Kantonsschule Im Lee

Informatik

Induktion und Rekursion

Skript

Thomas Graf

[git] ▪ Branch: main @ fdbb5a ▪ Release: (2025-01-09)

Inhaltsverzeichnis

Kapitel 1

Vorwissen

Diese Unterlagen behandeln anspruchsvolle Inhalte der Informatik und Mathematik. Um ihnen gerecht zu werden, bedarf es einer mathematisch präzisen Ausdrucksweise. Den überwiegenden Teil der benötigten mathematischen Werkzeuge werden die Lesenden typischerweise in den ersten Jahren des Gymnasiums kennengelernt haben. Einige der von uns benötigten Konzepte werden den meisten Lesenden jedoch vermutlich noch nicht bekannt sein. Diese Konzepte wollen wir in diesem Kapitel einführen. Wir werden dies in kompakter Form tun und an verschiedenen Stellen auf artifiziell wirkende (forcierte) Beispiele bewusst verzichten.

1.1 Mathematik

1.1.1 Direkter und indirekter Beweis

Seien A und C mathematische Aussagen. Wir möchten beweisen, dass die Aussage

$$A \Rightarrow C$$

(A impliziert C) gilt. Solch ein Beweis kann im Wesentlichen auf zwei Arten erbracht werden: entweder durch einen ***direkten Beweis*** oder einen ***indirekten Beweis***.

1.1.1.1 Direkter Beweis

Der direkte Beweis macht von folgender logischen Tatsache Gebrauch: Impliziert A eine weitere Aussage B

$$A \Rightarrow B$$

und B impliziert wiederum C:

$$B \Rightarrow C,$$

dann impliziert A auch C. Zusammengefasst gilt also:

$$((A \Rightarrow B) \text{ und } (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)\,. \tag{1.1}$$

Um die Richtigkeit der Implikation $A \Rightarrow C$ zu beweisen, zerlegt man die Implikation in bereits als für richtig befundene „Teilaussagen“ $A \Rightarrow B$ und $B \Rightarrow C$, also

$$(A \Rightarrow B) \text{ und } (B \Rightarrow C).$$

Danach folgt die Implikation $A \Rightarrow C$ aus Ausdruck 1.1. Diese Strategie kann wiederholt angewendet werden und man erhält eine „Kette“ logischer Implikationen (Schlüsse):

$$((A \Rightarrow B_1) \text{ und } (B_1 \Rightarrow B_2) \text{ und } (B_2 \Rightarrow B_3) \text{ und } \ldots \text{ und } (B_n \Rightarrow C)) \Rightarrow (A \Rightarrow C).$$

Dabei können die Begründungen der Implikationen $A \Rightarrow B_1$, $B_n \Rightarrow C$ sowie $B_k \Rightarrow B_{k+1}$ für jedes $k \in \{1, 2, \ldots, n-1\}$ als logische „Zwischenschritte“ verstanden werden.

1.1.1.2 Indirekter Beweis

Ein indirekter Beweis (Beweis durch Widerspruch) beginnt mit der Annahme, dass die Aussage C falsch sei, dass also $\neg C$ (nicht C) richtig ist. Nun wird einzig, unter Verwendung der Richtigkeit von A und $\neg C$ sowie bereits als wahr erkannter mathematischer Aussagen, die Richtigkeit einer Aussage B abgeleitet, von der bereits bekannt ist, dass sie falsch ist. Dadurch haben wir einen „Widerspruch“ erhalten und können folgern, dass $\neg C$ nicht richtig sein kann und somit C wahr sein muss. Damit ist, wie gewünscht, die Implikation $A \Rightarrow C$ nachgewiesen.

1.1.2 Kontraposition

Seien A und B mathematische Aussagen. Logisch äquivalent zur Behauptung $A \Rightarrow B$ ist die Aussage $\neg B \Rightarrow \neg A$, welche ***Kontraposition*** der Implikation $A \Rightarrow B$ genannt wird. Der Beweis für $A \Rightarrow B$ ist demnach auch erbracht, falls man zeigen kann, dass aus der Annahme, die Folgerungen seien nicht erfüllt, folgt, dass auch die Voraussetzungen nicht erfüllt sein können. Gelegentlich fällt es einem leichter, die Kontraposition $\neg B \Rightarrow \neg A$ einer Implikation $A \Rightarrow B$ zu zeigen.

Beispiel 1.1:
Sei A die Aussage „Es hat geregnet.“ und B die Aussage „Die Strasse ist nass.“. Die Implikation $A \Rightarrow B$ bedeutet: „Falls es geregnet hat, ist die Strasse nass.“

Falls die Strasse nass ist, muss das umgekehrt nicht bedeuten, dass es geregnet hat. Beispielsweise könnte die Strasse auch von der Strassenreinigung nass gemacht worden sein. Was wir aber sicherlich sagen können, ist, dass falls die Strasse *nicht* nass ist, es auch *nicht* geregnet haben kann, was genau $\neg B \Rightarrow \neg A$ bedeutet.

1.1.3 Funktionen

1.1.3.1 Definition einer Funktion

Definition 1.1 (Funktion als Vorschrift)**:**
Seien X und Y Mengen. Wir sagen, dass eine ***Funktion*** auf X mit Werten in Y definiert ist, wenn aufgrund einer *Vorschrift* (*Regel*) f jedem Element $x \in X$ genau ein Element $y \in Y$ zugehörig ist.

Wir sagen dann, dass die Menge X die ***Definitionsmenge*** der Funktion ist.

Das Symbol x, das benutzt wird, um ein allgemeines Element dieser Menge zu beschreiben, wird ***Argument*** oder *unabhängige Variable* der Funktion genannt.

Das Element $y_0 \in Y$, das einem Argument $x_0 \in X$ zugeordnet wird, wird ***Wert*** der Funktion in x_0 genannt oder auch Wert der Funktion an der Stelle $x = x_0$ und $f(x_0)$ geschrieben. Die Menge Y wird ***Zielmenge*** der Funktion genannt.

Bei Änderung der Argumente $x \in X$ verändern sich im Allgemeinen die Resultate $y = f(x) \in Y$ in Abhängigkeit von den Werten x. Aus diesem Grund wird die Grösse $y = f(x)$ oft auch *abhängige Variable* genannt.

Definition 1.2 (Bild einer Funktion)**:**
Die Menge

$$\text{im}\,(f) := \{\ y \in Y\ ;\ \text{es existiert ein } x \in X \text{ mit } y = f(x)\ \}$$

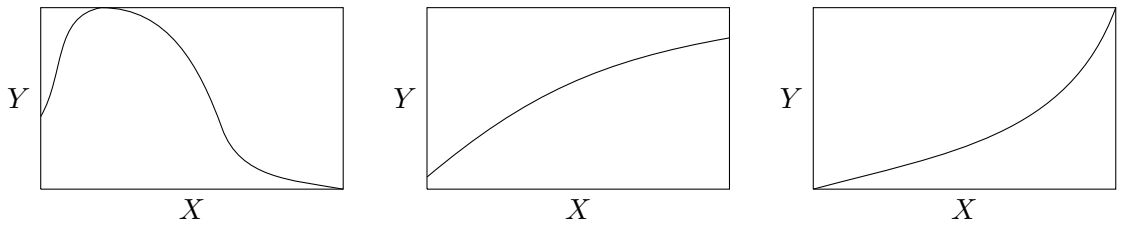
von Werten, die von einer Funktion f für alle Elemente in der Menge X angenommen werden, wird ***Bild*** (englisch: *image*) oder *Wertemenge* der Funktion $f : X \to Y$ genannt. Häufig wird $\text{im}\,(f)$ alternativ als $f(X)$ geschrieben, wobei X die Definitionsmenge von f ist.

1.1.3.2 Surjektion, Injektion, Bijektion

In diesen Unterlagen werden wir häufig über Funktionen sprechen. Insbesondere werden wir die folgenden drei Eigenschaften von Funktionen mehrmals verwenden.

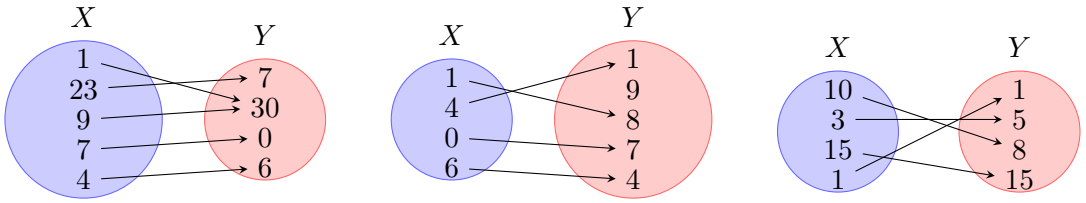
Definition 1.3 (surjektiv, injektiv, bijektiv)**:**
Seien X und Y Mengen und $f : X \to Y$ eine Funktion.

- f heisst ***surjektiv*** (eine *Surjektion*), falls $\text{im}\,(f) = Y$.
  Intuitiv gesprochen, nimmt eine surjektive Funktion jeden Wert in der Zielmenge an. Zu beliebigem $y \in Y$ existiert (mindestens) ein $x \in X$, sodass $y = f(x)$.
- f heisst ***injektiv*** (eine *Injektion*), falls für $x_1, x_2 \in X$ aus $x_1 \neq x_2$ stets $f(x_1) \neq f(x_2)$ folgt.
  Zwei verschiedene Eingaben erzeugen stets verschiedene Ausgaben.
- f heisst ***bijektiv*** (eine *Bijektion*), falls f sowohl surjektiv als auch injektiv ist.
  Da f surjektiv ist, existiert zu jedem $y \in Y$ ein $x \in X$ mit $f(x) = y$. Da f injektiv ist, kann es kein anderes $\tilde{x} \in X$ mit $\tilde{x} \neq x$ geben, sodass $f(\tilde{x}) = y$. Dadurch stellt eine Bijektion eine „Eins-zu-eins-Zuweisung“ zwischen den Elementen aus X und Y dar.



(a) surjektiv, nicht injektiv (b) injektiv, nicht surjektiv (c) bijektiv

Abbildung 1.1: schematische Darstellung der Eigenschaften in Definition 1.3



(a) surjektiv, nicht injektiv (b) injektiv, nicht surjektiv (c) bijektiv

Abbildung 1.2: Diagramm-Darstellung der Eigenschaften in Definition 1.3

Betrachten Sie Abbildung 1.1. Diese stellt die in Definition 1.3 beschriebenen Eigenschaften für

eine Funktion $f : X \to Y$ schematisch dar. Abbildung 1.1a zeigt eine Funktion, die surjektiv, aber nicht injektiv ist, Abbildung 1.1b eine Funktion, die injektiv, aber nicht surjektiv ist. Schliesslich wird in Abbildung 1.1c eine bijektive Funktion dargestellt. Abbildung 1.2 illustriert Surjektivität, Injektivität und Bijektivität für einfache Funktionen auf konkreten endlichen Mengen. Surjektivität, Injektivität und Bijektivität sind Eigenschaften, welche eine gegebene Funktion entweder haben kann oder nicht.

1.1.3.3 Folgen

Definition 1.4 (Folge)**:**
Ist X eine beliebige Menge und $f : \mathbb{N} \to X$ eine Funktion, welche auf $\mathbb{N}$ definiert ist und Werte in X annimmt, dann wird f eine ***Folge*** (in X) genannt. Ist $g : D \to X$ eine Funktion und $D \subsetneq \mathbb{N}$ eine endliche Teilmenge der natürlichen Zahlen, so wird g ***endliche Folge*** in X genannt.

Häufig schreibt man $(x_n)_{n\in\mathbb{N}}, (x_n)$ oder auch $(x_0, x_1, x_2, \dots)$ für die Folge f. Dabei bezeichnet $x_n := f(n)$ das n-te ***Glied*** der Folge $f = (x_0, x_1, x_2, \dots)$ für jedes $n \in \mathbb{N}$.

Beispiel 1.2:
Die Funktion $f : \mathbb{N} \to \mathbb{Z}$ mit $n \mapsto 2n$ ist die Folge der geraden natürlichen Zahlen.

1.2 Programmieren in Python

Wir werden hier keine Einführung in die Python-Programmiersprache geben. Wir führen im Folgenden einige Details der Python-Sprache auf, welche wir in den Übungsaufgaben der nächsten Kapitel verwenden werden.

- Der ***Modulo-Operator*** `%` (Prozent-Symbol) kann in der Form `a % b` verwendet werden und liefert den ganzzahligen Rest bei der Division von `a` mit `b`. Beispielsweise sind die folgenden Ausdrücke wahr:
  `7 % 3 == 1`, `5 % 8 == 5`, `22 % 4 == 2`, `42 % 6 == 0`.
  Wir werden den Modulo-Operator zur Überprüfung auf Teilbarkeit verwenden. Denn `b` ist offensichtlich genau dann ein Teiler von `a`, falls `a % b == 0` gilt.
- Eine Liste in Python wird durch eckige Klammern gekennzeichnet. Beispielsweise ist `[3,2,2,7]` eine Liste mit 4 Einträgen. Eine leere Liste (Liste ohne Einträge) wird durch „leere eckige Klammern“ `[]` bezeichnet. Mit dem Aufruf `L.append(x)` wird ein Eintrag `x` hinten (von rechts) in eine bereits bestehende Liste `L` eingefügt (angehängt). Mit `L[-1]` wird der letzte (am weitesten rechts stehende) Eintrag der Liste `L` bezeichnet. Der Ausdruck `L[:-1]` bezeichnet alle Einträge der Liste mit Ausnahme des letzten.
- In Python existiert die Möglichkeit, Argumenten einer Funktion ein *Default-Argument* zu geben.

```python
# Default-Argument in Python
def eineFunktion(a, b = 5):
    print(a + b)

eineFunktion(3,6) # gibt den Wert 9 aus
eineFunktion(3) # Aufruf mit Default-Argument b = 5: gibt den Wert 8 aus
```

Programm 1.1: Default-Argument

Dieses Default-Argument wird genau dann verwendet, falls kein Wert für dieses Argument (explizit) angegeben wird.

