeder Ja/Nein-Frage, die die Hälfte der Möglichkeiten eliminiert, ein Bit an Information gewinnen. Aber wie können wir dieses Konzept der Reduktion von Unsicherheit mathematisch quantifizieren? Wir gehen das Schritt für Schritt durch.

Stell dir vor, du bist am Anfang unseres Zahlenratespiels. Es gibt 16 mögliche Zahlen, an die ich denken könnte, daher ist deine Wahrscheinlichkeit, beim ersten Versuch die richtige Zahl zu erraten, ziemlich gering:

$$P_{correct} = \frac{1}{16}$$

Das entspricht einer Wahrscheinlichkeit von nur 0,0625 - definitiv nicht zu deinen Gunsten. Angenommen, du stellst eine Frage, die die Möglichkeiten auf die Hälfte reduziert. Jetzt verbessert sich deine Chance, richtig zu raten:

$$P_{correct} = \frac{1}{8}$$

Mit jeder weiteren Frage verdoppelt sich diese Wahrscheinlichkeit, während sich deine Unsicherheit halbiert. Ist Wahrscheinlichkeit die beste Methode, um Unsicherheit zu messen? Wenn wir möchten, dass unser Maß für Unsicherheit abnimmt, während wir mehr Informationen sammeln, ist es sinnvoll, den Kehrwert der Wahrscheinlichkeit zu verwenden. Das entspräche der Anzahl an Möglichkeiten, die noch im Rennen sind.

Zu Beginn gibt es 16 Möglichkeiten, also begänne die Unsicherheit bei 16. Nach der ersten Frage fiel sie auf 8, dann auf 4, 2 und schließlich 1. Allerdings würde dieses Maß uns selbst dann, wenn nur noch eine Option übrig ist, eine gewisse Unsicherheit, nämlich 1, anzeigen, was nicht ganz unserem intuitiven Verständnis entspricht. Wenn wir sicher sind, dann sollte die Unsicherheit 0 sein.

Claude Shannon, der Vater der Informationstheorie, schlug einen anderen Ansatz vor. Er empfahl, die Unsicherheit mithilfe des Logarithmus zur Basis 2 der Anzahl der Möglichkeiten zu messen:

$$H = \log_2(N)$$

Diese Methode hat mehrere Vorteile. Erstens ist die Unsicherheit gleich null, wenn es nur eine mögliche Antwort gibt ($N=1$), was mit unserer Intuition übereinstimmt:

$$\log_2(1) = 0$$

Ein weiterer Vorteil ist die Einfachheit der Berechnungen, besonders wenn es um mehrere unabhängige Unsicherheiten geht. Nehmen wir an, das Spiel ändert sich: Jetzt denke ich an zwei unabhängige Zahlen zwischen 1 und 16. Bevor du irgendwelche Fragen stellst, beträgt deine Unsicherheit für jede Zahl:

$$\log_2(16) = 4 \text{ bits}$$

Die Gesamtunsicherheit für beide Zahlen ist einfach die Summe ihrer individuellen Unsicherheiten:

$$\log_2(16) + \log_2(16) = 8 \text{ bits}$$

Wenn wir stattdessen Wahrscheinlichkeiten verwenden, müssten wir die Chancen multiplizieren, um die Wahrscheinlichkeit zu finden, beide Zahlen korrekt zu erraten:

$$P_{\text{correct,correct}} = \frac{1}{16} \times \frac{1}{16} = \frac{1}{256}$$

Mit beiden Zahlen gibt es 256 Möglichkeiten. Mit Shannons Methode erhalten wir das gleiche Ergebnis:

$$\log_2(256) = 8 \text{ bits}$$

Wie du siehst, vereinfacht die Verwendung von Logarithmen unsere Berechnungen, besonders wenn es um mehrere Quellen der Unsicherheit geht. Diese Klarheit und Einfachheit sind der Grund, warum Shannons Ansatz fundamental für die Informationstheorie geworden ist.

3.2.3. Information

Um den Begriff *Information* aus dem Begriff der *Unsicherheit* herzuleiten, betrachten wir zwei Zustände: Die ursprüngliche Unsicherheit vor einer Frage, die wir als H_0 bezeichnen, und die reduzierte Unsicherheit nach der Antwort, die wir H_1 nennen. Die Menge an Information I , die wir durch die Antwort gewinnen, entspricht der Differenz dieser beiden Unsicherheiten:

$$I = H_0 - H_1$$

Information ist also die Reduzierung von Unsicherheit. Mit der obigen Formel können wir die Information I präzise quantifizieren. Dies ermöglicht uns zu berechnen, wie viel Information wir durch jede Frage gewinnen. Betrachten wir ein konkretes Beispiel aus unserem Zahlenratespiel: Angenommen, deine erste Frage ist “Ist deine Zahl größer als 12?” und die Antwort lautet “nein”. Dann bleiben die Zahlen 1 bis 12 als Möglichkeiten übrig. Die gewonnene Information berechnet sich wie folgt:

$$I = \log_2(16) - \log_2(12) \approx 4 - 3.585 \approx 0.415 \text{ bits}$$

In diesem Fall lieferte die Antwort etwa 0,415 Bits an Information – also weniger als ein Bit. Das ist folgerichtig, da die Antwort weniger als die Hälfte der Möglichkeiten eliminiert hat.

Wie sähe es aus, wenn die Antwort “ja” gewesen wäre? In diesem Fall blieben nur die Zahlen 13, 14, 15 und 16 übrig, also vier mögliche Optionen:

$$I = \log_2(16) - \log_2(4) = 4 - 2 = 2 \text{ bits}$$

Diese Antwort lieferte mehr als ein Bit, nämlich 2 Bits. Wie lässt sich dieser Unterschied erklären?

3.2.4. Wahrscheinliche und unwahrscheinliche Antworten

Die Menge an Information hängt von der Wahrscheinlichkeit der jeweiligen Antwort ab. Bei einer Frage, die die Möglichkeiten halbiert, hat jede Antwort “Ja” oder “Nein” auf die Frage “Ist die Zahl größer als X?” eine Wahrscheinlichkeit von genau 50%. Diese gleichmäßige Verteilung vereinfacht die Berechnung und liefert genau ein Bit an Information.

Bei der Frage “Ist deine Zahl größer als 12?” sind die beiden Antworten “Ja” und “Nein” allerdings nicht gleich wahrscheinlich. Es gibt zwölf mögliche Zahlen, die ein “Nein” ergeben und nur vier, die ein “Ja” ergeben, was Wahrscheinlichkeiten von 75% für “Nein” und 25% für “Ja” bedeutet. Da die “Ja”-Antwort weniger wahrscheinlich ist, ist sie überraschender – und liefert deshalb mehr Information. Wie wir berechnet haben, gibt uns ein “Ja” 2 Bits an Information, während ein “Nein” nur 0,415 Bits liefert.

Dieser Zusammenhang ist ein Kernprinzip der Informationstheorie: Je unwahrscheinlicher ein Ereignis ist, desto mehr Information enthält sein Auftreten. Umgekehrt liefert eine sehr wahrscheinliche Antwort, wie das “Nein” im obigen Beispiel, weniger Information, da sie weniger überraschend ist.

Warum zielen wir darauf ab, Fragen zu stellen, die die Möglichkeiten gleichmäßig halbieren? Man könnte auch wild raten in der Hoffnung, durch eine unwahrscheinlichere Antwort mehr Informationen zu bekommen – auch wenn man seltener richtig liegt. Die Antwort ist einfach: Eine Frage, die den Raum der Möglichkeiten halbiert, maximiert unseren *erwarteten* Informationsgewinn. Zwar können wir mit einer riskanten Frage wie “Ist die Zahl größer 15?” unser Glück herausfordern und im Erfolgsfall die Unsicherheit stark verringern. Dies ist jedoch keine nachhaltige Strategie. Wenn wir das Spiel nicht nur einmal, sondern 10, 100 oder 1000 Mal spielen, nähern wir uns im Mittel dem Erwartungswert für die gewonnene Information $E[I]$. Und dieser Erwartungswert favorisiert eindeutig Fragen, die die Hälfte der Möglichkeiten eliminieren.

Um dies zu verdeutlichen, berechnen wir zunächst den Erwartungswert für die extreme Frage “Ist die Zahl größer 15?”:

$$E[I] = \left(\frac{1}{16} \times 4 \right) + \left(\frac{15}{16} \times 0.09311 \right) = 0.3373$$

Der Erwartungswert berechnet sich durch den mit der Wahrscheinlichkeit gewichteten Informationsgewinn für jede mögliche Antwort. Bei der Frage “Ist die Zahl größer als 15?” gibt es ein “Ja” nur dann, wenn die Zahl 16 ist. Von den sechzehn möglichen Zahlen erzeugt also nur eine diese Antwort. Die Wahrscheinlichkeit beträgt somit $\frac{1}{16}$. In diesem Fall reduziert sich unsere Unsicherheit sofort auf Null, da nur noch eine einzige Möglichkeit übrig bleibt. Die ursprüngliche Unsicherheit betrug 4 Bits:

$$H_0 = \log_2(16) = 4 \text{ bits}$$

Und wenn die Unsicherheit nach der Antwort “Ja” auf Null reduziert wird:

$$H_1 = \log_2(1) = 0 \text{ bits}$$

Daraus ergibt sich ein Informationsgehalt von 4 Bits:

$$I = H_0 - H_1 = 4 - 0 = 4 \text{ bits}$$

Dies erklärt den ersten Teil der Erwartungswertgleichung. Der zweite Teil folgt demselben Prinzip, allerdings mit einem wichtigen Unterschied: Die Wahrscheinlichkeit für die Antwort “Nein” ist mit $\frac{15}{16}$ deutlich höher. Gleichzeitig verbleibt eine große Restunsicherheit H_1 , da wir lediglich eine einzige Zahl ausschließen konnten:

$$H_1 = \log_2(15) \approx 3.9069 \text{ bits}$$

Die gewonnene Information ist somit sehr klein:

$$I = 4 - 3.9069 \approx 0.0931 \text{ bits}$$

Anhand der Erwartungswertformel konnten wir zeigen, dass eine extreme Frage, die mit viel Glück direkt zum Ziel führt, in unserem Ratespiel einen erwarteten Informationsgehalt von etwa einem Drittel Bit (0,3373) hat. Prüfen wir nun, ob der Erwartungswert für Fragen, die die Hälfte der Möglichkeiten eliminieren, tatsächlich höher ist:

$$E[I] = (0.5 \times 1) + (0.5 \times 1) = 0.5 + 0.5 = 1 \text{ bit}$$

Wie erwartet beträgt der Erwartungswert genau 1 Bit, da bei jeder Antwort – ob “Ja” oder “Nein” – jeweils die Hälfte der Optionen eliminiert wird.

3.2.5. Mehr als zwei Antworten

Bisher haben wir in unserem Zahlenratespiel nur Fragen mit zwei Antwortmöglichkeiten betrachtet. Informationen, also Antworten auf Fragen, können jedoch vielfältiger sein. Wie berechnen wir die Information, wenn es mehr als zwei Antwortmöglichkeiten gibt?

Als Beispiel betrachten wir eine Urne mit farbigen Kugeln in drei Farben: Rot, Grün und Blau. Aus dieser Urne ziehen wir zwei Kugeln, wobei wir nach jedem Zug die gezogene Kugel zurück in die Urne legen. Dieses statistische Experiment bezeichnet man als “Ziehen mit Zurücklegen”.

3.3. Wovon hängt der Informationsgehalt einer Nachricht ab?

Nach den obigen Ausführungen können wir die Frage nun präzise beantworten: Der Informationsgehalt einer Nachricht hängt hauptsächlich von zwei Faktoren ab: **(1)** der *Anzahl der möglichen Alternativen* und **(2)** deren *Wahrscheinlichkeiten*.

3.3.1. Anzahl der Alternativen

Je mehr verschiedene Möglichkeiten es gibt, was eine Nachricht aussagen kann, desto mehr Information kann im besten Fall vermittelt werden. Wenn zum Beispiel ein Lehrbuch-Test eine von zwei Antworten richtig haben kann (etwa wahr/falsch), dann liefert die Korrektur „richtig“ oder „falsch“ maximal 1 Bit Information – es gibt ja nur zwei mögliche Ausgänge. Hat eine Prüfungsfrage aber vier mögliche Antworten (A, B, C, D) von denen genau eine richtig ist, dann liefert die Angabe der korrekten Antwort bis zu $\log_2(4) = 2$ Bit Information. Allgemein: Ist eine Nachricht eine Auswahl aus N gleich möglichen Alternativen, beträgt ihr Informationsgehalt $\log_2 N$ Bit. Mehr Alternativen bedeuten einen höheren potentiellen Informationsgehalt.

3.3.2. Wahrscheinlichkeiten der Alternativen

Oft sind manche Nachrichten oder Ereignisse wahrscheinlicher als andere. Wie wir gesehen haben, tragen unwahrscheinliche (überraschende) Ereignisse mehr Information bei als sehr wahrscheinliche. Das klassische Beispiel ist das Wetter: In einer Region, in der es normalerweise jeden Tag sonnig ist, enthält die Nachricht „Heute scheint die Sonne“ kaum neue Information – es war ja ohnehin zu erwarten. Dagegen wäre in dieser Region die Nachricht „Heute regnet es“ deutlich informativer (weil Regen dort selten ist). In einer Regenwald-Region wäre es umgekehrt. **Der Informationsgehalt ist also höher, je unwahrscheinlicher (überraschender) der Inhalt der Nachricht ist.** Formal spiegelt das die Formel $I(p) = -\log_2 p$ wider: Kleine Wahrscheinlichkeit p (seltenes Ereignis) ergibt großes $I(p)$.

3.3.3. Länge der Nachricht (Anzahl Zeichen)

Ein weiterer offensichtlicher Faktor ist die Menge an Zeichen oder Symbolen, aus denen sich die Nachricht zusammensetzt. Eine längere Nachricht kann natürlich mehr Information tragen als eine sehr kurze, da sie mehr Inhalte übermitteln kann. Allerdings hängt das nicht nur von der Länge ab, sondern auch davon, ob die zusätzlichen Zeichen tatsächlich *neue* Information liefern oder nur bereits Bekanntes wiederholen. Mehr dazu gleich beim Thema Redundanz. Aber grundsätzlich können wir sagen: Der Informationsgehalt einer längeren Nachricht, die aus mehreren Symbolen besteht, ist – bei unabhängigen Symbolen – die Summe der Informationsgehalte der einzelnen Symbole

Beispiel: Besteht eine Nachricht aus 3 Buchstaben, und jeder Buchstabe trägt im Durchschnitt 4 Bit Information, dann hat die ganze Nachricht etwa $3 \cdot 4 = 12$ Bit Informationsgehalt (unter der Annahme, dass die Buchstaben unabhängig und nicht vorhersagbar voneinander sind).

Zusammengefasst: **Der Informationsgehalt einer Nachricht hängt von der Unsicherheit ab, die sie ausräumt.** Diese Unsicherheit wiederum ergibt sich aus der Zahl der möglichen Nachrichten und deren Verteilung. Maximal informativ ist eine Nachricht, die aus vielen möglichen, gleich wahrscheinlichen Alternativen ausgewählt wurde – dann ist die Überraschung am größten. Wenn einige Ergebnisse schon vorher sehr wahrscheinlich sind, verringert sich der Informationsgehalt entsprechend.

Fachbegriffe: Informationsgehalt bezeichnet die Menge an Information, die eine konkrete Nachricht liefert. Sie ist umso höher, je mehr mögliche Alternativen es gab und je unwahrscheinlicher die empfangene Nachricht war. Bei zusammengesetzten Nachrichten addieren sich die Informationsgehalte der Teile (wenn sie unabhängig voneinander sind).

3.4. Was ist Redundanz und welche Rolle spielt sie?

3.5. Welcher Zusammenhang besteht zwischen Information und Daten

Die kurze Antwort lautet: Daten repräsentieren Informationen. Daten sind die physische oder digitale Repräsentation von Informationen in einer bestimmten Form oder Struktur. Wenn wir zum Beispiel die Zahl “42” auf Papier schreiben oder in einem Computer speichern, dann sind die geschriebenen Ziffern oder die Bits im Speicher die Daten, während die tatsächliche Bedeutung – also die Zahl zweiundvierzig – die Information ist. Vielleicht eine Antwort auf eine Frage wie die nach dem Sinn des Lebens? Ein Computer verarbeitet dabei ausschließlich Daten in Form von Bits, die dann durch entsprechende Programme wieder in für Menschen verständliche Informationen umgewandelt werden. Dazu verwenden wir dann die Codesysteme, die wir in Kapitel 5 kennenlernen.