- Die Operation `math.ceil(x)` wird *Aufrundungs-Operation* genannt und gibt die kleinste ganze Zahl, die grösser oder gleich `x` ist. Beispielsweise gelten `math.ceil(3.2) = 4` und `math.ceil(3) = 3`. Um diese Operation verwenden zu können, muss die Bibliothek `math` durch den Befehl `import math` vor dem Gebrauch dieser Operation ins Programm eingefügt werden.

In Listing 1.2 haben wir die obigen Punkte noch einmal zusammengefasst.

```
# gibt die durch 19 teilbaren Zahlen in {0,1,2,...,99} aus:
for k in range(100):
    if (k % 19) == 0:
        print(k) # gibt aus: 0 19 38 57 76 95

# Listen
L = [3, 7, 2, 3]
L.append(99)  # L ist nun [3, 7, 2, 3, 99].
print(L[-1])  # gibt aus: 99
print(L[:-2]) # gibt aus: 3, 7, 2

# Default-Argument
def person(vorname, nachname = 'Nachname unbekannt', alter = '-'):
    print('Vorname: ', vorname, ', ' , 'Nachname: ', nachname, ', ', 'Alter: ',
     alter, sep='')

person('Sandra') # gibt aus: Vorname: Sandra, Nachname: Nachname unbekannt,
    Alter: -

person('Toni', 'Wildeisen') # gibt aus: Vorname: Toni, Nachname: Wildeisen,
    Alter: -

# Importieren einer Bibliothek und Verwendung der Aufrundungs-Funktion
import math # Einfügen der Bibliothek 'math'
print(math.ceil(3.00002)) # gibt aus: 4
```

Programm 1.2: Vorwissen Python

Kapitel 2

Die natürlichen Zahlen

2.1 Historische Betrachtung der natürlichen Zahlen

Die *natürlichen* Zahlen werden so genannt, da sie auf „natürliche Weise“ beim Zählen verwendet werden. Auf dem Gebiet der heutigen Demokratischen Republik Kongo wurde im 20. Jahrhundert ein Knochen, der sogenannte *Ishango-Knochen*, gefunden. Dieser Knochen wird auf die Zeit vor etwa 18’000 bis 20’000 Jahren vor unserer Zeitrechnung datiert. In dem Knochen sind ganz offensichtlich von Menschen einige Kerben eingeritzt worden. Die Bedeutung dieser Kerben ist nicht klar. Es wird jedoch angenommen, dass diese Kerben eine bestimmte Anzahl festhalten.

Stellen wir uns vor, dass die Kerben die Anzahl der gefangenen Fische (🐟) an einem bestimmten Tag darstellen. Gefangene Fische durch Kerben in einem Knochen zu repräsentieren, verlangt bereits einen recht hohen Grad an Abstraktion! Schliesslich haben gefangene Fische und Kerben in einem Knochen auf den ersten Blick keinen offensichtlichen Zusammenhang. Anstelle von Kerben in einem Knochen könnte die Fangmenge eines Tages auch durch die entsprechende Anzahl von Kieselsteinen oder durch (abstraktere) römische Numerale repräsentiert werden. Wichtig für uns ist, dass die konkrete Wahl der Darstellung, zumindest rein mathematisch betrachtet, unwichtig ist. Schliesslich stellen die verschiedenen Darstellungen alle dieselbe Anzahl von Fischen dar. Betrachten Sie Tabelle 2.1. Mit dem Symbol 🐟 meinen wir nicht ein Fisch-Emoji, sondern den eigentlich gefangenen Fisch. Die direkte Darstellung der Anzahl der gefangenen Fische durch sich selbst ist offensichtlich am wenigsten abstrakt. Die Darstellung durch Kerben oder Kieselsteine bedarf bereits einer nicht unerheblichen Abstraktion. Dhttps://www.youtube.com/watch?v=c8l7K67idZcie Darstellungen durch das römische, dezimale und binäre Zahlensysteme sind noch eine Stufe abstrakter.

Fische	Kieselsteine	römisch	binär	dezimal
(keine)	(keine)	nichts (nihil)	0	0
🐟	•	I	1	1
🐟🐟	• •	II	10	2
🐟🐟🐟	• • •	III	11	3
🐟🐟🐟🐟	• • • •	IV	100	4
🐟🐟🐟🐟🐟	• • • • •	V	101	5
🐟🐟🐟🐟🐟🐟	• • • • • •	VI	110	6
🐟🐟🐟🐟🐟🐟🐟	• • • • • • •	VII	111	7
🐟🐟🐟🐟🐟🐟🐟🐟	• • • • • • • •	VIII	1000	8
🐟🐟🐟🐟🐟🐟🐟🐟🐟	• • • • • • • • •	IX	1001	9

Tabelle 2.1: mögliche Darstellungen der Anzahl gefangener Fische

Die Zahl *Null* hat eine lange und kontroverse Geschichte hinter sich. Vermutlich hatten die Römer kein explizites eigenes Symbol für die Null. Wir werden jedoch die Null als die erste (kleinste) natürliche Zahl auffassen.

2.2 Unendlichkeit der natürlichen Zahlen und konstruktive Induktion

Bereits aufgrund der kurzen Anekdote über die gefangenen Fische und Kieselsteine in Abschnitt 2.1 kann man sich denken, dass es unendlich viele natürliche Zahlen geben muss. Wieder denken wir uns eine natürliche Zahl als genau das Symbol, welches eine bestimme Anzahl an Kieselsteinen darstellt. Dann ist klar, dass es keine grösste natürliche Zahl geben kann, denn schliesslich kann stets ein weiterer Kieselstein hinzugefügt werden, was uns eine noch grössere Zahl liefert. Zu jeder natürlichen Zahl n, angefangen mit der 0, erhalten wir (durch Hinzufügen eines Kieselsteins) eine weitere natürliche Zahl. Diese neue Zahl werden wir den *Nachfolger* von n nennen. Die folgenden Überlegungen dieses Kapitels formalisieren diese Vorstellungen und stellen diese mathematisch präzise dar. Doch auch ohne mathematische Formeln können wir das folgende Gedankenexperiment durchführen. Angenommen Sie sind in der Lage Folgendes zu tun:

- Sie können eine Treppe mit 0 Stufen (ohne Stufen) bauen.
- Falls Sie bereits eine Treppe mit einer beliebigen (natürlichen) Anzahl von Stufen gebaut haben, dann können Sie diese Treppe um eine Stufe erweitern (siehe Abbildung 2.1).

Dann werden Sie es für glaubhaft halten, dass Sie eine Treppe mit beliebig vielen Stufen bauen können. Wie dieses Gedankenexperiment mit der mathematischen Definition der natürlichen Zahlen zusammenhängt, werden Sie im nächsten Abschnitt sehen.

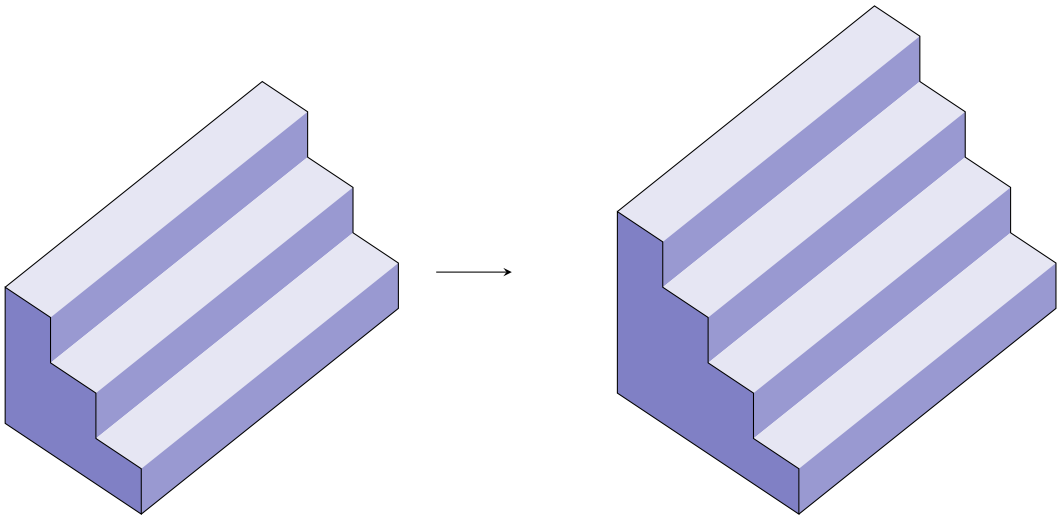


Abbildung 2.1: Erweiterung einer Treppe um eine weitere Stufe.

2.3 Die Peano-Axiome

Uns ist vollkommen bewusst, dass Sie bereits seit Ihrer Kindheit mit dem Konzept der natürlichen Zahlen vertraut sind. Sie konnten vermutlich schon als Kleinkind Gegenstände zählen und lernten spätestens in der Grundschule die Addition und Multiplikation natürlicher Zahlen kennen. Für diese einfachen Anwendungen reicht eine rein intuitive Beschreibung der natürlichen Zahlen vollkommen aus.

Sie werden jedoch festgestellt haben, dass mit fortschreitender schulischer Reife eine präzise Definition von Begriffen und Konzepten zunehmend an Bedeutung gewinnt. Auf der Stufe des Gymnasiums wird der bis dorthin rein intuitiv geprägte Begriff der natürlichen Zahlen durch das Konzept der *Mengen* formalisiert. Ab dann wird von der *Menge der natürlichen Zahlen*, bezeichnet durch das Symbol $\mathbb{N}$, gesprochen.

2.4 Informale Definition der natürlichen Zahlen

In Lehrbüchern des Gymnasiums (siehe zum Beispiel [1][1]) wird die Menge der natürlichen Zahlen typischerweise wie folgt definiert:

Definition 2.1 (Informale Definition der natürlichen Zahlen)**:**
Die Menge

$$\mathbb{N} := \{0, 1, 2, 3, 4, \ldots\}$$

wird als die Menge der **natürlichen Zahlen** bezeichnet.

Man beginnt also bei 0 und zählt dann unbegrenzt weit nach vorne. In einem gewissen Sinne beantwortet Definition 2.1 die Frage, was natürliche Zahlen sind: Eine natürliche Zahl ist ein Element der Menge $\mathbb{N}$. Dennoch ist die Definition nicht sehr befriedigend, denn sie beantwortet nicht die Frage, was $\mathbb{N}$ selbst ist. [2] Wir werden nicht von dieser Definition Gebrauch machen, sondern eine nützlichere Definition entwickeln.

Der folgende Abschnitt ist teilweise inspiriert durch die entsprechenden Teile in den hervorragenden Büchern [2] und [3].

Bei näherer Betrachtung wirft die informale Definition 2.1 insbesondere die folgenden drei Fragen auf:

1. Woher wissen wir, dass wir beliebig lange weiter vorwärts zählen können, ohne schliesslich (wie bei einer Uhr) wieder bei der 0 anzukommen?
2. Wie sollen nun Operationen wie die Addition, Multiplikation und die Potenz definiert werden?

Die zweite Frage wollen wir zuerst besprechen. Komplizierte Operationen können durch einfachere Operationen ausgedrückt werden. So ist Potenzieren lediglich wiederholtes Multiplizieren und Multiplizieren wiederum wiederholtes Addieren. Zum Beispiel sind 5^3 nichts weiter als drei Fünfer miteinander multipliziert und $6 \cdot 3$ lediglich sechs Dreien miteinander addiert. Wie sieht es mit der Addition aus? Die Addition kann durch wiederholtes *Inkrementieren* oder *Vorwärtszählen* realisiert werden. Bei der Addition $4 + 3$ wird die Vier dreimal inkrementiert (wir zählen von der Vier aus dreimal vorwärts). Inkrementieren scheint eine fundamentale Operation zu sein, welche sich nicht auf eine noch einfachere Operation reduzieren lässt.

Eine sinnvolle Definition der natürlichen Zahlen scheint also das Inkrementieren als fundamentales Konzept zu verwenden. Für eine natürliche Zahl n werden wir im Folgenden mit $\nu(n)$ das Inkrement von n bezeichnen. Wir werden $\nu(n)$ auch den ***Nachfolger*** von n nennen. Zum Beispiel gilt $3 = \nu(2), 4 = \nu(3) = \nu(\nu(2))$ und so weiter. Das Inkrementieren liefert uns also einen „Zählvorgang“, der bei 0 beginnt. Die bisherigen Überlegungen lassen vermuten, dass wir $\mathbb{N}$ als die Menge mit den Elementen

$$0, \nu(0), \nu(\nu(0)), \nu(\nu(\nu(0))), \nu(\nu(\nu(\nu(0)))), \nu(\nu(\nu(\nu(\nu(0))))), \ldots$$

[1] Dieses Buch bietet allerdings auch eine alternative Definition der natürlichen Zahlen an.

ansehen wollen. Diese Menge enthält 0 und alle Objekte, welche aus 0 durch Inkrementieren erhalten werden können. Sie wissen bereits, dass fundamentale (nicht beweisbare) Annahmen in der Mathematik als *Axiome* bezeichnet werden. Bislang haben wir zwei fundamentale Annahmen bezüglich der Menge $\mathbb{N}$ der natürlichen Zahlen getroffen. Diese fassen wir in zwei Axiomen zusammen:

Axiom 2.1:
Die 0 liegt in $\mathbb{N}$.

Axiom 2.2:
Falls n in $\mathbb{N}$ liegt, dann liegt auch der Nachfolger $\nu(n)$ von n in der Menge $\mathbb{N}$.

Dies sind die ersten zwei von insgesamt fünf Axiomen, welche zusammen bekannt sind als die ***Peano-Axiome*** der natürlichen Zahlen. Die Peano-Axiome sind benannt nach dem italienischen Mathematiker *Giuseppe Peano*, welcher diese Axiome im Jahr 1889 formulierte.