Übungsaufgaben

1. Ihr wollt euch für komplettes Skatspiel die Abfolge schwarzer (Pik, Kreuz) und roter (Herz, Karo) Karten merken und als Hexadezimalzahl kodieren. Wie viele Ziffern müsst ihr euch merken?
2. Du schreibst eine Kurznachricht und kodierst sie mit dem ASCII-Codesystem. Deine Nachricht hat inklusive Leerzeichen 40 Zeichen. Wie viele Bytes benötigst du für Kodierung deiner Nachricht?
3. Du möchtest ein Foto speichern, dessen Pixel als RGB-Werte repräsentiert werden. Das Foto soll 200 x 300 Pixel haben. Welchen Speicherbedarf haben die Farbwerte aller Pixel zusammen?
4. Du möchtest ein Foto im RGB-Format speichern und hast 0,75 MB zur Verfügung. Das Foto ist quadratisch. Welche Dimension hat das größtmögliche Bild, dass in den vorhandenen Speicher passt?
5. Du möchtest ein Bild in insgesamt 256 Graustufen speichern. Das Bild hat eine Auflösung von 1200 x 800 Pixel. Wie viel Speicherplatz benötigst du?
6. Du möchtest ein Bild im Schwarz/Weiß-Format speichern. Du hast 0,5 KB zur Verfügung. Wie viele Pixel kannst du speichern?
7. Ermittle die maximale Farbtiefe, die du für ein Bild mit 1000 x 800 Pixeln und einer Dateigröße von 0,8 MB verwenden kannst.
8. Wie viele Farben könnten wir kodieren, wenn wir nur 7 Bits statt 8 für jede der Grundfarben Rot, Grün und Blau verwenden würden?
9. Wäre es sinnvoll, eine erweiterte Version des RGB-Codes zu erstellen, die für jede der Grundfarben ein zusätzliches Nibble hinzufügt? Wie viele Farben könnten wir dann kodieren? Finde Argumente für und gegen diese Idee!
10. Wir hatten nicht immer die vollen 16.777.216 Farben auf unseren Computerbildschirmen. Recherchiere, wie sich Farbcodes im Laufe der Geschichte der digitalen Computer entwickelt haben. Kannst du einige Screenshots aus längst vergangenen Zeiten finden?
11. Du möchtest eine Audiodatei mit einer Länge von 1 Minute speichern. Du wählst eine Sample Rate von 4000 Hz und eine Bit Depth von 16 Bit. Wie groß wird die Datei?
12. Du hast eine Audiodatei (unkomprimiert) mit einer Länge von 10 Sekunden und einer Größe von 100 KB. Du weißt, dass die Sample Rate 5 KHz beträgt. Welche Bit Depth hat die Aufnahme?
13. Du entwirfst ein Logo für deine Firma, das du auf möglichst unterschiedlichen Medien, vom Flyer bis zum Plakat, in hoher Qualität drucken möchtest. Welches Dateiformat wählst du und warum?

14. Erläutere den Unterschied zwischen Codesystemen mit fester und variabler Länge. Nenne jeweils ein Beispiel.
15. Ist der Morsecode ein binäres Codesystem? Begründe deine Antwort.
16. Wie viele Bits verwendet der Braille-Code, um ein Symbol darzustellen? Ist der Braille-Code ein Binärcode?

4. Bits



Zu diesem Kapitel gibt es ein Video auf YouTube. Scanne einfach den QR-Code oder klicke in der digitalen Version des Buches auf den QR-Code.

4.1. Was ist ein Zahlensystem?

4.1.1. Stellenwertsysteme

Bevor wir uns speziellen Zahlensystemen zuwenden, müssen wir verstehen, was ein **Stellenwertsystem**** ist. In einem Stellenwert- oder Positionssystem hängt der Wert einer Ziffer vom Platz ab, an dem sie steht. Das bedeutet: Eine Zahl besteht aus einer Reihe von Ziffern, und jede Ziffer hat je nach ihrer Position einen unterschiedlichen Wert (Stellenwert). Unser gewohntes Zehnersystem (Dezimalsystem) **ist genau so ein Stellenwertsystem.

Stellenwertsysteme folgen dem *Bündelungsprinzip*: Man bündelt eine gewisse Anzahl an Einheiten und schiebt dann eine Stelle weiter. Ein einfaches Beispiel: Stell dir vor, du zählst Punkte und machst immer einen Strich, sobald du 10 Punkte gezählt hast. Zehn einzelne Punkte werden also zu einem Bündel (einem Strich) zusammengefasst. Dieses Bündel entspricht einer neuen Stelle in unserer Zahlenschreibweise. So wird z. B. die Anzahl “13” im Dezimalsystem interpretiert als *1 Bündel zu 10 und 3 einzelne*. Allgemein gesagt: Im Dezimalsystem hat jede Stelle einen Wert, der eine Potenz von 10 ist, also etwa $10^0 = 1$, $10^1 = 10$, $10^2 = 100$ und so weiter. Die Ziffern an diesen Stellen geben an, wie viele Bündel dieses Werts vorhanden sind.

Das Besondere daran ist, dass man mit begrenzt vielen Ziffern unendlich viele Zahlen darstellen kann. Jede ganze Zahl ab 0 kann durch eine geeignete Kombination der Ziffern dargestellt werden, solange man ausreichend Stellen zur Verfügung hat. Stellenwertsysteme machen das

Rechnen und Darstellen von Zahlen äußerst effizient, weil jede Stelle für sich einen bestimmten Wert *beiträgt* und wir diese Beiträge einfach addieren können, um den Gesamtwert zu erhalten

4.1.2. Das Dezimalsystem

Das **Dezimalsystem** ist das Zahlensystem, das wir im Alltag fast überall verwenden. Es hat die Basis 10, was bedeutet, dass wir zehn verschiedene Ziffern haben: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Jede Stelle in einer Dezimalzahl repräsentiert eine Potenz von 10. Zum Beispiel bedeutet die Zahl **203** im Dezimalsystem folgendes:

- Die “3” steht an der **1er-Stelle** (10^0), sie repräsentiert also 3×1 .
- Die “0” steht an der **10er-Stelle** (10^1), sie repräsentiert 0×10 .
- Die “2” steht an der **100er-Stelle** (10^2), sie repräsentiert 2×100 .

Wenn wir das zusammenrechnen:

$$2 \times 100 + 0 \times 10 + 3 \times 1 = 200 + 0 + 3 = 203$$

So einfach ist das Prinzip – vermutlich kennst du es schon längst, ohne bisher über den Begriff *Stellenwertsystem* nachgedacht zu haben.

Warum benutzen wir ausgerechnet zehn verschiedene Ziffern? Historisch liegt das vermutlich daran, dass der Mensch zehn Finger hat und es nahelag, damit zu zählen. Tatsächlich wurde das Dezimalsystem in Indien entwickelt und durch die Einführung der Ziffer “0” (Null) im 7. Jahrhundert n. Chr. besonders leistungsfähig. Über die arabische Welt gelangte es nach Europa und setzte sich schließlich durch. Heute verwenden wir das Dezimalsystem so selbstverständlich, dass es wichtig ist, uns bewusst zu machen: Es ist nur eines von vielen möglichen Zahlensystemen.

4.1.3. Zahlensysteme mit einer anderen Basis

Theoretisch kann ein Stellenwertsystem **jede ganze Zahl ab 2 als Basis** haben. Die Basis bestimmt, wie viele unterschiedliche Ziffern es gibt und um welchen Faktor der Stellenwert zunimmt, wenn man eine Stelle nach links rückt. Im Dezimalsystem (Basis 10) gibt es 10 Ziffern und jede Stelle ist 10 mal mehr wert als die Stelle rechts daneben. In einem System Basis 5 gäbe es nur die Ziffern 0–4, und jede Stelle wäre 5 mal so viel wert wie die vorige, usw.

4.2. Ist das Bit die kleinste Informationseinheit?

4.3. Was ist ein Byte?

Übungsaufgaben

1. Konvertiert die folgenden Dezimalzahlen in das Binärsystem!
 - a. 12
 - b. 27
 - c. 85
 - d. 128
 - e. 255
2. Konvertiert die folgenden Binärzahlen in das Dezimalsystem!
 - a. 1001
 - b. 1110
 - c. 101011
 - d. 1101010
 - e. 10000001
3. Konvertiert die folgenden Oktalzahlen in das Dezimalsystem!
 - a. 7
 - b. 10
 - c. 20
 - d. 77
 - e. 100
4. Recherchiert einen einfachen Algorithmus zur Konvertierung einer Dezimalzahl in eine Binärzahl. Beschreibt den Algorithmus mit euren eigenen Worten und erläutert ihn an einem konkreten Beispiel.
5. Im Hexadezimalsystem zur Basis 16 reichen die Ziffern 0-9 nicht aus, um alle möglichen Werte darzustellen. Wie wird dieses Problem gelöst?
6. Erläutere in eigenen Worten, warum die meisten digitalen Computer das Binärsystem verwenden.
7. Erläutere die Vor- und Nachteile bei der Verwendung des Binärsystems als Grundlage für digitale Computer! Gäbe es grundsätzlich Alternativen?

5. Codesysteme

5.1. ASCII-Code

5.2. Verallgemeinerung von Codesystemen

5.3. Unicode

5.4. RGB-Code

5.5. Code vs Codec

Der Begriff *Codec* leitet sich von den zwei Wörtern **enc**oder und **dec**oder ab. Die beiden Begriffe werden zwar ähnlich geschrieben, unterscheiden sich aber in ihrer Bedeutung. Ein Codec beschreibt ein Vorgehen oder einen Algorithmus, um Daten vor dem Senden nach einem definierten Verfahren zu kodieren und nach dem Empfang wieder zu dekodieren. Dies ist besonders sinnvoll, um Informationen effizienter zu übertragen und weniger Bandbreite zu beanspruchen. Zudem können Daten verschlüsselt und die Kommunikation dadurch sicherer gemacht werden. Wir lernen einige Methoden für beide Anwendungsfälle in einem späteren Kapitel kennen. Im Gegensatz dazu beschreibt ein Code ein System, das einer Reihe von Symbolen eine bestimmte Bedeutung zuordnet, wie wir in diesem Abschnitt gesehen haben.