Bemerkung 2.1:

- Beachten Sie, dass Axiom 2.2 lediglich aussagt, dass der Nachfolger $\nu(n)$ einer natürlichen Zahl n wieder eine natürliche Zahl ist. Das Axiom sagt nichts darüber aus, wie dieser Nachfolger lautet.
- Manche Autorinnen und Autoren ziehen es vor, den „Zählvorgang“ nicht bei 0, sondern bei 1 zu beginnen. Dies ist mathematisch ohne Bedeutung. [3]
- Wir definieren $1 := \nu(0)$, $2 := \nu(1) = \nu(\nu(0))$, $3 := \nu(2) = \nu(\nu(\nu(0)))$ und so weiter. Anstelle von

  $$0, \nu(0), \nu(\nu(0)), \nu(\nu(\nu(0))), \nu(\nu(\nu(\nu(0)))), \nu(\nu(\nu(\nu(\nu(0))))), \ldots$$

  schreiben wir üblicherweise $0, 1, 2, 3, 4, 5, \ldots$

In Abschnitt 2.1 haben wir bereits begründet, warum die konkrete Darstellung einer Zahl nicht von Bedeutung ist. Wir haben Tabelle 2.1 um eine Spalte erweitert:

Fische	**Kieselsteine**	**römisch**	**binär**	**dezimal**	**Nachfolger**
(keine)	(keine)	nichts (nihil)	0	0	0
🐟	•	I	1	1	$\nu(0)$
🐟🐟	• •	II	10	2	$\nu(\nu(0))$
🐟🐟🐟	• • •	III	11	3	$\nu(\nu(\nu(0)))$
🐟🐟🐟🐟	• • • •	IV	100	4	$\nu(\nu(\nu(\nu(0))))$
🐟🐟🐟🐟🐟	• • • • •	V	101	5	$\nu(\nu(\nu(\nu(\nu(0)))))$
🐟🐟🐟🐟🐟🐟	• • • • • •	VI	110	6	$\nu(\nu(\nu(\nu(\nu(\nu(0))))))$
🐟🐟🐟🐟🐟🐟🐟	• • • • • • •	VII	111	7	$\nu(\nu(\nu(\nu(\nu(\nu(\nu(0)))))))$
🐟🐟🐟🐟🐟🐟🐟🐟	• • • • • • • •	VIII	1000	8	$\nu(\nu(\nu(\nu(\nu(\nu(\nu(\nu(0))))))))$
🐟🐟🐟🐟🐟🐟🐟🐟🐟	• • • • • • • • •	IX	1001	9	$\nu(\nu(\nu(\nu(\nu(\nu(\nu(\nu(\nu(0)))))))))$

Tabelle 2.2: Illustration des Nachfolgers einer natürlichen Zahl. Beachten Sie, dass 0 nicht Nachfolger einer anderen natürlichen Zahl ist.

Abbildung 2.2: Giuseppe Peano (1858-1932)

Aufgabe 2.1

Beweisen Sie, dass 2 eine natürliche Zahl ist. Verwenden Sie dazu lediglich die beiden ersten Peano-Axiome.

Lösungsvorschlag zu Aufgabe 2.1

Gemäss Axiom 2.1 ist 0 eine natürliche Zahl. Gemäss Axiom 2.2 ist $1 = \nu(0)$ eine natürliche Zahl. Schliesslich ist $2 = \nu(1)$ wiederum gemäss Axiom 2.2 eine natürliche Zahl.

Bereits Giuseppe Peano stellte fest, dass diese ersten beiden Axiome nicht ausreichend sind, um unsere intuitive Vorstellung der natürlichen Zahlen einzufangen. Es könnte sein, dass wir bei dem Vorwärtszählen schliesslich wieder bei 0 ankommen. Betrachten wir dazu ein Modell der natürlichen Zahlen, welches von 3 zurück zur 0 zählt, genauer: $\nu(0)$ ist 1, $\nu(1)$ ist 2, $\nu(2)$ ist 3, aber $\nu(3)$ ist wieder 0 (und gemäss der Definition von 4 auch gleich der 4). So erfüllt also auch die Menge $\{0, 1, 2, 3\}$ die ersten zwei Peano-Axiome und könnte als Menge der natürlichen Zahlen angesehen werden.

Aufgabe 2.2

Welches ist die kleinste Menge, welche die ersten zwei Peano-Axiome erfüllt?

Lösungsvorschlag zu Aufgabe 2.2

Bereits die Menge $\{0\}$ erfüllt die ersten beiden Peano-Axiome. Sie enthält offensichtlich die 0 und mit $\nu(0) := 0$ auch den Nachfolger von jedem $n \in \{0\}$ (es gibt nur das eine $n \in \{0\}$ und das ist die 0).

Die Axiome 2.1 und 2.2 erlauben also auch Mengen, welche wir sicherlich nicht als Modelle der natürlichen Zahlen anschauen möchten. Wir müssen die erlaubten Nachfolger der Elemente der Mengen weiter einschränken. Auf jeden Fall stellen wir fest, dass die 0 nicht Nachfolger einer natürlichen Zahl sein soll und fordern deshalb:

Axiom 2.3:
Die 0 selbst ist nicht Nachfolger einer natürlichen Zahl. Es gilt also $\nu(n) \neq 0$ für jede natürliche Zahl n.

Bemerkung 2.2:
Wir bezeichnen mit $\mathbb{N}^\times$ die Menge der natürlichen Zahlen ohne die 0. Für jede natürliche Zahl n soll der Nachfolger $\nu(n)$ von n wieder eine natürliche Zahl sein. Da gemäss Axiom 2.3 die 0 nicht Nachfolger einer natürlichen Zahl ist, können wir uns ν als Funktion

$$\begin{aligned} \nu : \mathbb{N} &\to \mathbb{N}^\times, \\ n &\mapsto \nu(n) \end{aligned}$$

denken. Diese erhält eine natürliche Zahl n als Eingabe und liefert die natürliche Zahl $\nu(n)$, welche nicht die 0 ist, als Ausgabe.

Aufgabe 2.3

Beweisen Sie, dass $0 \neq 3$ gilt. Verwenden Sie lediglich die ersten drei Peano-Axiome.

Lösungsvorschlag zu Aufgabe 2.3

In Aufgabe 2.1 haben wir gezeigt, wie aus den ersten zwei Peano-Axiomen folgt, dass 2 eine natürliche Zahl ist. Definitionsgemäss ist $3 = \nu(2)$. Somit ist 3 der Nachfolger einer natürlichen Zahl und deshalb muss 3 gemäss dem dritten Peano-Axiom von 0 verschieden sein, also $0 \neq 3$.

Betrachten Sie ein Zahlensystem, welches 0 enthält und für das gilt: $\nu(0)$ ist 1, $\nu(1)$ ist 2, $\nu(2)$ ist 3, aber $\nu(3)$ bleibt 3 (also $4 = 3$, $5 = 3$, $6 = 3$ und so weiter). Es ist auch ein Zahlensystem vorstellbar, welches von 3 zurück zu 1 geht, also $\nu(3) = 1, \nu(1) = 2, \nu(2) = 3, \nu(3) = 1$ und so weiter. Die beiden Beispiele erfüllen alle drei ersten Peano-Axiome. Das Problem ist, dass die ersten drei Peano-Axiome erlauben, dass verschiedene natürliche Zahlen gleiche Nachfolger haben können. Diese Möglichkeit wollen wir also ausschliessen. Dazu fügen wir das vierte Peano-Axiom hinzu:

Axiom 2.4:
Unterschiedliche natürliche Zahlen haben unterschiedliche Nachfolger. Sind also n, m natürliche Zahlen und $n \neq m$, dann gilt $\nu(n) \neq \nu(m)$. Die Kontraposition dieser Aussage lautet: Gilt $\nu(n) = \nu(m)$, dann folgt $n = m$.

Aufgabe 2.4

Verwenden Sie die ersten vier Peano-Axiome, um zu beweisen, dass $1 \neq 4$ gilt.

Lösungsvorschlag zu Aufgabe 2.4

Wir beweisen die Aussage durch Widerspruch und nehmen also an, dass $1 = 4$. Gleichzeitig gilt aber $1 = \nu(0)$ und $4 = \nu(3)$ und somit $\nu(0) = \nu(3)$. Wegen des vierten Peano-Axioms folgt also $0 = 3$. Doch in Aufgabe 2.3 haben wir gezeigt, dass $0 \neq 3$ gilt. Wir haben also einen Widerspruch gefunden und somit muss $1 \neq 4$ gelten.

Wir wollen $\mathbb{N}$ als die Menge verstehen, welche die 0 enthält und alle Objekte, welche aus 0 durch Inkrementieren erhalten werden kann. Diese Intuition wird schliesslich auf geniale Weise durch das fünfte Peano-Axiom formalisiert. Dieses letzte Peano-Axiom formuliert auf mathematisch präzise Weise, was mit „aus 0 durch Inkrementieren erhalten werden kann“, gemeint ist:

Axiom 2.5:
Enthält eine Teilmenge $N \subseteq \mathbb{N}$ das Element 0 und mit jedem $n \in N$ auch den Nachfolger $\nu(n)$ von n, so gilt $N = \mathbb{N}$.

2.4.1 Kompakte Formulierung der Peano-Axiome

Unser Wissen über Funktionen und ihre Eigenschaften erlaubt uns, die fünf Axiome 2.1 bis 2.5, kompakt und sehr präzise in der folgenden Definition zusammenzufassen:

Definition 2.2 (Formale Definition der natürlichen Zahlen)**:**
Die **natürlichen Zahlen** sind eine Menge $\mathbb{N}$, in der ein Element $0 \in \mathbb{N}$ ausgezeichnet ist und für die es eine Funktion $\nu : \mathbb{N} \to \mathbb{N}^\times$ (siehe Bemerkung 2.2) mit den folgenden zwei Eigenschaften gibt:
(N_0) Die Funktion ν ist injektiv.
(N_1) Enthält eine Teilmenge $N \subseteq \mathbb{N}$ das Element 0 und mit jedem $n \in N$ auch den Nachfolger $\nu(n)$ von n, so gilt $N = \mathbb{N}$.

Dabei bezeichnet $\mathbb{N}^\times := \mathbb{N} \setminus \{0\}$ die Menge der natürlichen Zahlen ohne die 0. Das Element $\nu(n)$ heisst Nachfolger von n und ν heisst Nachfolgerfunktion. Die Eigenschaft (N_1) ist identisch zum Axiom 2.5 und wird auch ***Induktionsaxiom*** genannt.

Aufgabe 2.5

Weisen Sie nach, dass Definition 2.2 zu den Peano-Axiomen äquivalent ist.

Lösungsvorschlag zu Aufgabe 2.5

Wir zeigen zuerst, dass Definition 2.2 alle fünf Peano-Axiome „abdeckt“.

- Axiom 2.1: Dieses Axiom ist erfüllt, da in der Definition gefordert wird, dass ein Element 0 in $\mathbb{N}$ ausgezeichnet ist.
- Axiom 2.2: Dieses Axiom ergibt sich dadurch, dass $\nu : \mathbb{N} \to \mathbb{N}^\times$ eine Funktion von den natürlichen Zahlen in eine Teilmenge der natürlichen Zahlen ist. Doch die Elemente einer Teilmenge der natürlichen Zahlen sind offensichtlich ebenfalls natürliche Zahlen.
- Axiom 2.3: Wegen $\nu : \mathbb{N} \to \mathbb{N}^\times$ liegt die 0 nicht im Bild der Funktion ν. Somit ist also die 0 nicht Nachfolger einer natürlichen Zahl und damit ist auch dieses Axiom erfüllt.
- Axiom 2.4: Dieses Axiom sagt genau, dass die Funktion ν injektiv ist und entspricht somit exakt der Eigenschaft (N_0).
- Axiom 2.5: Dieses Axiom entspricht exakt der Eigenschaft (N_1).

Da Definition 2.2 keine zusätzlichen Forderungen stellt, ist diese Definition tatsächlich äquivalent zu den Peano-Axiomen.

Aufgabe (Challenge) 2.6

Erfüllt auch die Menge $\{0, 2, 4, 6, \ldots\}$ der geraden natürlichen Zahlen die Peano-Axiome?

✔ Lösungsvorschlag zu Aufgabe 2.6

Die Frage der Aufgabenstellung geht an der eigentlichen Aussage der Peano-Axiome vorbei und ist im Grunde bedeutungslos. Die Peano-Axiome verlangen nicht, dass die natürlichen Zahlen genau die Form

$$\{0, 1, 2, 3, 4, \ldots\}$$

haben. Die konkreten Schriftsymbole, mit denen die natürlichen Zahlen bezeichnet werden, werden von den Peano-Axiomen nicht vorgeschrieben. In der üblichen westlichen Schreibweise wird der Nachfolger der 0 mit dem Symbol 1 bezeichnet, der Nachfolger von 1 mit 2 und so weiter. Anstelle von der Konvention

$$1 := \nu(0), 2 := \nu(1) = \nu(\nu(0)), 3 := \nu(2) = \nu(\nu(\nu(0))), \ldots$$

könnten wir den Nachfolger der 0 auch mit 2 bezeichnen und den Nachfolger der 2 mit 4 und so weiter. Offensichtlich ist auch das exakte Symbol der Null bedeutungslos. Anstelle von dem Symbol 0 könnten wir auch das Symbol 🧸 für dieses ausgezeichnete Element verwenden. Es ist nicht einfach, sich auf diese abstrakte Ebene zu begeben. Sie müssen sich von den konkret verwendeten Schriftsymbolen lösen und sich stattdessen auf die Bedeutung der verwendeten Symbole fokussieren.

🏆 Aufgabe (Challenge) 2.7

Begründen Sie, dass die Nachfolgerfunktion $\nu : \mathbb{N} \to \mathbb{N}^{\times}$ bijektiv ist.

Tipp: Verwenden Sie die beiden Eigenschaften N_0 und N_1 in Definition 2.2.

✔ Lösungsvorschlag zu Aufgabe 2.7

Wie können wir beweisen, dass ν bijektiv ist? Wir dürfen für den Nachweis lediglich bereits bekannte Tatsachen sowie die Peano-Axiome in Definition 2.2 verwenden. Sicherlich können wir zunächst einmal feststellen, dass die Funktion ν wegen der Eigenschaft N_0 als injektiv vorausgesetzt ist. Damit ist ν also injektiv. Für den Nachweis der Bijektivität von ν müssen wir nur noch seine Surjektivität zeigen.