5.6. Übungsaufgaben

1. Ihr wollt euch für komplettes Skatspiel die Abfolge schwarzer (Pik, Kreuz) und roter (Herz, Karo) Karten merken und als Hexadezimalzahl kodieren. Wie viele Ziffern müsst ihr euch merken?
2. Du schreibst eine Kurznachricht und kodierst sie mit dem ASCII-Codesystem. Deine Nachricht hat inklusive Leerzeichen 40 Zeichen. Wie viele Bytes benötigst du für Kodierung deiner Nachricht?
3. Du möchtest ein Foto speichern, dessen Pixel als RGB-Werte repräsentiert werden. Das Foto soll 200 x 300 Pixel haben. Welchen Speicherbedarf haben die Farbwerte aller Pixel zusammen?
4. Du möchtest ein Foto im RGB-Format speichern und hast 0,75 MB zur Verfügung. Das Foto ist quadratisch. Welche Dimension hat das größtmögliche Bild, dass in den vorhandenen Speicher passt?
5. Du möchtest ein Bild in insgesamt 256 Graustufen speichern. Das Bild hat eine Auflösung von 1200 x 800 Pixel. Wie viel Speicherplatz benötigst du?
6. Du möchtest ein Bild im Schwarz/Weiß-Format speichern. Du hast 0,5 KB zur Verfügung. Wie viele Pixel kannst du speichern?
7. Ermittle die maximale Farbtiefe, die du für ein Bild mit 1000 x 800 Pixeln und einer Dateigröße von 0,8 MB verwenden kannst.
8. Wie viele Farben könnten wir kodieren, wenn wir nur 7 Bits statt 8 für jede der Grundfarben Rot, Grün und Blau verwenden würden?
9. Wäre es sinnvoll, eine erweiterte Version des RGB-Codes zu erstellen, die für jede der Grundfarben ein zusätzliches Nibble hinzufügt? Wie viele Farben könnten wir dann kodieren? Finde Argumente für und gegen diese Idee!
10. Wir hatten nicht immer die vollen 16.777.216 Farben auf unseren Computerbildschirmen. Recherchiere, wie sich Farbcodes im Laufe der Geschichte der digitalen Computer entwickelt haben. Kannst du einige Screenshots aus längst vergangenen Zeiten finden?
11. Du möchtest eine Audiodatei mit einer Länge von 1 Minute speichern. Du wählst eine Sample Rate von 4000 Hz und eine Bit Depth von 16 Bit. Wie groß wird die Datei?
12. Du hast eine Audiodatei (unkomprimiert) mit einer Länge von 10 Sekunden und einer Größe von 100 KB. Du weißt, dass die Sample Rate 5 KHz beträgt. Welche Bit Depth hat die Aufnahme?
13. Du entwirfst ein Logo für deine Firma, das du auf möglichst unterschiedlichen Medien, vom Flyer bis zum Plakat, in hoher Qualität drucken möchtest. Welches Dateiformat wählst du und warum?

14. Erläutere den Unterschied zwischen Codesystemen mit fester und variabler Länge. Nenne jeweils ein Beispiel.
15. Ist der Morsecode ein binäres Codesystem? Begründe deine Antwort.
16. Wie viele Bits verwendet der Braille-Code, um ein Symbol darzustellen? Ist der Braille-Code ein Binärcode?

6. Datenstrukturen

6.1. Wie stellen wir Daten im Computer dar?

6.2. Was ist eine Datenstruktur und wozu brauchen wir sie?

6.3. Welche Datenstrukturen gibt es?

6.3.1. Listen

6.3.2. Mengen

6.3.3. Graphen

6.3.4. Bäume

6.3.5. Warteschlangen

Übungsaufgaben

Teil III.

Verarbeitung

7. Analog vs. Digital



Zu diesem Kapitel gibt es ein Video auf YouTube. Scanne einfach den QR-Code oder klicke in der digitalen Version des Buches auf den QR-Code.

7.1. Was ist der Unterschied zwischen der analogen und digitalen Welt?

7.1.1. Endlich viele Möglichkeiten vs. Unendlichkeit

7.1.2. Diskrete und kontinuierliche Werte

8. Speicher

8.1. Ist die Speicherung von Informationen an ein Material gebunden?

8.1.1. Substratunabhängigkeit

8.2. Welche Speicherformen gibt es in der Praxis?

9. Logik und Arithmetik

9.1. Schalter

9.1.1. Relais, Vakuumröhren und Transistoren

9.2. Logikgatter

9.3. Binäre Addition

9.4. Binäre Subtraktion

9.5. Substratunabhängigkeit

10. Computer

10.1. Komponenten eines Computers

- von Neumann-Architektur

Teil IV.

Kommunikation

11. Signale

11.1. Was ist ein Signal?

“A signal is a function that conveys information about the behavior of a system or attributes of some phenomenon”.

Ein Signal ist eine physikalische Größe, die Informationen trägt. Stellen wir uns vor, du stehst an einer Straßenecke und winkst einem Freund auf der anderen Seite zu. Deine Handbewegung ist ein **optisches Signal**, das die Information „Komm her!“ überträgt. Signale können auch Schallwellen (wie deine Stimme), elektrische Impulse (in Kabeln) oder Licht (in Glasfasern) sein

11.1.1. Der Kern ist die Veränderung

Eine statische physikalische Größe kann keine Informationen übertragen. Dies haben wir bereits im Kapitel Kapitel 3 kennengelernt. Erst die Veränderung einer physikalischen Größe über die Zeit ermöglicht es, Informationen zu kodieren und zu übertragen. Ein einfaches Beispiel ist das An- und Ausschalten einer Lampe, wobei die Veränderung der Helligkeit das Signal darstellt. Die zeitliche Abfolge dieser Veränderungen kann dann als Nachricht interpretiert werden, wie etwa beim Morsecode.

11.1.2. Häufig elektrisch

Ein Signal muss nicht unbedingt eine elektrische Größe wie Spannung (gemessen in Volt) sein. Signale können auch mechanische Größen (Druck, Beschleunigung), optische Größen (Helligkeit, Farbe) oder akustische Größen (Schallwellen) sein. Das Entscheidende ist, dass sich das Signal als messbare Größe über die Zeit verändert und dadurch Informationen transportiert.

Für die Verarbeitung mit Computern müssen wir alle Signale in elektrische Signale umwandeln. Von dort aus ist es dann ein kleiner Schritt zum digitalen Signal, das nur die Werte Null oder Eins annehmen kann. Dafür nutzen wir sogenannte **Analog-to-Digital-Converter (ADC)**.

11.2. Wie können wir Signale erzeugen?

11.2.1. Von digital zu analog

11.2.2. Die LED

11.3. Wie können wir Signale empfangen?

11.3.1. Von analog zu digital

11.3.2. Der Farbsensor

11.4. Welche Bedeutung hat ein Signal?

Ein Signal ist zunächst nur eine messbare Abfolge von Veränderungen einer physikalischen Größe über die Zeit. Die Bedeutung, die wir diesen Veränderungen – oder der Modulation – zuschreiben, bestimmen wir selbst.

12. Protokolle

12.1. Peer-To-Peer

12.2. Handshake

12.3. TCP/IP & HTTPs

- How can we communicate between two computers?
 - Protocols / Handshake
 - Netzwerk / Internet / P2P-Communication

13. Verschlüsselung

13.1. Verschlüsselung

13.1.1. Symmetrisch

- Symmetric encryption with Caesar Cipher

13.1.2. Asymmetrisch

- Assymmetric with RSA Toy

13.2. Digitale Signaturen

- Digital signatures

14. Kompression

14.1. Verlustfreie Kompression

- Huffman

14.2. Verlustbehaftete Kompression

- JPEG

Teil V.

Programmierung

Die Programmierung ist der Schlüssel zur Entfaltung des vollen Potenzials von Computern. Computer führen auf Hardware-Ebene mithilfe winziger Bauteile blitzschnelle Berechnungen und Datenverarbeitungen durch. Das Herzstück dieser Operationen ist der Hauptprozessor (CPU).

Die CPU verarbeitet Befehle in einer komplexen Maschinensprache - einer Sprache, die für Menschen schwer verständlich und unpraktisch für die direkte Problemlösung ist. Deshalb nutzen wir höhere Programmiersprachen wie Python. Diese Sprachen sind mit ihrer verständlichen Syntax und ihrem natürlicheren Vokabular wesentlich besser für Menschen geeignet.

In diesem vierten Teil des Buches lernen wir die fundamentalen Konzepte und Werkzeuge kennen, mit denen wir Computer effektiv programmieren können. Wir verstehen, wie wir komplexe Probleme in klare Anweisungen übersetzen, die der Computer ausführen kann.

Diese Fähigkeit ist auch für das LiFi-Projekt unverzichtbar, um die Algorithmen für das Senden und Empfangen von Nachrichten sowie für die

15. Willkommen in Python

15.1. Unser erstes Programm

Unser erstes Programm könnte so aussehen:

```
# Ask the user for their name
name = input("What's your name? ")

# Greet the user
print(f"Hello {name}")
```

15.1.1. Nichts als Text

Ein **Programm** besteht aus einer Reihe von Anweisungen, die von einem Computer ausgeführt werden. Ein Programm wird als reiner Text geschrieben, und üblicherweise verwenden wir für jede neue Anweisung eine separate Zeile. Um ein Programm zu schreiben, benötigen wir lediglich einen einfachen Texteditor. Allerdings bevorzugen wir eine vollständige Entwicklungsumgebung wie [Visual Studio Code](#), da sie viele hilfreiche Funktionen wie Syntaxhervorhebung, automatische Vervollständigung und Debugging-Tools bietet, die das Programmieren erleichtern.

15.1.2. Zeile für Zeile

Wenn ein Computer ein Programm ausführt, geht er Zeile für Zeile durch das Programm und führt aus, was jede Zeile - oder **Anweisung** - ihm vorgibt. Im Beispiel oben wird zuerst die Zeile

```
name = input("What's your name? ")
```

ausgeführt. Diese Zeile nutzt die Funktion `input()`, um eine Eingabe vom Benutzer abzufragen. Das Ergebnis wird in einer Variable namens `name` gespeichert. Anschließend geht der Computer zur nächsten Zeile, wo er die Anweisung

```
print(f"Hello {name}")
```

vorfindet. `print()` ist eine Funktion, die eine Nachricht auf der Konsole ausgibt.

15.1.3. Kommentare

Vielleicht fragst du dich, was es mit den beiden Zeilen auf sich hat, die mit einem `#`-Symbol beginnen?

```
# Ask the user for their name  
# Greet the user
```

Diese Zeilen sind **Kommentare** und gehören nicht zum eigentlichen Programm. Der Computer überspringt sie – sie dienen ausschließlich uns Menschen als Hilfestellung. Ein gut platzierter Kommentar kann das Verständnis des Codes erheblich erleichtern.

15.1.4. Nicht nur sequenziell

Im Beispiel oben werden die Befehle Zeile für Zeile ausgeführt – das ist der Normalfall in einem Programm. Wir sprechen dann auch von einer sequenziellen Ausführung. Es gibt jedoch spezielle Anweisungen wie **Schleifen**, **Kontrollstrukturen** und **Funktionen**, die den Computer veranlassen, Befehle in einer anderen Reihenfolge auszuführen. Mit diesen Anweisungen kann der Computer bestimmte Aktionen wiederholen oder Codezeilen überspringen. Keine Sorge, wenn dir diese Konzepte noch nicht ganz klar sind – wir werden sie bald näher kennenlernen.

15.2. Ein Programm ausführen

Um ein Python-Programm auszuführen, benötigen wir einen Python-Interpreter – das Programm, das unseren Python-Code Zeile für Zeile liest und ausführt. Diesen musst du [für dein Betriebssystem herunterladen](#) und wie jedes andere Programm installieren. In Visual Studio Code können wir ein Python-Programm einfach mit einem Klick auf den „Play“-Button oder durch Drücken der Taste F5 starten. Ich persönlich bevorzuge jedoch den Weg über die Kommandozeile (oder Terminal), da wir so die volle Kontrolle über die Ausführung unseres Programms haben. Ein weiterer Vorteil: Wir lernen dabei wichtige Kommandozeilenbefehle kennen, die in der Softwareentwicklung unverzichtbar sind. Mach dich also mit dem Terminal vertraut.

Öffne direkt ein Terminal in Visual Studio Code über das Menü “Terminal” → “New Terminal”. Bei Windows-Systemen solltest du darauf achten, dass es sich um ein “Command Prompt”

(cmd) handelt. Bei Linux-Systemen heißt es “Bash”, auf einem Mac-Rechner heisst es einfach “Shell”.

Wenn unser Programm in einer Datei namens `my_first_program.py` im aktuellen Verzeichnis gespeichert ist, können wir es mit dem `python`-Befehl ausführen:

```
python my_first_program.py
```

Nach Eingabe des Namens und Drücken der Enter-Taste erscheint eine freundliche Begrüßung.

15.3. Befehle in Python

15.3.1. Eingebaute Befehle

In unserem ersten Programm haben wir bereits zwei wichtige Befehle kennengelernt, die wir in Python verwenden können:

- `input()`: Liest einen Text über die Tastatur ein.
- `print()`: Gibt einen Text auf der Konsole (Terminal) aus.

Beide Befehle stehen uns direkt zur Verfügung, da sie Teil der Standard-Python-Installation sind. Neben diesen beiden Befehlen gibt es noch viele weitere solche vordefinierten Funktionen.

15.3.2. Befehle aus mitgelieferten Modulen

Neben den standardmäßig verfügbaren Befehlen gibt es in Python auch spezialisierte Funktionen, die erst importiert werden müssen. Ein gutes Beispiel ist die Funktion `sqrt()` zur Berechnung von Quadratwurzeln. Diese Funktion ist im `math`-Modul enthalten, das zwar mit Python installiert wird, aber erst durch eine explizite Import-Anweisung in unserem Programm verfügbar wird. Dafür verwenden wir folgenden Befehl:

```
from math import sqrt
```

Anschließend können wir diese Funktion in unserem Programm nutzen, um eine Quadratwurzel zu berechnen:

```
square_root = sqrt(16)
```

Auf diese Weise laden wir aus dem Modul `math` nur die einzelne Funktion `sqrt`. Alternativ können wir auch alle Funktionen des Moduls `math` auf einmal laden:

```
import math
```

Wenn wir die Funktion auf diese Weise laden, müssen wir den qualifizierten Namen verwenden und den Modulnamen durch einen Punkt getrennt angeben:

```
square_root = math.sqrt(16)
```

In Abbildung 15.1 werden weitere Beispiele für Befehle aus sogenannten *built-in modules*, also in Python eingebauten Modulen, gezeigt. Außerdem sehen wir ganz rechts eine dritte Kategorie von Funktionen: solche, die über externe Module eingebunden werden.



Abbildung 15.1.: Unterschiedliche Arten von Befehlen in Python.

15.3.3. Befehle aus externen Modulen

Die externen Module sind nicht Bestandteil der Python-Installation und müssen daher getrennt installiert werden. Dafür verwenden wir den mitgelieferten Paketmanager Pip:


```
pip install tinkerforge
```

Das Beispiel oben installiert, wenn wir den Befehl im Terminal eingeben, das Modul mit dem Bezeichner `tinkerforge`. Dabei handelt es sich um eine Bibliothek, die es uns ermöglicht, mit der Tinkerforge-Hardware zu kommunizieren, die wir im Rahmen des LiFi-Projektes verwenden. Nach der Installation können wir die Funktionen des Moduls in unserem Programm verwenden, genauso wie wir es bei den eingebauten Modulen getan haben. Solche externen Module erweitern die Funktionalität von Python erheblich und ermöglichen uns den Zugriff auf verschiedenste Hardware und Dienste.

15.3.4. Argumente eines Befehls

Ein Befehl (oder eine Funktion) in Python kann Argumente entgegennehmen – das sind Werte, die wir dem Befehl zur Verarbeitung übergeben. Diese Argumente werden in runden Klammern nach dem Befehlsnamen angegeben. Bei der Funktion `print()` ist das zum Beispiel der Text, den wir ausgeben möchten, und bei `sqrt()` ist es die Zahl, deren Quadratwurzel wir berechnen wollen.

Es gibt auch Befehle, die kein Argument benötigen. Die `input()`-Funktion ist ein Beispiel dafür – obwohl wir ihr bisher immer einen Text in den Klammern mitgegeben haben, damit der Benutzer weiß, was er eingeben soll:

```
name = input("What's your name? ")
```

Auf der Konsole erscheint dann der Text “What’s your name?”, gefolgt von einem blinkenden Cursor, der die erwartete Eingabe signalisiert.

Wir können die Funktion auch ohne Argument aufrufen:

```
name = input()
```

In diesem Fall erscheint nur der blinkende Cursor ohne Text. Wichtig: Auch wenn wir kein Argument übergeben, müssen wir die runden Klammern angeben. Sie sind das universelle Zeichen für einen Funktionsaufruf.

15.3.5. Rückgabewerte eines Befehls

Programme erzeugen nach dem EVA-Modell (Eingabe-Verarbeitung-Ausgabe) Ergebnisse. In Programmiersprachen wie Python können nicht nur Programme, sondern auch einzelne Befehle oder Funktionen Ergebnisse produzieren. Diese Ergebnisse nennen wir Rückgabewerte.

Ein typisches Beispiel ist die `input`-Funktion:

```
name = input("What's your name?")
```

Diese Funktion wartet auf eine Benutzereingabe und gibt den eingegebenen Text als Rückgabewert zurück. Mit dem Zuweisungsoperator (=) speichern wir diesen Rückgabewert in der Variable `name`, um ihn später im Programm weiterverwenden zu können. Das Konzept der Rückgabewerte und Variablen werden wir in den folgenden Kapiteln noch ausführlicher behandeln.