Möchte man eine Eigenschaft nachweisen, ist es zunächst einmal sinnvoll, aufzuschreiben, was diese Eigenschaft genau bedeutet. Definitionsgemäss ist $\nu : \mathbb{N} \to \mathbb{N}^\times$ genau dann surjektiv, wenn das Bild $\mathrm{im}\,(\nu)$ der Zielmenge $\mathbb{N}^\times$ entspricht, falls also

$$\mathrm{im}\,(\nu) = \mathbb{N}^\times$$

gilt. Nun haben wir Eigenschaft N_0 in Definition 2.2 bereits ausgenutzt. Es ist daher naheliegend, Eigenschaft N_1 zu betrachten. Da N_1 eine Aussage über $\mathbb{N}$ und nicht etwa über $\mathbb{N}^\times$ macht, tun wir uns selbst einen Gefallen, wenn wir anstelle von $\mathbb{N}^\times$ die um das Element 0 erweiterte „Hilfsmenge“

$$N := \mathrm{im}\,(\nu) \cup \{0\}$$

betrachten. Wir zeigen nun, dass diese Hilfsmenge N eine Teilmenge von $\mathbb{N}$ ist, welche die Bedingungen in N_1 erfüllt. Offensichtlich gelten $N \subseteq \mathbb{N}$ und $0 \in N$. Für $n \in N$ gilt $\nu(n) \in \mathrm{im}\,(\nu) \subseteq N$. Somit liegt für $n \in N$ auch der Nachfolger $\nu(n)$ in der Menge N und damit folgt aus N_1, dass $N = \mathbb{N}$ gilt. Wir haben also die Gleichheit

$$\mathrm{im}\,(\nu) \cup \{0\} = \mathbb{N}$$

gezeigt und folgern daraus

$$\mathrm{im}\,(\nu) = \mathbb{N}^\times.$$

Bemerkung 2.3:

(a) Aus den grundlegenden Axiomen der Mengenlehre folgt, dass es in der Tat Systeme $(\mathbb{N}, 0, \nu)$ gibt, welche die Peano-Axiome erfüllen. Diese Modelle der natürlichen Zahlen sind bis auf die Benennung der Elemente gleichwertig und ergeben dieselbe Mathematik. Beispielsweise könnte man anstelle der arabischen Zahlenschrift auch die römische Zahlenschrift verwenden. Die konkrete Wahl der Symbole ist mathematisch nicht von Bedeutung. Deshalb ist es sinnvoll, von *den* natürlichen Zahlen zu sprechen.

(b) Die aus der Schule bekannten Rechenregeln in den natürlichen Zahlen (zum Beispiel das Distributivgesetz) lassen sich allein durch logische Folgerungen aus den Peano-Axiomen beweisen. Diese Beweise werden zum Beispiel in dem Buch [4] geführt, welches im Jahr 1930 erschien. Im Vorwort dieses Buchs schreibt der Autor *Edmund Landau* unter anderem:
- „Ich setze nur logisches Denken und die deutsche Sprache als bekannt voraus; nichts aus der Schulmathematik oder gar der höheren Mathematik."
- „Bitte vergiß alles, was Du auf der Schule gelernt hast; denn Du hast es nicht gelernt."

Wir werden diese (recht umfangreichen) Beweise hier nicht führen. Besonders interessierten Lesenden empfehlen wir in diesem Zusammenhang die entsprechenden Teile in den Büchern [4, 2, 3] zu studieren.

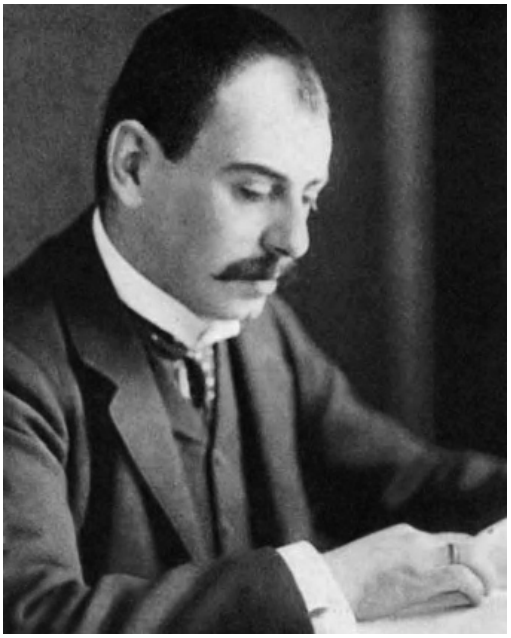


Abbildung 2.3: Edmund Landau (1877-1938)

Kapitel 3

Das Induktionsprinzip

Axiom 2.5 wird ***Prinzip der vollständigen Induktion*** genannt. Dieses Kapitel wird sich mit diesem wichtigen Prinzip befassen. Sie werden sehen, welche enorm weitreichenden Konsequenzen das Axiom 2.5 (Induktionsaxiom) der natürlichen Zahlen hat. Die Notation dieses Kapitels sowie einige Beweise stammen aus Kapitel 5 in [3].

3.1 Einführende Beispiele

3.1.1 Teilbarkeit durch 2

Ihre gute Freundin Anika behauptet, eine mathematische Entdeckung gemacht zu haben. Sie hat nämlich bemerkt, dass, wenn zu dem Quadrat n^2 einer natürlichen Zahl n die Zahl n addiert wird, die entstandene Summe stets gerade ist. Sie behauptet also, dass für jede natürliche Zahl n die Zahl $n^2 + n$ gerade ist.

Aufgabe 3.1

Anika hat in der Vergangenheit schon öfters mathematische Behauptungen aufgestellt. Nicht selten haben sich diese bei genauerer Untersuchung als falsch erwiesen. Bevor Sie also viel Zeit in die genauere Analyse Anikas neuer Behauptung investieren, möchten Sie eine kurze „Plausibilitätsprüfung“ durchführen. Schreiben Sie dazu ein Python-Programm, welches für die ersten 100 natürlichen Zahlen $n \in \{\, m \in \mathbb{N} \,;\, 0 \leq m < 100 \,\}$ überprüft, ob $n^2 + n$ jeweils gerade ist.

Das folgende Beispiel zeigt in aller Ausführlichkeit, wie das Prinzip der vollständigen Induktion verwendet werden kann, um eine mathematische Vermutung zu beweisen.

Beispiel 3.1:
Wir betrachten erneut die Vermutung Ihrer Freundin Anika:

Für jedes $n \in \mathbb{N}$ ist die Zahl $n^2 + n$ gerade.

Wie lässt sich eine solche Vermutung überprüfen? Sicherlich können wir die Vermutung für einige natürliche Zahlen „von Hand“ durch simples „Nachrechnen“ überprüfen. In Aufgabe 3.1 haben Sie die Vermutung mit einem Computerprogramm für die ersten 100 natürlichen Zahlen überprüft. Das Problem ist jedoch, dass wir (selbst unter Verwendung von Supercomputern) immer nur endlich viele Zahlen überprüfen können. Wenn wir die Vermutung für 100 Milliarden Zahlen geprüft haben, bleiben immer noch unendlich viele Zahlen, die wir noch nicht

betrachtet haben. Was wir benötigen, ist ein mathematisches Argument, welches grundsätzlich erklärt, warum die Vermutung stimmen muss. Ein solches mathematisches Argument liefert uns das Prinzip der vollständigen Induktion. Um dieses direkt anwenden zu können, definieren wir die „Hilfsmenge“

$$N := \{\, n \in \mathbb{N} \,;\, n^2 + n \text{ ist gerade} \,\}.$$

Die so definierte Menge N enthält genau die natürlichen Zahlen, für welche die Zahl $n^2 + n$ gerade ist, für welche also die Vermutung gilt. Somit ist die Vermutung bewiesen, wenn wir nachweisen können, dass alle natürlichen Zahlen in N liegen, dass also die Gleichheit $N = \mathbb{N}$ gilt. Es gilt somit:

$$(\text{für jedes } n \in \mathbb{N} \text{ ist die Zahl } n^2 + n \text{ gerade}) \quad \Longleftrightarrow \quad (N = \mathbb{N}).$$

Nun besagt das Prinzip der vollständigen Induktion (Axiom 2.5), dass für eine Teilmenge $N \subseteq \mathbb{N}$ tatsächlich $N = \mathbb{N}$ gilt, falls zwei Bedingungen erfüllt sind:

1. $0 \in N$,
2. ist $n \in N$, dann enthält N auch den Nachfolger $\nu(n) = n + 1$.

Wir prüfen diese beiden Bedingungen separat:

1. In der Tat gilt $0 \in N$, denn $0^2 + 0 = 0$ und 0 ist gerade. ✓
2. Sei $n \in N$ und somit $n^2 + n$ eine gerade Zahl. Wir müssen beweisen, dass auch $\nu(n) = (n+1) \in N$ gilt, dass also auch $(n+1)^2 + (n+1)$ eine gerade Zahl ist. Dazu betrachten wir die folgende Berechnung:

$$(n+1)^2 + (n+1) = n^2 + 2n + 1 + (n+1) = n^2 + 3n + 2 =$$
$$n^2 + n + (2n+2) = n^2 + n + 2(n+1).$$

   Betrachten wir den Ausdruck $n^2 + n + 2(n+1)$. Wir wissen, dass die Zahl $n^2 + n$ gerade ist (da n in der Menge N liegt). Die Zahl $2(n+1)$ ist offensichtlich ebenfalls gerade. Somit ist $(n+1)^2 + (n+1)$ als Summe zweier gerader Zahlen ebenfalls gerade. (Da die Zahl $n^2 + n$ gerade ist, existiert eine natürliche Zahl k, sodass $n^2 + n = 2k$. Dann ist die Summe $(n+1)^2 + (n+1) = 2k + 2(n+1) = 2(k+n+1)$ das Doppelte der Zahl $k + n + 1$ und somit gerade). ✓

Damit sind die Bedingungen des Prinzips der vollständigen Induktion erfüllt und die Gleichheit $N = \mathbb{N}$ (und dadurch die ursprüngliche Vermutung Anikas) bewiesen.

Aufgabe 3.2

Verwenden Sie das Prinzip der vollständigen Induktion, um zu beweisen, dass $n^5 - n$ für jede natürliche Zahl n durch 5 teilbar ist.

3.1.2 Klingende Gläser

In dem Gasthaus *Inn of the Prancing Pony* stossen n Gäste auf das neue Jahr an. Jede Person stösst mit jeder anderen (nicht mit sich selbst) genau einmal an. Wie viele Male klingen die Gläser (wie viele Male wird angestossen)? Die folgende elegante Überlegung gibt uns eine Formel, um die gesuchte Zahl rasch zu berechnen. Jede der n Personen stösst offensichtlich mit genau $(n-1)$ Personen an (mit allen anderen). Damit klingen die Gläser also $n(n-1)$-mal. Die Formel ist so aber noch nicht richtig, denn wir haben jedes Klingen doppelt gezählt (anstatt nur einfach). Stösst nämlich Person A mit Person B an, dann haben wir dieses Anstossen einmal aus Sicht von A gezählt und noch einmal aus Sicht von B. Somit müssen wir die gesuchte Zahl $n(n-1)$ noch durch den Faktor 2 teilen. Die gesuchte Zahl ist also $n(n-1)/2$.

Beispiel 3.2:
Diese Formel wollen wir nun mithilfe des Prinzips der vollständigen Induktion überprüfen. Dazu definieren wir, wie bereits in Beispiel 3.1, die Behauptung in geeigneter Form. Für jedes $n \in \mathbb{N}$ definieren wir also die Aussage $\mathcal{A}(n)$ als:

$\mathcal{A}(n) : \Longleftrightarrow$

Stossen n Personen (alle mit allen) an, klingen die Gläser genau $\dfrac{n(n-1)}{2}$ Male.

Um das Prinzip der vollständigen Induktion direkt anwenden zu können, definieren wir erneut eine „Hilfsmenge“ N der Form

$$N := \{\, n \in \mathbb{N} \,;\, \mathcal{A}(n) \text{ ist wahr} \,\}.$$

Somit ist die Vermutung bewiesen, wenn wir nachweisen können, dass alle natürlichen Zahlen in N liegen, dass also die Gleichheit $N = \mathbb{N}$ gilt. Erneut prüfen wir die beiden Bedingungen des Induktionsaxioms separat:

1. Die behauptete Formel besagt, dass $n = 0$ Personen genau $0 \cdot (0-1) = 0$-mal anstossen. Diese ist offensichtlich korrekt und somit haben wir $0 \in N$ begründet. ✓
2. Sei $n \in N$ und somit $\mathcal{A}(n)$ wahr. Wir müssen beweisen, dass auch $\nu(n) = (n+1) \in N$ gilt, dass also bei $(n+1)$ Personen genau $(n+1)n/2$ Male die Gläser klingen.
   Dazu stellen wir uns vor, dass erst n Personen im Restaurant sind und diese bereits alle miteinander angestossen haben. Da $\mathcal{A}(n)$ wahr ist, wissen wir also, dass die Gläser bereits genau $n(n-1)/2$ Male klangen. Nun kommt eine weitere Person hinzu. Diese Person stösst mit allen n bereits Anwesenden an. Damit klingen die Gläser genau n weitere Male und insgesamt also

$$\frac{n(n-1)}{2} + n = \frac{n(n-1)+2n}{2} = \frac{n(n-1+2)}{2} = \frac{(n+1)n}{2}$$

   Male. ✓

Somit ist die intuitiv gefundene Formel formal mithilfe des Prinzips der vollständigen Induktion nachgewiesen.