15.4. Fehler

Wenn wir ein Python-Programm ausführen, können verschiedene Fehler auftreten. Die häufigsten sind **Syntaxfehler**, die entstehen, wenn wir uns nicht an die Regeln der Python-Sprache halten, zum Beispiel wenn wir vergessen, Klammern zu schließen oder Einrückungen falsch setzen. **Laufzeitfehler** treten dagegen erst während der Programmausführung auf, etwa wenn wir versuchen, durch Null zu teilen oder auf nicht vorhandene Dateien zuzugreifen.

Übungsaufgaben

15.4.1. Reflektionsfragen

1. Welche Befehle kennst du, die zum Funktionsumfang von Python gehören? Schlag in der [Dokumentation](#) nach und nenne mindestens 5 Befehle und erkläre ihren Zweck!
2. Wozu können Kommentare in einem Programm nützlich sein, obwohl sie vom Computer ignoriert werden?
3. Was versteht man unter dem Begriff *bug* in der Programmierung? Welche Arten kennst du?
4. Nenne 3 Beispiele für Laufzeitfehler!

15.4.2. Programmieraufgaben

1. Schreibe ein Programm, das den Benutzer um die Eingabe seines Namens bittet und ihn anschließend unter Nennung seines Namens begrüßt! Formuliere dein Programm zunächst als Pseudocode und schreibe die Schritte als Kommentare in dein Programm!
2. Recherchiere, wie wir den Benutzer um die Eingabe von Zahlen bitten können und wie wir diese Zahlen in Python verarbeiten können! Schreibe ein Programm, das den Benutzer um die Eingabe von zwei Zahlen bittet und anschließend die Summe der beiden Zahlen ausgibt! Welche Art von Fehler kann dabei auftreten?

3. Nutze das Modul `math` und schreibe ein Programm, das die Quadratwurzel einer Zahl berechnet, die der Nutzer eingibt.
4. Schreibe ein Python-Programm, das eine Temperatur in Grad Celsius vom Nutzer abfragt und anschließend die entsprechende Temperatur in Grad Fahrenheit berechnet und ausgibt. Nutze dazu folgende Umrechnungsformel: $F = C \times \frac{9}{5} + 32$
5. Schreibe ein Programm, das den Nutzer nach seinem Namen, Alter und seiner Heimatstadt fragt und anschließend eine freundliche Zusammenfassung ausgibt. Verwende dabei sinnvolle Kommentare, um deinen Code verständlich zu machen.

16. Variablen und Datentypen

16.1. Wozu dienen Variablen?

Programme bestehen aus einer Abfolge von Anweisungen, die der Computer von oben nach unten ausführt. Jede Anweisung kann bei ihrer Ausführung ein Ergebnis erzeugen, beispielsweise wenn wir eine einfache Berechnung durchführen.

```
width * height
```

Das obige Beispiel zeigt eine einfache Berechnung einer Rechteckfläche anhand von Breite und Höhe. Um das Ergebnis dieser Berechnung in späteren Programmschritten nutzen zu können, müssen wir es speichern. Hierfür verwenden wir in der Programmierung eine Variable.

```
area = width * height
```

Im Beispiel oben weisen wir der Variable `area` das Ergebnis der Flächenberechnung zu. Eine Variable funktioniert dabei wie ein Notizzettel, auf dem wir uns etwas Wichtiges aufschreiben. Über den Variablennamen können wir dann jederzeit innerhalb unseres Programms auf den gespeicherten Wert zugreifen. Zum Beispiel können wir ihn auf der Konsole ausgeben:

```
print(area)
```

Oder wir nutzen die Fläche, um das Volumen einer Kiste zu berechnen, nachdem der Benutzer dem Programm die Tiefe mitgeteilt hat:

```
volume = area * depth
```

16.2. Konstanten

Eine Konstante ist ein Sonderfall einer Variable. Der wesentliche Unterschied besteht darin, dass einer Konstante nur einmal ein Wert zugewiesen wird, der sich im weiteren Programmverlauf nicht mehr ändert. Dies ähnelt dem Konzept mathematischer Konstanten wie π , die einen unveränderlichen Wert besitzen.

In Python definieren wir Konstanten, indem wir alle Buchstaben des Namens groß schreiben:

```
PI = 3.14159
INPUT_FILE = "data.csv"
```

In Python gibt es technisch gesehen keine echten Konstanten – sie existieren nur als Konvention. Programmierer nutzen die Großschreibung des Namens, um zu signalisieren, dass der Wert nicht mehr verändert werden soll. Wird der Wert dennoch geändert, führt dies zu keinem technischen Fehler. Dies unterscheidet Python von anderen Programmiersprachen wie Java, wo Konstanten tatsächlich unveränderlich sind.

16.3. Benennung von Variablen

Den Namen einer Variable vergeben wir übrigens selbst, wenn wir ein Programm schreiben. Anstatt wie im Beispiel oben die Variable `volume` zu nennen, hätten wir sie ebenso gut `volumen` nennen können. Grundsätzlich sind wir frei bei der Benennung von Variablen, wenn wir uns an ein paar kleine Regeln halten, die wir gleich besprechen. Es gibt neben diesen Regeln, die wir einhalten *müssen*, weil es sonst zu Fehlern kommt, auch eine Konvention unter Programmierern, sich an bestimmte Abmachungen zu halten. Dazu zählt zum Beispiel, dass wir Variablennamen in Englischer Sprache halten, damit Programme weltweit geteilt und verstanden werden können.

16.3.1. Regeln bei der Benennung von Variablen

Die zwingenden Regeln sind oft spezifisch für die Programmiersprache, die wir verwenden. Allerdings gibt es zwischen den Sprachen bezüglich dieser Regeln eine große Einigkeit. Für Python gelten folgende Regeln, die bei Missachtung zu einem Fehler führen:

- Ein Variablenname muss mit einem Buchstaben beginnen.
- In einem Variablennamen dürfen nur Buchstaben aus dem römischen Alphabet, Zahlen und Unterstriche (`_`) verwendet werden. Leerzeichen, Umlaute oder andere Sonderzeichen sind nicht erlaubt.
- Ein Variablenname darf kein reserviertes Schlüsselwort der Programmiersprache sein. Beispielsweise können wir keine Variable `print` oder `if` nennen, da diese Wörter in Python bereits eine besondere Bedeutung haben.

16.3.2. Konventionen bei der Benennung von Variablen

Neben den festen Regeln solltest du folgende Konventionen unbedingt berücksichtigen:

- Ein Variablenname sollte mit einem Kleinbuchstaben beginnen. Also `area` und nicht `Area`.
- Variablennamen sollten in englischer Sprache gehalten werden. Also `area` und nicht `flaeche`.
- Variablennamen sollten aussagekräftig sein und ihre Bedeutung klar vermitteln. Ein Name wie `a` anstelle von `area` wäre ungeeignet, auch wenn er kürzer ist. Dank der Autovervollständigung in unserem Code-Editor sind längere Variablennamen kein Problem – sie sind sogar vorzuziehen, da sie die Lesbarkeit des Programms verbessern.
- Wenn ein Variablenname aus zwei oder mehreren Wörtern zusammengesetzt ist, dann solltest du einen Unterstrich zur Trennung der Wörter verwenden. Also `area_rectangle` statt `arearectangle`. Auch das erhöht die Lesbarkeit des Programms.

Dazu sei noch erwähnt, dass Python Groß- und Kleinschreibung unterscheidet. Das heisst konkret, dass die Variablen `area` und `Area` in einem Programm zwei unterschiedliche Variablen sind.

16.4. Datentypen

16.4.1. Zeichenketten (*Strings*)

Variablen können verschiedene Arten von Daten speichern. In den obigen Beispielen haben wir Zahlen gespeichert (Fläche, Volumen). Neben Zahlen können wir auch Zeichenketten in Variablen speichern, zum Beispiel eine E-Mail-Adresse:

```
email = "m.mustermann@mail.de"
```

Eine Zeichenkette erkennt man an den Anführungszeichen, die den Wert umschließen. Die Anführungszeichen selbst gehören nicht zum Wert, sondern dienen nur der Abgrenzung. Dabei können sowohl doppelte als auch einfache Anführungszeichen verwendet werden:

```
email = 'm.mustermann@mail.de'
```

Das Besondere an diesem Datentyp ist, dass innerhalb der Anführungszeichen eine beliebige Abfolge von Zeichen stehen darf. Es können also ganze Texte als String abgebildet werden. Aber auch eher kryptische Werte wie Passwörter:

```
password = "%?Zha$-ÄiK,!/"
```

Eine Besonderheit tritt auf, wenn innerhalb einer Zeichenkette ein Anführungszeichen vorkommen soll. Da das Anführungszeichen schon als Begrenzungszeichen fungiert, müssen wir es mit einem Backslash kennzeichnen. Der Backslash fungiert dann als so genanntes Escape-Zeichen und ist selbst nicht Teil der Zeichenkette:

```
password_with_quotation_mark = "abc\"123"
```

Der Wert der Variable oben wäre somit `abc"123`, der Backslash ist selbst nicht sichtbar. Was aber, wenn wir nun einen Backslash in einer Zeichenkette verwenden möchten? Ganz einfach, dann brauchen wir zwei Backslashes. Der erste „escaped“ den zweiten:

```
string_with_backslash = "abc\\123"
```

Der Wert der Variable wäre somit `abc\123`.