3.2 Erklimmen einer Leiter

In Beispiel 3.1 und dem Beweis von Theorem 3.1 haben wir das Prinzip der vollständigen Induktion als Beweistechnik verwendet. Dabei haben wir jeweils auf geschickte Weise eine „Hilfsmenge“ N definiert und gezeigt, dass N die Menge aller natürlichen Zahlen $\mathbb{N}$ ist. Nun möchten wir aber mathematische Aussagen beweisen und nicht unbedingt über Mengen sprechen. Deshalb ist es intuitiv einfacher, im Beweisverfahren der vollständigen Induktion den Begriff der *Menge* durch den Begriff der *Aussage* zu ersetzen. Um dies konkreter zu machen, betrachten wir nochmals Beispiel 3.1. In dem Beispiel kann man für jedes $n \in \mathbb{N}$ die Aussage $\mathcal{A}(n)$ definieren als

$$\mathcal{A}(n) : \Longleftrightarrow \text{ Die Zahl } n^2 + n \text{ ist gerade.}$$

In der Sprache der *Aussagen* erhalten wir das folgende rezeptartige Beweisverfahren:

Zusammenfassung 3.1 (Beweis durch vollständige Induktion)**:**
Für jedes $n \in \mathbb{N}$ sei $\mathcal{A}(n)$ eine Aussage. Wir wollen beweisen, dass $\mathcal{A}(n)$ für jedes $n \in \mathbb{N}$ richtig ist. Der Beweis kann mithilfe des Prinzips der vollständigen Induktion erbracht werden, indem

wie folgt vorgegangen wird:
(a) *Induktionsanfang*: Es wird gezeigt, dass $\mathcal{A}(0)$ richtig ist.
(b) *Induktionsschluss*: Dieser besteht aus zwei Teilen:
  (i) *Induktionsvoraussetzung*: Es sei n eine natürliche Zahl und $\mathcal{A}(n)$ sei richtig.
  (ii) *Induktionsschritt* ($n \to n+1$): Man zeigt, dass aus der Induktionsvoraussetzung (i), logischen Schlüssen und bereits als wahr erkannten Aussagen die Richtigkeit von $\mathcal{A}(n+1)$ abgeleitet werden kann.

Damit ist die Richtigkeit von $\mathcal{A}(n)$ für alle $n \in \mathbb{N}$ gezeigt.

Beachten Sie, dass das Prinzip der vollständigen Induktion dem Axiom 2.5 entspricht und als solches nicht bewiesen werden kann. Was wir aber anbieten können, ist eine Metapher, welche das Prinzip veranschaulicht:

Bemerkung 3.1 (Metapher der Leiter)**:**
Die vollständige Induktion beweist, dass wir auf einer Leiter beliebig weit hochsteigen können, indem sie beweist, dass wir den untersten Tritt (Induktionsanfang) erreichen können und, dass wir von jedem Tritt den nächsthöheren Tritt erreichen können (Induktionsschluss).

Aufgabe 3.3

Erklären Sie, wie das rezeptartige Beweisverfahren in Zusammenfassung 3.1 aus Axiom 2.5 folgt. Tipp: Definieren Sie die „Hilfsmenge“

$$N := \{\, n \in \mathbb{N} \,;\, \mathcal{A}(n) \text{ ist richtig} \,\}$$

aller natürlichen Zahlen, für welche $\mathcal{A}(n)$ richtig ist.

Aufgabe 3.4

Beweisen Sie durch vollständige Induktion, dass für jede natürliche Zahl n die Zahl $5^n - 1$ durch 4 teilbar ist.

3.3 Zwei bedeutende Sätze über natürliche Zahlen

Das Prinzip der vollständigen Induktion ist in der Mathematik und Informatik von enormer Bedeutung. Wir wollen in diesem anspruchsvollen Abschnitt noch mehr verdeutlichen, wie weitreichend die Beweiskraft dieses Prinzips ist. Dazu möchten wir zwei bedeutende Sätze der Mathematik besprechen und zeigen, wie diese aus dem Prinzip der vollständigen Induktion folgen.

3.3.1 Wohlordnungsprinzip

Beachten Sie, dass die Menge der ganzen Zahlen $\mathbb{Z}$ kein Minimum (kleinstes Element) besitzt. Zu jeder ganzen Zahl m ist offensichtlich $m-1$ ebenfalls eine ganze Zahl, die (noch) kleiner ist als m. Die Teilmenge $\{-3, -1, 0, 7\}$ von $\mathbb{Z}$ hingegen besitzt -3 als Minimum.

Definition 3.1 (Minimum)**:**
Es sei A eine nichtleere Menge, in der sich Elemente durch die Relation $\leq$ vergleichen lassen.

Ein Element $m \in A$ heisst ***Minimum*** von A, falls

$$m \leq x$$

für alle $x \in A$ gilt. Das Minimum einer Menge A muss selbst Element von A sein. Beachten Sie, dass A offensichtlich höchstens ein Minimum enthalten kann. Dieses wird mit $\min(A)$ bezeichnet.

Aufgabe 3.5

(a) Welches ist das Minimum von $\mathbb{N}$?
(b) Erklären Sie, warum das halboffene Intervall

$$(0, 1] := \{\, x \in \mathbb{R} \,;\, 0 < x \leq 1 \,\}$$

kein Minimum besitzt.

Wir haben gesehen, dass durchaus nicht jede Menge ein Minimum besitzt. Das sogenannte ***Wohlordnungsprinzip*** der natürlichen Zahlen garantiert uns aber, dass jede Teilmenge von $\mathbb{N}$, welche nicht die leere Menge ist, ein Minimum besitzt. Die leere Menge besitzt keine Elemente und somit offensichtlich auch kein minimales Element. Weshalb ist das Wohlordnungsprinzip für uns interessant? Dieses Prinzip kommt in den Beweisen einiger wichtiger mathematischer Behauptungen zum Einsatz, so zum Beispiel in unserem Beweis des berühmten *Fundamentalsatzes der Arithmetik*, den wir in Unterabschnitt 3.3.2 besprechen.

Definition 3.2 (untere Schranke)**:**
Seien D eine Menge und $A \subseteq D$ eine nichtleere Teilmenge von D. Jedes Element $s \in D$, welches $s \leq a$ für alle $a \in A$ erfüllt, heisst ***untere Schranke*** von A. Eine untere Schranke von A muss selbst nicht ein Element von A sein.

Wir erkennen nun: Ein Element $m \in \mathbb{R}$ heisst *Minimum* von $A \in \mathbb{R}$, falls m eine untere Schranke von A ist und zusätzlich $m \in A$ gilt.

Beispiel 3.3: (a) Die Zahlen $-5, 0, 3, 4$ sind Beispiele von unteren Schranken der Menge $A := \{4, 7, 9, 18\}$. Da $4 \in A$ und 4 eine untere Schranke von A ist, gilt

$$\min(A) = 4.$$

(b) Die Menge $\mathbb{Z}$ der ganzen Zahlen besitzt keine untere Schranke.

Aufgabe 3.6

Welches ist die grösste untere Schranke des halboffenen Intervalls

$$(0, 1] := \{\, x \in \mathbb{R} \,;\, 0 < x \leq 1 \,\}?$$

Aufgabe 3.7

(a) Sei $A \subseteq \mathbb{N}$ eine Teilmenge der natürlichen Zahlen. Angenommen $n \in \mathbb{N}$ sei eine untere Schranke von A, wobei aber n selbst nicht Element der Menge A ist (n ist also nicht das Minimum von A). Begründen Sie, warum auch der Nachfolger $n+1$ eine untere Schranke von A ist.

(b) Sei $A \subseteq \mathbb{N}$ eine Teilmenge der natürlichen Zahlen mit der Eigenschaft, dass jede natürliche Zahl eine untere Schranke von A ist. Beweisen Sie, dass A die leere Menge ist. Tipp: Nehmen Sie an, A sei nichtleer. Dann enthält A also (zumindest) ein Element $m \in A$. Betrachten Sie nun die natürliche Zahl $m+1$.

Wir verwenden nun das Prinzip der vollständigen Induktion, um das *Wohlordnungsprinzip* zu beweisen. Wir haben den Beweis dieses Satzes inzwischen recht gut vorbereitet. Dennoch ist der Beweis recht anspruchsvoll! Nehmen Sie sich Zeit, um die einzelnen Schritte zu studieren.

Theorem 3.1 (Wohlordnungsprinzip)**:**
Jede nichtleere Teilmenge der natürlichen Zahlen besitzt ein Minimum.

Beweis 3.1:
Wir beweisen den Satz durch Widerspruch. Angenommen eine Teilmenge $A \subseteq \mathbb{N}$ sei **nichtleer** und besitze **kein Minimum**. Dann definieren wir zu A die „Hilfsmenge“

$$N := \{\, n \in \mathbb{N} \,;\, n \text{ ist untere Schranke von } A \,\}$$

aller unteren Schranken von A. Mit dem Prinzip der vollständigen Induktion beweisen wir, dass jede natürliche Zahl eine untere Schranke von A ist, dass also $N = \mathbb{N}$ gilt. In Aufgabe 3.6 haben Sie bereits bewiesen, dass dies nur möglich ist, falls A die leere Menge ist. Um $N = \mathbb{N}$ durch die vollständige Induktion zu beweisen, müssen wir, wie immer, zwei Bedingungen prüfen:

1. $0 \in N$,
2. Ist $n \in N$, dann enthält N auch den Nachfolger $\nu(n) = n+1$ von n. (Ist n eine untere Schranke von A, dann ist auch $n+1$ eine untere Schranke von A.)

Wir prüfen diese beiden Bedingungen separat:

1. 0 ist die kleinste Zahl in $\mathbb{N}$ und somit gilt $0 \leq a$ für jedes Element $a \in A$. Damit ist 0 eine untere Schranke von A und wir haben $0 \in N$ gezeigt. ✓
2. Es sei $n \in N$ und somit n eine untere Schranke von A. Beachten Sie, dass n nicht in A liegt, denn sonst würde die Menge A die Zahl n als ihr Minimum besitzen (doch A besitzt gemäss Annahme kein Minimum). Damit ist $n \in \mathbb{N}$ eine untere Schranke von A, wobei aber n selbst nicht Element der Menge A ist. In Aufgabe 3.7 haben Sie gezeigt, dass dann auch $n+1$ eine untere Schranke von A und somit ein Element von N ist. ✓

Aus dem Prinzip der vollständigen Induktion folgt nun, dass $N = \mathbb{N}$ gilt. Also ist jede natürliche Zahl eine untere Schranke von A. Somit ist $A \subseteq \mathbb{N}$ eine Teilmenge der natürlichen Zahlen mit der Eigenschaft, dass jede natürliche Zahl eine untere Schranke von A ist. In Aufgabe 3.7 haben Sie gezeigt, dass A somit die leere Menge ist. Damit haben wir den gewünschten Widerspruch $A \neq \emptyset$ und $A = \emptyset$ gefunden. Somit besitzt jede nichtleere Teilmenge von $\mathbb{N}$ ein Minimum.

3.3.2 Fundamentalsatz der Arithmetik

Die Tatsache, dass jede Teilmenge der natürlichen Zahlen ein Minimum besitzt, erlaubt uns, den berühmten *Fundamentalsatz der Arithmetik* zu beweisen.

Theorem 3.2:
Fundamentalsatz der Arithmetik (Primfaktorzerlegung) Ausser 0 und 1 kann jede natürliche Zahl als Produkt endlich vieler Primzahlen dargestellt werden. Diese Darstellung ist bis auf die Reihenfolge der Faktoren eindeutig und wird *Primfaktorzerlegung* genannt. Erlaubt sind auch Produkte, die nur aus einem Faktor bestehen.

Beispiel 3.4: (a) Die Zahl 63 besitzt die Primfaktorzerlegung $63 = 3 \cdot 3 \cdot 7$ und die Zahl 286 die Darstellung $286 = 2 \cdot 11 \cdot 13$.
(b) Die Primfaktorzerlegung der Primzahl 19 ist 19 selbst. Sie besteht also aus dem Produkt mit nur dem einen Faktor 19. Jede Primzahl p ist also bereits in Primfaktoren zerlegt.

Beweis 3.2:
Wir wollen nun den Fundamentalsatz (Theorem 3.2) beweisen. Dies tun wir unter der Verwendung des Wohlordnungsprinzips. Allerdings zeigen wir nur, dass immer eine Zerlegung in Primfaktoren existiert. Auf den Beweis der Eindeutigkeit verzichten wir an dieser Stelle.
Angenommen die Behauptung sei falsch. Dann gibt es eine natürliche Zahl ≥ 2, welche keine Primfaktorzerlegung besitzt. Mit anderen Worten: Die Menge

$$A := \{\, n \in \mathbb{N} \,;\, n \geq 2 \text{ und } n \text{ besitzt keine Primfaktorzerlegung} \,\}$$

ist **nichtleer**. Dann liegt in A aber gemäss Theorem 3.1 eine kleinste Zahl $n_0 \geq 2$, welche nicht in Primfaktoren zerlegt werden kann. Insbesondere ist n_0 keine Primzahl. Da n_0 keine Primzahl ist, existieren natürliche Zahlen n und m, sodass $n_0 = n \cdot m$ und $n, m > 1$. Es ist klar, dass die Faktoren n und m jeweils kleiner als das Produkt n_0 sind. Da aber n_0 die kleinste natürliche Zahl ist, welche keine Primfaktorzerlegung besitzt, können sowohl n als auch m jeweils als Produkt endlich vieler Primzahlen geschrieben werden. Dann ist aber auch n_0 als Produkt von n und m ein Produkt endlich vieler Primzahlen. Dieses Produkt wäre dann aber eine Primfaktorzerlegung von n_0. Das ist ein Widerspruch.

Aufgabe 3.8

Die Zahl 30031 lässt sich als Produkt

$$30031 = p_1 p_2$$

zweier verschiedener Primfaktoren p_1 und p_2 schreiben. Schreiben Sie ein Python-Programm, welches p_1 und p_2 berechnet.

Aufgabe 3.9

Der griechische Mathematiker *Euklid von Alexandria* bewies bereits im 3. Jahrhundert v. Chr., dass unendlich viele Primzahlen existieren. Dazu nahm er (indirekter Beweis) an, die Menge P aller Primzahlen sei endlich. Wir können P also schreiben als

$$P = \{p_1, p_2, p_3, \ldots, p_n\}\,.$$

Nun betrachtete Euklid das Produkt dieser n Primzahlen und addierte dazu 1:

$$m := p_1 \cdot p_2 \cdot p_3 \cdot \ldots \cdot p_n + 1.$$

Betrachten Sie die so entstandene Zahl m. Welche Eigenschaften hat m gemäss unseren Annahmen? Vervollständigen Sie den Beweis.

Bemerkung 3.2:
Beachten Sie, dass das Vorgehen in Aufgabe 3.9 in keinster Weise ein Rezept zur Konstruktion von Primzahlen liefert. Beispielsweise gilt

$$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 + 1 = 30031,$$

doch Sie haben in Aufgabe 3.8 bereits gezeigt, dass 30031 keine Primzahl ist.

3.4 Leicht verallgemeinertes Induktionsprinzip

Wie Sie sofort nachprüfen können, ist die mathematische Aussage $n^2 - 2n - 1 > 0$ für die natürlichen Zahlen $n = 0, 1, 2$ falsch. Nun möchten Sie aber beweisen, dass die Aussage für alle $n \in \mathbb{N}$ mit $n \geq 3$ gilt. Unser Beweisverfahren in Zusammenfassung 3.1 muss also dahingehend angepasst werden, dass es uns auch erlaubt, den Induktionsanfang bei einer beliebigen Zahl $n_0 \in \mathbb{N}$ anzusetzen, wobei n_0 auch grösser als 0 sein darf. Diese sehr geringfügige (aber wichtige) Verallgemeinerung fassen wir in dem folgenden Satz zusammen:

Theorem 3.3 ((leicht verallgemeinertes) Induktionsprinzip)**:**
Sei $n_0 \in \mathbb{N}$ und für jedes $n \in \mathbb{N}$ mit $n \geq n_0$ sei $\mathcal{A}(n)$ eine Aussage. Zudem gelte:

(i) $\mathcal{A}(n_0)$ ist richtig.

(ii) Für jede Zahl $n \in \mathbb{N}$ mit $n \geq n_0$ folgt aus der Richtigkeit von $\mathcal{A}(n)$ die Richtigkeit von $\mathcal{A}(n+1)$.

Dann ist $\mathcal{A}(n)$ für jedes $n \geq n_0$ richtig.

Beweis 3.3:
Der Beweis ist fast komplett analog zu der Begründung in Aufgabe 3.3. Der einzige Unterschied besteht darin, dass wir einen Index „verschieben“ müssen. Intuitiv gesprochen, möchten wir, dass wir immer noch bei 0 beginnen, anstelle von $\mathcal{A}(0)$ aber bereits $\mathcal{A}(n_0)$ betrachten. Wir definieren also die Menge N der um „n_0 verschobenen Aussagen“

$$N := \{\, n \in \mathbb{N} \,;\, \mathcal{A}(n + n_0) \text{ ist richtig} \,\}.$$

Beachten Sie, dass für $n = 0$ dadurch bereits die Aussage $\mathcal{A}(0 + n_0) = \mathcal{A}(n_0)$ gemeint ist. Mit dem Induktionsaxiom 2.5 folgt sofort $N = \mathbb{N}$ und somit ist $\mathcal{A}(n)$ für jedes $n \geq n_0$ richtig.

Bemerkung 3.3:
Wir werden das leicht verallgemeinerte Induktionsprinzip von Theorem 3.3 ebenfalls einfach nur *Induktionsprinzip* nennen.

3.5 Übungsaufgaben zum Induktionsprinzip

Aufgabe 3.10

Setzen Sie $n_0 := 3$ und beweisen Sie die Korrektheit der Ungleichung $n^2 - 2n - 1 > 0$ für alle $n \in \mathbb{N}$ mit $n \geq n_0$ mittels vollständiger Induktion.

Aufgabe 3.11

Sie vermuten, dass die Zahl $n^3 - n$ für jede natürliche Zahl $n \geq 2$ durch 3 teilbar ist. Beweisen Sie diese Behauptung durch vollständige Induktion. Finden Sie auch einen direkten Beweis (welcher nicht das Induktionsprinzip verwendet)? Tipp für den direkten Beweis: Schreiben Sie die Zahl $n^3 - n$ als Produkt dreier Faktoren.

Aufgabe 3.12 Bernoullische Ungleichung

Es sei $x \in \mathbb{R}$ mit $x \geq -1$ eine reelle Zahl. Für jede natürliche Zahl n gilt die Bernoullische Ungleichung $(1 + x)^n \geq 1 + nx$.

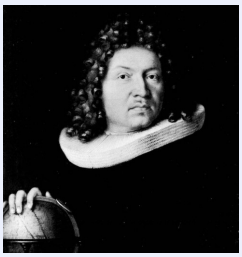


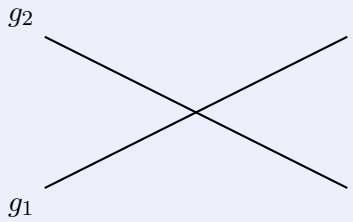
Abbildung 3.1: Jakob I Bernoulli (1654-1705) war ein Schweizer Mathematiker und Mitglied der angesehenen Gelehrtenfamilie Bernoulli.

Aufgabe 3.13

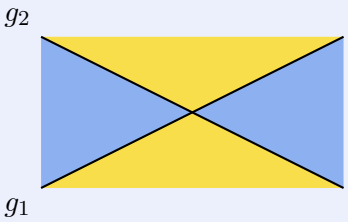
In die Ebene wurden $n \in \mathbb{N}$ verschiedene Geraden gelegt. Die Geraden teilen die Ebene in verschiedene Bereiche. Wir sagen, dass zwei Bereiche *benachbart* sind, falls sie sich eine (möglicherweise unendlich lange) Grenzlinie teilen. Falls zwei Bereiche lediglich einen Grenzpunkt teilen (keine Grenzlinie), sind sie nicht benachbart.

Wir haben zwei Farben zur Verfügung und müssen jeden Bereich mit genau einer dieser Farben einfärben. Eine Färbung wird *zulässig* genannt, falls zusätzlich keine zwei benachbarten Bereiche mit derselben Farbe gefärbt wurden. In Abbildung 3.2 ist eine mögliche zulässige Färbung für zwei Geraden ($n = 2$) gezeigt.

Skizzieren Sie eine zulässige Färbung für $n = 3$ für den Fall, dass sich alle drei Geraden jeweils gegenseitig schneiden. Beweisen Sie anschliessend durch vollständige Induktion, dass es immer möglich ist, die verschiedenen Bereiche mit nur zwei Farben so zu färben, dass keine zwei benachbarten Bereiche von derselben Farbe sind.



(a) zwei sich schneidende Geraden



(b) zulässige Färbung für $n = 2$

Abbildung 3.2: Beispiel einer zulässigen Färbung für $n = 2$. Beachten Sie, dass die beiden goldenen Bereiche sich nur einen Grenzpunkt (keine Grenzlinie) teilen und somit nicht benachbart sind. Das Gleiche gilt für die beiden blauen Bereiche. Zwei sich schneidende Geraden teilen die Ebene in vier Bereiche ein. Zwei parallele Geraden teilen die Ebene in drei Bereiche ein.

Aufgabe 3.14

Die Idee für das folgende scheinbare Paradoxon wird häufig dem ungarischen Mathematiker George Pólya zugeschrieben. Pólya war von 1914 bis 1940 Professor der Mathematik an der ETH Zürich. Das scheinbare Paradoxon besteht darin, dass vermeintlich korrekt bewiesen wird, dass alle Pferde dieselbe Farbe haben (alle Zahlen sind gleich / alle Mädchen haben dieselbe Augenfarbe).

Erklären Sie ganz präzise, an welcher Stelle / welchen Stellen die folgende Argumentation fehlerhaft ist.

Behauptung:

Wir verwenden Theorem 3.3 um zu beweisen, dass alle Pferde dieselbe Farbe haben. Dazu definieren wir für jedes $n \in \mathbb{N}^\times$ die Aussage $\mathcal{A}(n)$ als

$$\mathcal{A}(n) : \Longleftrightarrow \text{ In einer Menge von } n \text{ Pferden haben alle dieselbe Farbe.}$$

Wir behaupten die Richtigkeit von $\mathcal{A}(n)$ für alle $n \in \mathbb{N}^\times$.

Beweis 3.4: (a) *Induktionsanfang*: $\mathcal{A}(1)$ ist richtig, denn in einer Menge mit nur einem Pferd haben offensichtlich alle Pferde dieselbe Farbe. Somit stimmt die Aussage für $n = 1$.

(b) *Induktionsschluss*:

(i) *Induktionsvoraussetzung*: Es sei $n \geq 1$ eine natürliche Zahl und $\mathcal{A}(n)$ sei richtig. Das heisst, in einer Menge von n Pferden haben alle dieselbe Farbe.

(ii) *Induktionsschritt* ($n \to n+1$): Durch Hinzufügen eines weiteren (neuen) Pferdes p' zu einer Menge von n Pferden erhalten wir eine Menge von $n+1$ Pferden. Nun nehmen wir ein Pferd $\tilde{p}$, welches aber nicht p' ist ($\tilde{p} \neq p'$), aus der Menge heraus und erhalten dadurch wieder eine Menge von n Pferden. In dieser neuen Menge, die p' enthält, haben gemäss Induktionsvoraussetzung alle Pferde dieselbe Farbe. Damit hat aber das neue Pferd p' dieselbe Farbe wie die n anderen. Nun fügen wir das entfernte Pferd $\tilde{p}$ wieder zur Menge hinzu und erhalten eine Menge mit $n+1$ Pferden, welche alle dieselbe Farbe haben. Dies zeigt den Induktionsschritt.

Damit ist die Richtigkeit von $\mathcal{A}(n)$ für alle $n \in \mathbb{N}^\times$ gezeigt. Somit sind in jeder (beliebigen) endlichen Menge von Pferden nur Pferde derselben Farbe enthalten. Das geht aber nur, wenn tatsächlich alle Pferde dieselbe Farbe haben.

3.6 Starke Induktion

Das Induktionsprinzip (Theorem 3.3) verlangt, dass zum Nachweis der Richtigkeit von $\mathcal{A}(n+1)$, nebst bereits bekanntem Wissen, lediglich die Richtigkeit von $\mathcal{A}(n)$ verwendet werden darf. Manchmal wäre es aber sehr nützlich, zum Nachweis von $\mathcal{A}(n+1)$ nebst $\mathcal{A}(n)$ zusätzlich noch die Aussagen $\mathcal{A}(k)$ mit $k < n$ annehmen zu dürfen. Durch diese zusätzlichen Annahmen wird der Nachweis der Korrektheit von $\mathcal{A}(n+1)$ entweder erleichtert oder bleibt genauso schwierig wie im bisherigen Induktionsschritt. Sicherlich wird der Nachweis dadurch nicht schwieriger. Man sagt dann, dass wir *stärkere Annahmen* treffen.

Theorem 3.4 (starke Induktion)**:**
Sei $n_0 \in \mathbb{N}$ und für jedes $n \in \mathbb{N}$ mit $n \geq n_0$ sei $\mathcal{A}(n)$ eine Aussage. Zudem gelte:

(i) $\mathcal{A}(n_0)$ ist richtig.

(ii) Für jede Zahl $n \in \mathbb{N}$ mit $n \geq n_0$ gilt:
Aus der Richtigkeit der Aussagen $\mathcal{A}(k)$ für $n_0 \leq k \leq n$ folgt die Richtigkeit von $\mathcal{A}(n+1)$.

Dann ist $\mathcal{A}(n)$ für jedes $n \geq n_0$ richtig.

Bemerkung 3.4:
Die ***starke Induktion*** besagt im Grunde, dass wir uns den Nachweis des Induktionsschritts (im Vergleich zu Theorem 3.3) durch zusätzliche (stärkere) Annahmen vereinfachen dürfen und trotzdem die komplette Aussagekraft von Theorem 3.3 behalten.

Beispiel 3.5:
Auf einer Auslandsreise sprechen wir mit anderen Touristen über Währungen verschiedener Länder. Der US-Amerikaner *Doug* kennt sich gut mit der Geschichte seines Landes aus und erzählt uns, dass für eine gewisse Zeit in den noch jungen vereinigten Staaten 4-Dollarmünzen und 5-Dollarmünzen geprägt wurden. Diese kunstvollen Münzen sind in Abbildung 3.3 gezeigt.

(a) 4-Dollarmünze „Stella“ (b) 5-Dollarmünze „Half Eagle“

Abbildung 3.3: US-amerikanische 4-Dollar- und 5-Dollarmünze

Doug findet es schade, dass diese Münzen nicht mehr geprägt werden. Schliesslich liesse sich jeder ganzzahlige Dollar-Betrag, der grösser als 11 Dollar ist, alleine durch eine Kombination dieser beiden Münzen auszahle — dies natürlich nur unter der Annahme, dass beliebig viele Exemplare von beiden Münzen zur Verfügung stehen.
Als kritische Menschen sind wir nicht bereit, Doug einfach so zu glauben. Um Dougs Behauptung zu überprüfen, verwenden wir die starke Induktion (Theorem 3.4). Wir definieren für jedes $n \in \mathbb{N}$ die Aussage $\mathcal{A}(n)$ als

$\mathcal{A}(n) :\Longleftrightarrow$ „Der Dollar-Betrag n kann durch eine Kombination aus 4-Dollarmünzen und 5-Dollarmünzen ausbezahlt werden.“

Dougs Behauptung besagt also, dass $\mathcal{A}(n)$ für jedes $n \in \mathbb{N}$ mit $n \geq 12$ gilt.
Der Betrag $n_0 = 12$ kann wegen $12 = 3 \cdot 4 + 0 \cdot 5$ durch drei 4er und null 5er ausbezahlt werden. Somit stimmt $\mathcal{A}(n_0) = \mathcal{A}(12)$. Zum Nachweis von $\mathcal{A}(n_0 + 1) = \mathcal{A}(13)$ dürften wir $\mathcal{A}(12)$ verwenden. Zum Nachweis von $\mathcal{A}(14)$ dürften wir $\mathcal{A}(12)$ und $\mathcal{A}(13)$ verwenden und zum Nachweis von $\mathcal{A}(15)$ gar $\mathcal{A}(12), \mathcal{A}(13)$ und $\mathcal{A}(14)$. Wir sehen jedoch sofort, dass

$$13 = 2 \cdot 4 + 1 \cdot 5, \quad 14 = 1 \cdot 4 + 2 \cdot 5, \quad 15 = 0 \cdot 4 + 3 \cdot 5,$$

und somit sind nebst $\mathcal{A}(12)$ auch $\mathcal{A}(13), \mathcal{A}(14)$ und $\mathcal{A}(15)$ nachgewiesen.
Sei nun $n \in \mathbb{N}$ mit $n \geq 15$. Wir beweisen, dass aus der Richtigkeit von $\mathcal{A}(k)$ für $12 \leq k \leq n$ die Richtigkeit von $\mathcal{A}(n+1)$ folgt. Wir betrachten den Dollar-Betrag $n+1-4$. Offensichtlich gilt $12 \leq n+1-4 \leq n$. Gemäss (starker) Induktionsannahme kann der Betrag $n+1-4$ aber durch eine Kombination aus 4-Dollarmünzen und 5-Dollarmünzen ausbezahlt werden. Dies ist die Aussage $\mathcal{A}(n+1-4) = \mathcal{A}(n-3)$. Durch das Hinzufügen einer einzigen 4-Dollarmünze zu dieser Kombination erhalten wir eine gewünschte Auszahlung des Dollar-Betrags $n+1$.