16.4.2. Numerische Werte (*Integer* und *Float*)

Zahlen sind in Algorithmen und Programmen von zentraler Bedeutung, da wir regelmäßig Berechnungen durchführen müssen. Wie im Beispiel zu Beginn dieses Tutorials gezeigt, haben wir die Fläche eines Rechtecks berechnet und das Ergebnis in einer Variable gespeichert. Solch ein Ergebnis kann entweder eine ganze Zahl oder eine Zahl mit Dezimalstellen sein. Python und andere Programmiersprachen unterscheiden zwischen ganzen Zahlen und Dezimalzahlen, da beide unterschiedliche Anforderungen an den Speicher stellen. Ganze Zahlen werden in Python als *Integer* bezeichnet, während Dezimalzahlen (oder Gleitkommazahlen) als *Float* bekannt sind.

Ob es sich um den einen oder den anderen numerischen Datentyp handelt, wird anhand des Ausdrucks entschieden, der einer Variable zugewiesen wird. Betrachten wir zunächst den einfachsten Fall, die direkte Zuweisung eines Wertes:

```
width = 5  
height = 2.5
```

Im obigen Beispiel wird der Variable `width` eine ganze Zahl zugewiesen, weshalb Python den Datentyp *Integer* wählt. Die Variable `height` erhält eine Dezimalzahl, wodurch Python automatisch den Datentyp *Float* verwendet.

```
area = width * height
```

Welchen Datentyp hat nun die Variable **area**? Da das Produkt einer ganzen Zahl und einer Dezimalzahl eine Dezimalzahl ergeben kann, weist Python automatisch den Datentyp *Float* zu – selbst wenn das Ergebnis theoretisch eine ganze Zahl sein könnte. Python geht auf Nummer sicher. Wäre **height** allerdings ebenfalls eine ganze Zahl, dann wäre auch das Ergebnis **area** vom Typ *Integer*.

16.4.3. Wahrheitswerte (*Boolean*)

Für manche Sachverhalte wollen wir lediglich wissen, ob eine Aussage wahr oder falsch ist. Ist der Kunde bereits 18 Jahre alt? Besitzt der Fahrer die Erlaubnis, einen 3,5-Tonner zu fahren? Wurde die Prüfung bestanden? In solchen Fällen gibt es nur zwei mögliche Antworten: „Ja“ oder „Nein“ – oder in der Sprache der Logik: „Wahr“ oder „Falsch“. Für diesen Zweck existiert der Boolesche Datentyp, der nur zwei mögliche Werte annehmen kann: **True** und **False**.

```
is_underage = False  
exam_passed = points_reached >= 50
```

Im Codebeispiel in Zeile 1 weisen wir der Variable **is_underage** den Wert **False** zu. Beachtet, dass dieser Datentyp, anders als eine Zeichenkette, keine Anführungszeichen benötigt. Die Begriffe **True** und **False** sind so genannte Schlüsselwörter in Python und werden automatisch als Boolesche Werte erkannt.

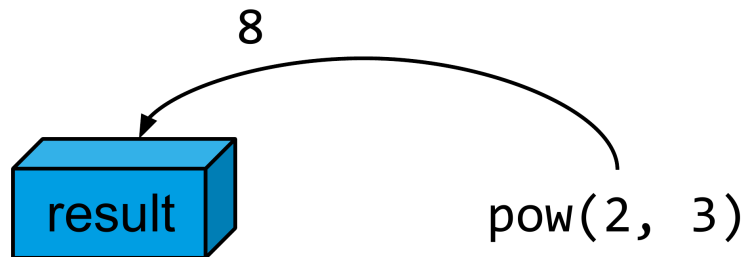
Die zweite Zeile weist keinen direkten Wahrheitswert zu, sondern das Ergebnis eines logischen Ausdrucks. Dieser Ausdruck wird entweder zu wahr oder falsch ausgewertet, abhängig vom aktuellen Wert der Variable **points_reached**. Diese Punktzahl wurde vielleicht in vorherigen Programmzeilen berechnet.

16.4.4. Komplexe Datentypen

Variablen können neben den primitiven Datentypen wie Zeichenketten, Zahlen oder Boolesche Werte auch komplexere Dinge speichern. Im Prinzip lässt sich alles in einer Variable speichern, was in Python existiert. Dazu gehören beispielsweise Funktionen, die wir in einem späteren Tutorial kennenlernen. Auch sogenannte Collections (zu Deutsch: Sammlungen) zählen dazu. Collections sind Strukturen, die mehrere primitive Datentypen enthalten können. Ein Beispiel dafür ist eine Liste von Zahlen. Collections werden wir in einem späteren Tutorial ausführlich behandeln.

16.5. Variablen im Computer

Eine Variable lässt sich bildlich als Behälter mit einem Namensetikett vorstellen. In diesen Behälter können wir beliebige Werte ablegen. Mithilfe des Namens finden wir den Behälter jederzeit wieder, können seinen Inhalt einsehen und für weitere Zwecke verwenden. Wir können auch einen neuen Wert in den Behälter legen – dabei wird der bisherige Inhalt entweder gelöscht oder muss zuvor in einen anderen Behälter umgelagert werden.



In Wirklichkeit sind Variablen bestimmte Bereiche im Arbeitsspeicher (RAM) des Computers. Jeder dieser Speicherorte besitzt eine eindeutige Adresse. Bei der Zuweisung eines Wertes zu einer Variable speichert der Computer diesen Wert an einer bestimmten Adresse im RAM. Der Variablenname wird dann mit dieser Adresse verknüpft, damit wir den Wert später über den Namen abrufen können. Eine Variable ist also im Grunde ein Zeiger auf eine bestimmte Stelle im Computerspeicher.

Übungsaufgaben

16.5.1. Reflektionsfragen

1. Warum sind Variablen in einem Programm nützlich?

2. Wie kann man den Wert einer Variable auf der Konsole ausgeben?
3. Was ist der Unterschied zwischen einer Variablen und einer Konstante?
4. Wie kann man in Python eine Konstante definieren?
5. Welche Zeichen sind in einem Variablennamen erlaubt?
6. Was bedeutet es, dass Python *case sensitive* ist?
7. Beschreibe die Namenskonventionen für Variablen in Python und erkläre, warum es wichtig ist, diese Konventionen zu befolgen.
8. Warum ist es eine gute Praxis, Variablennamen auf Englisch zu schreiben?
9. Was sind reservierte Schlüsselwörter, und warum dürfen sie nicht als Variablennamen verwendet werden?
10. Welche primitiven Datentypen kennst du?
11. Welche numerischen Datentypen unterscheidet Python?
12. Warum muss man Anführungszeichen benutzen, um eine Zeichenkette zu definieren?
13. Wie kann man in einer Zeichenkette ein Anführungszeichen verwenden, ohne einen Fehler zu verursachen?
14. Erläutere die Funktionsweise einer Format-Strings in Python? Was passiert mit den Platzhaltern, wenn wir das "f" als Präfix weglassen?
15. Was ist der Boolesche Datentyp, und welche Werte kann er annehmen?
16. Warum werden `True` und `False` in Python nicht in Anführungszeichen geschrieben?
17. Was passiert mit dem alten Wert einer Variable, wenn ihr ein neuer Wert zugewiesen wird?
18. Beschreibe mit eigenen Worten, was unter einer *Variable* zu verstehen ist! Was ist eine Variable aus technischer Sicht?

16.5.2. Programmieraufgaben

Erstelle für jede der folgenden Aufgaben ein separates Python-Programm in einer eigenen Datei mit der Endung `.py`!

1. Schreibe ein kleines Python-Programm, das eine Variable für die Fläche eines Kreises berechnet und speichert. Nutze für den Wert von π die Konstante aus dem `math` Modul.
2. Definiere eine Variable mit einer Zeichenkette, die einen Backslash enthält, und gib sie auf der Konsole aus. Was beobachtest du?

3. Addiere zwei Zeichenketten und lass dir das Ergebnis anzeigen! Multipliziere anschließend eine Zeichenkette mit einer ganzen Zahl!
4. Frage den Benutzer nach seinem Vor- und Nachnamen und speichere das Ergebnis in einer Variablen `name`. Teile diese Variable anschließend in zwei Variablen auf: `first_name` für den Vornamen und `last_name` für den Nachnamen. Nutze die `find()`-Funktion, um das Leerzeichen zwischen Vor- und Nachnamen zu finden. Speichere die gefundene Position auf einer weiteren Variable. Verwende dann die Notation mit eckigen Klammern, um die beiden Namensteile zu extrahieren und in den entsprechenden Variablen zu speichern.
5. Schreibe ein Programm, in dem du zwei boolesche Werte addierst und subtrahierst. Betrachte und interpretiere das Ergebnis. Multipliziere anschließend einen booleschen Wert mit einem numerischen Wert. Was passiert?
6. Erstelle eine beliebige Variable und gib die Speicheradresse der Variable auf der Konsole aus!