Aufgabe 3.15

Diese Aufgabe bezieht sich auf Beispiel 3.5. Schreiben Sie ein einfaches Python-Programm, welches für einen gegebenen ganzzahligen Dollar-Betrag $n \geq 12$ sämtliche möglichen Auszahlungen durch 4-Dollarmünzen und 5-Dollarmünzen ausgibt.

Tipp: Verwenden Sie eine geschachtelte Schleife (eine Doppel-Schleife).

Wir sind noch einen Beweis von Theorem 3.4 schuldig:

Beweis 3.5:
Wir beweisen Theorem 3.4. Angenommen der Satz sei falsch und somit $\mathcal{A}(n)$ nicht jedes $n \geq n_0$ richtig. Dann ist die Menge

$$N := \{\, n \in \mathbb{N} \,;\, n \geq n_0 \text{ und } \mathcal{A}(n) \text{ ist falsch} \,\}$$

nichtleer. Gemäss Theorem 3.1 besitzt die Menge N ein minimales Element m. Da $\mathcal{A}(n_0)$ wegen Bedingung (i) richtig ist, muss $m > n_0$ gelten. Es existiert eine eindeutige natürliche Zahl n mit $n + 1 = m$. Aufgrund der Definition von m gilt, dass die Aussagen $\mathcal{A}(k)$ für alle natürlichen Zahlen k mit $n_0 \leq k \leq n$ richtig sind. Dann garantiert Bedingung (ii) aber die Richtigkeit von $\mathcal{A}(n + 1) = \mathcal{A}(m)$. Doch dies ist nach der Definition von m unmöglich und wir haben einen Widerspruch gefunden.

Aufgabe 3.16

Der junge Tim besitzt beliebig viele Holzklötze der Länge 7 und beliebig viele der Länge 8. Klötze anderer Längen hat er keine.

(a) Tim weiss, dass sein Plüschkrokodil die Länge 38 hat. Nun möchte er mit seinen Klötzen eine Strecke derselben Länge bauen. Doch bislang blieben alle seine Versuche erfolglos. Helfen Sie Tim, indem Sie ein Python-Programm schreiben, welches Ihnen angibt, wie viele Klötze der Länge 7 und wie viele der Länge 8 benötigt werden, um die Strecke zu bauen.

(b) Weisen Sie nach, dass es mit Tims Klötzen nicht möglich ist, eine Strecke der Länge 41 zu bauen. Schreiben Sie dazu ein Python-Programm, welches alle denkbaren Möglichkeiten ausprobiert.

(c) Beweisen Sie mithilfe von Theorem 3.4 zur starken Induktion, dass jede Strecke der Länge $n \in \mathbb{N}$ mit $n \geq 42$ mit Tims Klötzen gebaut werden kann.

Aufgabe 3.17 kommutative Verknüpfung

Seien $+$ und $\cdot$ assoziative und kommutative Verknüpfungen auf einer Menge X, welche das Distributivgesetz

$$(x + y) \cdot z = x \cdot z + y \cdot z$$

für $x, y, z \in X$ erfüllen. Beweisen Sie mithilfe von Theorem 3.4 zur starken Induktion die Richtigkeit des „verallgemeinerten“ Distributivgesetztes:

$$\mathcal{A}(n) : \Longleftrightarrow \; c \sum_{k=0}^{n} (x_k) = \sum_{k=0}^{n} (c x_k)$$

für alle $n \in \mathbb{N}$. Dabei sind $c \in X$ und (x_k) eine Folge in X.

3.7 Lösungen der Aufgaben

Lösungsvorschlag zu Aufgabe 3.1

```python
def vermutung(m):
    for n in range(m):
        if (n * n + n) % 2 != 0:
            return False
    return True

m = 100
print(vermutung(m))
```

Programm 3.1: Vermutung überprüfen

Lösungsvorschlag zu Aufgabe 3.2

Wir definieren die „Hilfsmenge“

$$N := \left\{ n \in \mathbb{N} \,;\, n^5 - n \text{ ist durch 5 teilbar} \right\}.$$

Die Behauptung der Aufgabe ist bewiesen, wenn wir nachweisen können, dass alle natürlichen Zahlen in N liegen, dass also die Gleichheit $N = \mathbb{N}$ gilt. Das Prinzip der vollständigen Induktion (Axiom 2.5) besagt, dass für eine Teilmenge $N \subseteq \mathbb{N}$ tatsächlich $N = \mathbb{N}$ gilt, falls zwei Bedingungen erfüllt sind:

1. $0 \in N$,
2. ist $n \in N$, dann enthält N auch den Nachfolger $\nu(n) = n + 1$.

Wir prüfen nun diese beiden Bedingungen.

1. In der Tat gilt $0 \in N$, denn $0^5 - 0 = 0$ ist durch 5 teilbar. ✓
2. Sei $n \in N$ und somit $n^5 - n$ durch 5 teilbar. Wir müssen beweisen, dass auch $\nu(n) = (n+1) \in N$ gilt, dass also auch $(n+1)^5 - (n+1)$ durch 5 teilbar ist. Dazu betrachten wir die folgende Berechnung:

$$(n+1)^5 - (n+1) = n^5 + 5n^4 + 10n^3 + 10n^2 + 5n + 1 - n - 1 = \\ \left(n^5 - n\right) + 5\left(n^4 + 2n^3 + 2n^2 + n\right)$$

   Da n in der Menge N liegt, ist die Zahl $n^5 - n$ durch 5 teilbar. Somit ist $(n+1)^5 - (n+1)$ als Summe zweier durch 5 teilbarer Zahlen ebenfalls durch 5 teilbar. ✓

Lösungsvorschlag zu Aufgabe 3.3

Im *Induktionsanfang* (a) wird gezeigt, dass $0 \in N$ gilt. Der *Induktionsschluss* liefert schliesslich, dass für $n \in N$ auch $(n+1) \in N$ gilt. Gemäss Axiom 2.5 gilt somit $N = \mathbb{N}$. Dies bedeutet aber, dass $\mathcal{A}(n)$ in der Tat für alle $n \in \mathbb{N}$ gilt.

Lösungsvorschlag zu Aufgabe 3.4

Für jedes $n \in \mathbb{N}$ definieren wir die Aussage $\mathcal{A}(n)$ als

$$\mathcal{A}(n) : \Longleftrightarrow \text{ Die Zahl } 5^n - 1 \text{ durch 4 teilbar.}$$

(a) *Induktionsanfang*: Wir zeigen, dass $\mathcal{A}(0)$ richtig ist. Dies ist aber klar, denn $5^0 - 1 = 0$ und 0 ist durch 4 teilbar.

(b) *Induktionsschluss*:

(i) *Induktionsvoraussetzung*: Es sei n eine natürliche Zahl und $\mathcal{A}(n)$ sei richtig. Sei also $5^n - 1$ durch 4 teilbar.

(ii) *Induktionsschritt* ($n \to n+1$): Wir zeigen, dass aus der Induktionsvoraussetzung (i), logischen Schlüssen und bereits als wahr erkannten Aussagen die Richtigkeit von $\mathcal{A}(n+1)$ abgeleitet werden kann. Die Aussage $\mathcal{A}(n+1)$ lautet: $5^{n+1} - 1$ ist durch 4 teilbar. Wir berechnen:

$$5^{n+1} - 1 = 5 \cdot 5^n - 1 = (4+1) \cdot 5^n - 1 = 4 \cdot 5^n + 5^n - 1.$$

Die Zahl $4 \cdot 5^n$ ist als Vielfaches von 4 offensichtlich durch 4 teilbar und $5^n - 1$ ist gemäss Induktionsvoraussetzung durch 4 teilbar. Somit haben wir gezeigt, dass $5^{n+1} - 1$ eine Summe von Termen ist, welche jeweils durch 4 teilbar sind. Damit ist aber auch $5^{n+1} - 1$ durch 4 teilbar und der Induktionsschritt ist bewiesen.

Aus dem Induktionsprinzip folgt somit, dass $\mathcal{A}(n)$ für jedes $n \in \mathbb{N}$ korrekt ist.

Lösungsvorschlag zu Aufgabe 3.5

(a) Die Zahl 0 ist das Minimum von $\mathbb{N}$, da $0 \in \mathbb{N}$ und $0 \leq x$ für alle $x \in \mathbb{N}$.

(b) Für jedes Element $m \in (0, 1]$ ist auch die Hälfte $m/2$ von m Element des Intervalls. Dann gilt aber auch $(m/2) \in (0, 1]$. Anders gesagt: Angenommen $m \in (0, 1]$ sei das Minimum des Intervalls. Dann gilt aber auch $(m/2) \in (0, 1]$. Wegen $m/2 < m$ ist dies aber ein Widerspruch zur Minimalität von m.

Lösungsvorschlag zu Aufgabe 3.6

Das halboffene Intervall $(0, 1] := \{ x \in \mathbb{R} \; ; \; 0 < x \leq 1 \}$ besitzt die grösste untere Schranke 0, aber kein Minimum.

Lösungsvorschlag zu Aufgabe 3.7

(a) Da n eine untere Schranke von A ist, gilt $n \leq a$ für alle $a \in A$. Da aber $n \notin A$, ist klar, dass tatsächlich sogar $n < a$ (echt kleiner) für alle $a \in A$ gilt. Der Nachfolger (die nächstgrössere natürliche Zahl) von n ist $n + 1$. Damit ist klar, dass $n + 1$ höchstens so gross wie jedes der Elemente in A ist, dass also $n + 1 \leq a$ für alle $a \in A$ gilt.

(b) Angenommen A wäre nichtleer. Dann enthält A mindestens ein Element $m \in A$. Die natürliche Zahl $m + 1$ ist grösser als $m \in A$ und somit keine untere Schranke von A. Doch A besitzt die Eigenschaft, dass jede natürliche Zahl eine untere Schranke von A ist. Dies ist ein Widerspruch und somit muss A die leere Menge sein.

Lösungsvorschlag zu Aufgabe 3.8

```
n = 30031
for k in range(2,n):
    if (n % k == 0):
        p1 = k
        break
p2 = int(n / p1)
print(n,'=', p1, '*',p2)
```

Lösungsvorschlag zu Aufgabe 3.9

Da $m \geq 2$ ist, lässt sich m gemäss Theorem 3.2 als Produkt von Primzahlen schreiben. Für jeden Primteiler p von m gilt $p \neq p_i$ für alle $i = 1, 2, \ldots, n$. Es gibt also eine weitere Primzahl $p \notin P$. Dies ist ein Widerspruch zur Annahme, dass P alle Primzahlen enthält.

Lösungsvorschlag zu Aufgabe 3.9

Für jedes $n \in \mathbb{N}$ mit $n \geq n_0$ definieren wir die Aussage $\mathcal{A}(n)$ als

$$\mathcal{A}(n) : \Longleftrightarrow \; n^2 - 2n - 1 > 0.$$

(a) *Induktionsanfang*: Wir zeigen, dass $\mathcal{A}(n_0)$ richtig ist. Für $n = 3$ ist

$$n^2 - 2n - 1 = 9 - 6 - 1 = 2 > 0.$$

Somit stimmt die Aussage für $n = 3$.

(b) *Induktionsschluss*:

(i) *Induktionsvoraussetzung*: Es sei $n \geq n_0$ eine natürliche Zahl und $\mathcal{A}(n)$ sei richtig. Sei also

$$n^2 - 2n - 1 > 0.$$

(ii) *Induktionsschritt* ($n \to n + 1$): Wir formulieren die Aussage $\mathcal{A}(n + 1)$ explizit:

$$(n + 1)^2 - 2(n + 1) - 1 > 0.$$

Beweis des Induktionsschritts:

$$\begin{aligned}
&(n + 1)^2 - 2(n + 1) - 1 = n^2 + 2n + 1 - 2n - 2 - 1 = \\
&n^2 - 2n - 1 + 2n + 1 - 2 = n^2 - 2n - 1 + 2n - 1 \overset{\text{Induktionsvoraussetzung}}{>} \\
&2n - 1 > 0.
\end{aligned}$$

Beachten Sie, dass für $n \geq 0$ der Term $2n - 1$ grösser als 0 ist.

Mit Theorem 3.3 folgt, dass $\mathcal{A}(n)$ für jedes $n \in \mathbb{N}$ mit $n \geq 3$ korrekt ist.

Lösungsvorschlag zu Aufgabe 3.11

Die Behauptung lässt sich sehr elegant direkt beweisen, wenn man die Zahl $n^3 - n$ als das Produkt

$$n^3 - n = (n-1)n(n+1)$$

von drei Faktoren schreibt. Da die drei Zahlen $(n-1), n$ und $(n+1)$ direkt aufeinander folgen, muss genau eine von ihnen durch 3 teilbar sein. Damit besitzt aber mindestens ein Faktor den Teiler 3. Somit ist das Produkt $(n-1)n(n+1)$ und dadurch auch $n^3 - n$ durch 3 teilbar. Dies stellt einen direkten Beweis der Aussage dar.