17. Funktionen

- Funktionen definieren
- Funktionen verwenden
- Module laden und Funktionen daraus verwenden

Übungsaufgaben

17.0.1. Reflektionsfragen

1. Erkläre, wann es sinnvoll ist, Code in eine Funktion auszulagern.
2. Was solltest du bei der Benennung einer Funktion beachten?
3. Funktionen können aus verschiedenen Quellen stammen. Erläutere die folgenden Arten:
 - a. Built-in Funktionen
 - b. Funktionen aus internen Modulen
 - c. Funktionen aus externen Modulen
 - d. Selbst geschriebene Funktionen
4. Warum ist es nicht sinnvoll, hinter einer **return**-Anweisung innerhalb einer Funktion weitere Anweisungen hinzuzufügen?
5. Erläutere die Begriffe *Parameter* und *Rückgabewert* im Zusammenhang mit Funktionen in der Programmierung und nenne jeweils ein Beispiel!

17.0.2. Programmieraufgaben

1. Schreibe eine Funktion, die prüft, ob eine Zahl größer als die andere ist. Die Funktion soll beide Zahlen als Parameter entgegennehmen und **True** zurückgeben, wenn die erste Zahl größer ist als die zweite, andernfalls **False**.
2. Definiere eine Funktion `celsius_to_fahrenheit`, die einen Celsius-Wert entgegennimmt und den entsprechenden Fahrenheit-Wert zurückgibt. Die Formel lautet: $^{\circ}F = ^{\circ}C \times \frac{9}{5} + 32$
3. Schreibe die beiden Funktionen `square(x)` und `cube(x)`. Teste beide, indem du den Benutzer um einen Wert bittest und beide Ergebnisse auf der Konsole ausgibst.

4. Erstelle eine Funktion, die BMI berechnet. Frage den Benutzer nach den benötigten Eingaben und gib das Ergebnis aus.
5. Erstelle eine weitere Funktion, um auf Basis des BMI zu ermitteln, ob eine Person untergewichtig ($< 18,5$), normalgewichtig (zwischen 18,5 und 25) oder übergewichtig ist (> 25).
6. Erstelle eine Funktion, die prüft, ob ein String ein Palindrom ist. Sie soll **True** oder **False** zurückliefern. Groß- und Kleinschreibung soll ignoriert werden.
7. Schreibe eine Funktion `factorial(n)`, die die Fakultät einer ganzen Zahl berechnet. Prüfe, wie das Konzept der *Rekursion* dir dabei helfen kann!
8. Implementiere ein kleines Spiel, bei dem der Computer sich eine Zufallszahl zwischen 1 und 100 ausdenkt, und der Spieler solange raten muss, bis er sie trifft. Implementiere dazu die folgenden Funktionen:
 - a. `generate_secret(min, max)` - erzeugt eine zufällige Zahl zwischen `min` und `max`
 - b. `get_guess_from_player()` - fragt den Spieler nach seinem Tipp
 - c. `check_guess(secret, guess)` - prüft, ob der Tipp zu hoch, zu niedrig, oder richtig war.
 - d. **Hinweis:** Du wirst eine Schleife benötigen - lies nach wie das geht.
9. Schreibe eine Funktion `generate_password(length)` die ein Passwort der angegebenen Länge zurückgibt. Das Passwort soll aus Groß- und Kleinbuchstaben, Ziffern und Sonderzeichen bestehen. Hinweise: Schau dir das `string`-Modul in Python an und insbesondere die dort vordefinierten Listen wie `string.ascii_lowercase`, `string.ascii_uppercase`, `string.digits` und `string.punctuation`.

18. Collections

Übungsaufgaben

18.0.1. Reflektionsfragen

18.0.2. Programmieraufgaben

19. Kontrollstrukturen

Übungsaufgaben

19.0.1. Reflektionsfragen

19.0.2. Programmieraufgaben

20. Schleifen

Übungsaufgaben

20.0.1. Reflektionsfragen

1. Welche Arten von Schleifen kennst du, und wann würdest du sie einsetzen? Beschreibe für jede Schleifenart ein anschauliches Beispiel.

20.0.2. Programmieraufgaben

1. Schreibe ein Programm, das die Gregory-Leibniz-Serie für eine Anzahl Durchläufe n berechnet, die als Eingabe des Benutzers abgefragt wird. Die Approximation für π soll als Ausgabe auf der Kommandozeile erscheinen.
2. Schreibe ein Programm, das die Summe aller Zahlen von 1 bis 100 berechnet und ausgibt. Verwende dazu eine Schleife!
3. Du hast in der Vorlesung im Zusammenhang mit dem **Traveling Salesman Problem** den Brute Force-Algorithmus kennengelernt. Nutze deine Pythonkenntnisse, um folgende Probleme zu lösen:
 - a. Schreibe ein Programm zur Berechnung der Laufzeit des Brute-Force-Ansatzes beim Traveling Salesman Problem (TSP). Das Programm soll die Anzahl der Städte als Benutzereingabe entgegennehmen und die Anzahl möglicher Routen zusammen mit der Laufzeit in Sekunden und Jahren ausgeben. Die Laufzeit des Brute-Force-Ansatzes beträgt $O(n!)$, wobei n die Anzahl der Städte ist. Schreibe dazu deine eigene Funktion `factorial()` und verwende darin eine Schleife, um die Fakultät der Anzahl der Städte zu berechnen. Nimm an, dass ein Schritt eine halbe Nanosekunde dauert. Dies entspricht einer Prozessortaktung von 2 GHz.
 - b. Lass Dir von deinem Programm die Laufzeiten für 10, 15 und 20 Städte ausgeben. Wie viele Jahre würde es jeweils dauern, um alle Routen für die jeweilige Anzahl Städte zu berechnen?
 - c. Untersuche, welchen Einfluss die Prozessortaktung oder die benötigte Zeit in Nanosekunden für einen Schritt auf die Laufzeit hat. Was passiert, wenn die Prozessortaktung auf 3 GHz erhöht wird?

4. Du hast in der Vorlesung Algorithmen zur **Annäherung der Kreiszahl** π kennengelernt. Versuche dich nun in der Umsetzung mit Python:
- Schreibe ein Programm, das die Kreiszahl π mit Hilfe der Formel von Leibniz berechnet. Die Formel lautet: $\pi = 4 \cdot (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots)$. Verwende dazu eine Schleife, die die ersten 1000 Glieder der Reihe berechnet. Gib die Annäherung von π nach 1000 Schritten aus.
 - Schreibe ein Programm, das die Kreiszahl π mit Hilfe einer Monte Carlo-Simulation annähert. Die Formel zur Berechnung der Kreiszahl lautet: $\pi = 4 \cdot \frac{N_C}{N_Q}$, wobei N_C die Anzahl der Punkte im Kreis und N_Q die Anzahl der Punkte im Quadrat ist. Verwende dazu eine Schleife, die 1000 Punkte zufällig im Quadrat verteilt und zähle, wie viele Punkte innerhalb des Kreises liegen. Gib die Annäherung von π nach 1000 Schritten aus.

Hinweis: Du kannst feststellen, ob ein Punkt innerhalb des Kreises liegt, indem du die Entfernung des Punktes zum Kreismittelpunkt berechnest. Nutze dazu die Kreisformel $x^2 + y^2 = r^2$, die bei einem Kreis mit $r = 1$ entsprechend einfach zu überprüfen ist.

21. Fehlersuche- und Behandlung

Übungsaufgaben

21.0.1. Reflektionsfragen

21.0.2. Programmieraufgaben

Literaturverzeichnis

- Adami, Christoph. 2016. „What is Information;“ *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374 (2063): 20150230. <https://doi.org/10.1098/rsta.2015.0230>.
- Brookshear, J. Glenn, und Dennis Brylow. 2020. *Computer science: an overview*. 13th edition, global edition. NY, NY: Pearson.
- Petzold, Charles. 2022. *Code: the hidden language of computer hardware and software*. 2. Aufl. Hoboken: Microsoft Press.
- Pólya, George, und John Horton Conway. 2004. *How to solve it: a new aspect of mathematical method*. Expanded Princeton Science Library ed. Princeton science library. Princeton [N.J.]: Princeton University Press.
- Scott, John C. 2009. *But how do it know?: the basic principles of computers for everyone*. Oldsmar, FL: John C. Scott.

Index

Algorithmus, [42](#)

Bayes'sche Optimierung, [57](#)

Dezimalsystem, [80](#)

Dijkstra-Algorithmus, [47](#), [57](#)

Divide-and-Conquer-Ansatz, [55](#)

Evolutionäre Algorithmen, [56](#)

Flussdiagramm, [44](#)

Maschinelles Lernen, [58](#)

Monte-Carlo-Methode, [58](#)

Programm, [43](#)

Prozess, [43](#)

Pseudocode, [44](#)

Python, [20](#)

Stellenwertsystem, [79](#)

Traveling Salesman-Problem, [57](#)