Nun beweisen wir die Aussage auch durch vollständige Induktion. Für jedes $n \in \mathbb{N}$ mit $n \geq 2$ definieren wir die Aussage $\mathcal{A}(n)$ als

$$\mathcal{A}(n) : \Longleftrightarrow \text{ Die Zahl } n^3 - n \text{ ist durch 3 teilbar.}$$

(a) *Induktionsanfang*: Wir zeigen, dass $\mathcal{A}(2)$ richtig ist. Für $n = 2$ ist

$$n^3 - n = 2^3 - 2 = 6$$

und 6 ist durch 3 teilbar. Somit stimmt die Aussage für $n = 2$.

(b) *Induktionsschluss*:

(i) *Induktionsvoraussetzung*: Es sei $n \geq n_0$ eine natürliche Zahl und $\mathcal{A}(n)$ sei richtig. Sei also $n^3 - n$ durch 3 teilbar.

(ii) *Induktionsschritt* $(n \to n+1)$:

$$\begin{aligned}
&(n+1)^3 - (n+1) = \\
&n^3 + 3n^2 + 3n + 1 - (n+1) = \\
&n^3 + 3n^2 + 2n = \\
&n^3 - n + 3n^2 + 3n = \\
&\underbrace{n^3 - n}_{\text{Induktionsvoraussetzung}} + \underbrace{3(n^2+n)}_{\text{ganzzahliges Vielfaches von 3}}
\end{aligned}$$

Der erste Summand ist nach Induktionsvoraussetzung durch 3 teilbar. Der zweite Summand ist als ganzzahliges Vielfaches von 3 ebenfalls durch 3 teilbar. Damit ist die Aussage auch für $n+1$ erfüllt.

Mit Theorem 3.3 folgt, dass $\mathcal{A}(n)$ für jedes $n \in \mathbb{N}$ mit $n \geq 2$ korrekt ist.

Lösungsvorschlag zu Aufgabe 3.12

Es sei $x \in \mathbb{R}$ mit $x \geq -1$ eine reelle Zahl. Für jedes $n \in \mathbb{N}$ definieren wir die Aussage $\mathcal{A}(n)$ als

$$\mathcal{A}(n) : \Longleftrightarrow \ (1+x)^n \geq 1 + nx.$$

(a) *Induktionsanfang*: Wir zeigen, dass $\mathcal{A}(0)$ richtig ist. Für $n = 0$ ist

$$(1+x)^0 = 1 \geq 1 + 0 \cdot x = 1.$$

Somit stimmt die Aussage für $n = 0$.

(b) *Induktionsschluss*:

(i) *Induktionsvoraussetzung*: Es sei $n \geq n_0$ eine natürliche Zahl und $\mathcal{A}(n)$ sei richtig. Sei also

$$(1+x)^n \geq 1 + nx.$$

(ii) *Induktionsschritt* $(n \to n+1)$: Wir formulieren die Aussage $\mathcal{A}(n+1)$ explizit:

$$(1+x)^{n+1} \geq 1 + (n+1)x.$$

Beweis des Induktionsschritts:

$$(1+x)^{n+1} = (1+x)^n(1+x) \overset{\text{Induktionsvoraussetzung und } (1+x)\geq 0}{\geq}$$

$$(1+nx)(1+x) = 1 + x + nx + nx^2 = 1 + (n+1)x + nx^2 \overset{(nx^2)\geq 0}{\geq}$$

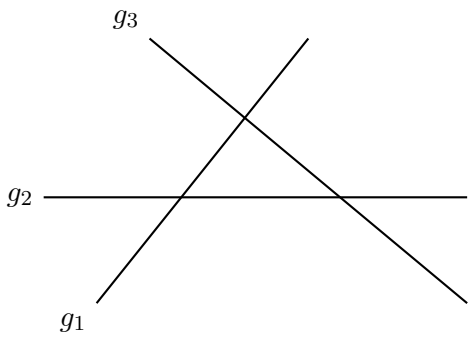
$$1 + (n+1)x.$$

Doch dies ist genau die Behauptung $\mathcal{A}(n+1)$.

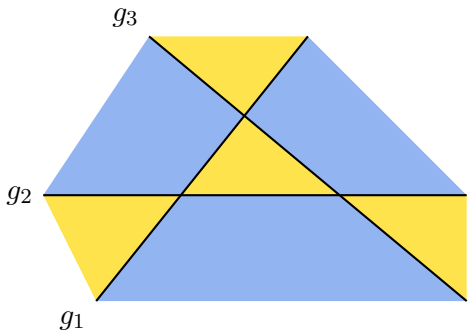
Aus dem Induktionsprinzip folgt somit, dass $\mathcal{A}(n)$ für jedes $n \in \mathbb{N}$ korrekt ist.

Lösungsvorschlag zu Aufgabe 3.13

In Abbildung 3.4 ist eine zulässige Färbung für drei Geraden skizziert.



(a) Drei verschiedene Geraden g_1, g_2 und g_3 teilen die Ebene in sieben Bereiche ein.



(b) zulässige Färbung für $n = 3$

Abbildung 3.4: zulässige Färbung für drei Geraden

Für jedes $n \in \mathbb{N}$ definieren wir die Aussage $\mathcal{A}(n)$ als

$$\mathcal{A}(n) : \Longleftrightarrow \text{ Die Bereiche, welche durch } n \text{ verschiedene Geraden in der Ebene geformt werden, besitzen eine zulässige Färbung.}$$

(a) *Induktionsanfang*: Wir zeigen, dass $\mathcal{A}(0)$ richtig ist. Für $n = 0$ haben wir gar keine Gerade und nur einen einzigen Bereich (nämlich die gesamte Ebene). Dieser Bereich kann entweder golden oder blau gefärbt werden, um eine zulässige Färbung zu erhalten.

(b) *Induktionsschluss*:

(i) *Induktionsvoraussetzung*: Es sei n eine natürliche Zahl und $\mathcal{A}(n)$ sei richtig.

(ii) *Induktionsschritt* $(n \to n+1)$:

1. Es liegen $n+1$ verschiedene Geraden in der Ebene. Wählen Sie eine beliebige dieser Geraden und nennen Sie diese g. Entfernen Sie nun g aus der Ebene.
2. Gemäss Induktionsvoraussetzung besitzen die Bereiche, welche durch die n verbleibenden Geraden geformt werden, eine zulässige Färbung. Wir nennen diese zulässige Färbung die *ursprüngliche Färbung*.
3. Legen Sie die entfernte Gerade g wieder zurück an ihre ursprüngliche Position. Die Gerade g teilt die Ebene in zwei Hälften auf.
4. Wählen Sie eine der beiden Hälften und drehen Sie alle Farben in dieser Hälfte um: Bereiche, die zuvor golden waren, sind nun blau und umgekehrt. In der anderen Hälfte wird keine Anpassung vorgenommen. Wir beweisen nun, dass die so erhaltene neue Färbung zulässig ist.
5. Wählen Sie zwei beliebige benachbarte Bereiche A und B in der neuen Färbung.

   Fall 1: Falls sich A und B die Gerade g nicht als Grenzlinie teilen, dann liegen sie auf derselben Seite von g (in derselben Hälfte). Dann waren A und B bereits in der ursprünglichen Färbung verschieden gefärbt. Entweder wurden beide Farben umgekehrt oder beide wurden nicht umgekehrt. Die Färbungen bleiben trotzdem unterschiedlich.

   Fall 2: Die Bereiche A und B teilen sich die Gerade g als Grenzlinie. Dann liegen A und B auf unterschiedlichen Seiten von g (in unterschiedlichen Hälften) und hatten dieselbe Farbe in der ursprünglichen Färbung. Bei dem Umkehren der Farben wurde die Farbe von genau einem der Bereiche A oder B geändert. Damit besitzen A und B in der neuen Färbung verschiedene Farben.

Wir haben dadurch bewiesen, dass $\mathcal{A}(n)$ für jedes $n \in \mathbb{N}$ korrekt ist.

Lösungsvorschlag zu Aufgabe 3.14

Der Nachweis der Induktionsvoraussetzung für $n = 1$ ist vollkommen korrekt. Die Argumentation im Induktionsschritt enthält aber einen entscheidenden Fehler. Der Fehler der Argumentation liegt darin, dass sie unerlaubterweise davon ausgeht, dass die Menge der $n + 1$ Pferde mindestens 3 Elemente enthält.

Zum Nachweis von $\mathcal{A}(n)$ für alle $n \in \mathbb{N}^{\times}$ muss gemäss Theorem 3.3 (ii) für **jede** natürliche Zahl $n \geq 1$ aus der Richtigkeit von $\mathcal{A}(n)$ die Richtigkeit von $\mathcal{A}(n + 1)$ folgen. Insbesondere muss also aus der Richtigkeit von $\mathcal{A}(1)$ die Richtigkeit von $\mathcal{A}(2)$ folgen. Betrachten wir also eine Menge mit nur einem Pferd und sei dieses Pferd schwarz. Wir fügen ein weiteres, diesmal ein weisses Pferd, hinzu und entfernen das schwarze Pferd. Wir haben also wieder eine Menge mit nur einem Pferd, diesmal einem weissen. Jede der beiden einelementigen Mengen enthält jeweils tatsächlich nur Pferde derselben Farbe. Fügen wir das schwarze Pferd wieder zur Menge mit dem weissen Pferd hinzu, erhalten wir eine Menge mit zwei Pferden, welche aber nicht alle von derselben Farbe sind. Was der Induktionsschritt der Aufgabenstellung also tatsächlich (und zwar korrekt) beweist, ist die Richtigkeit von $\mathcal{A}(n)$ für alle $n \in \mathbb{N}$ mit $n \geq 2$ und nicht für alle natürlichen $n \geq 1$. Dann müsste der Induktionsanfang allerdings $\mathcal{A}(2)$ nachweisen. Dies wird aber nicht gelingen, da es ja tatsächlich zwei Pferde mit unterschiedlichen Farben gibt. [5]

Lösungsvorschlag zu Aufgabe 3.15

```python
import math

def pay(n):
    combs = []
    amax = int(math.ceil(n / 4)) + 1
    bmax = int(math.ceil(n / 5)) + 1
    for a in range(amax):
        for b in range(bmax):
            if (4*a + 5*b) == n:
                combs.append((a,b))
    return combs
```

Programm 3.2: Dollar-Betrag auszahlen

Lösungsvorschlag zu Aufgabe 3.16

(a) Das entsprechende Programm ist in Listing 3.3 gegeben.

```python
import math

def klotz(n):
    combs = []
    amax = int(math.ceil(n / 7)) + 1
    bmax = int(math.ceil(n / 8)) + 1
    for a in range(amax):
        for b in range(bmax):
            if (7*a + 8*b) == n:
                combs.append((a,b))
    return combs

print(klotz(38))
print(klotz(41))
```

Programm 3.3: Bauklötze

(b) siehe Teil (a)

(c) Wir definieren für jedes $n \in \mathbb{N}$ die Aussage $\mathcal{A}(n)$ als
$\mathcal{A}(n) :\iff$ „Die Strecke der Länge n kann durch eine Kombination aus Klötzen der Längen 7 und 8 gebaut werden.“
Die Strecke $n_0 = 42$ kann wegen $42 = 6 \cdot 7 + 0 \cdot 8$ gebaut werden. Somit stimmt $\mathcal{A}(n_0) = \mathcal{A}(42)$. Zum Nachweis von $\mathcal{A}(n_0+1) = \mathcal{A}(43)$ dürften wir $\mathcal{A}(42)$ verwenden. Zum Nachweis von $\mathcal{A}(44)$ dürften wir $\mathcal{A}(42)$ und $\mathcal{A}(43)$ verwenden und zum Nachweis von $\mathcal{A}(45)$ gar $\mathcal{A}(42), \mathcal{A}(43)$ und $\mathcal{A}(44)$ und so weiter. Wir sehen jedoch sofort, dass

$$43 = 5 \cdot 7 + 1 \cdot 8, \quad 44 = 4 \cdot 7 + 2 \cdot 8, \quad 45 = 3 \cdot 7 + 3 \cdot 8$$
$$46 = 2 \cdot 7 + 4 \cdot 8, \quad 47 = 1 \cdot 7 + 5 \cdot 8, \quad 48 = 0 \cdot 7 + 6 \cdot 8,$$

und somit sind nebst $\mathcal{A}(42)$ auch $\mathcal{A}(s)$ für $s \in \{43, 44, 45, 46, 47, 48\}$ nachgewiesen.
Sei nun $n \in \mathbb{N}$ mit $n \geq 48$. Wir beweisen, dass aus der Richtigkeit von $\mathcal{A}(k)$ für $42 \leq k \leq n$ die Richtigkeit von $\mathcal{A}(n+1)$ folgt. Wir betrachten die Strecke der Länge $n+1-7$. Offensichtlich gilt $42 \leq n+1-7 \leq n$. Gemäss (starker) Induktionsannahme kann die Strecke der Länge $n+1-7$ aber durch eine Kombination aus Klötzen der Längen 7 und 8 gebaut werden. Dies ist Aussage $\mathcal{A}(n+1-7) = \mathcal{A}(n-6)$. Durch das Ansetzen eines Klotzes der Länge 7 zu dieser Kombination erhalten wir eine Strecke der Länge $n+1$.

Lösungsvorschlag zu Aufgabe 3.17

Aussage $\mathcal{A}(0)$ ist offensichtlich richtig.

Wir zeigen nun, dass aus der Richtigkeit von $\mathcal{A}(k)$ für $0 \leq k \leq n$ die Richtigkeit von $\mathcal{A}(n+1)$ folgt. Dazu betrachten wir:

$$
\begin{aligned}
& c \sum_{k=0}^{n+1} (x_k) = \\
& c \left(x_{n+1} + \sum_{k=0}^{n} (x_k) \right) \overset{\text{Distributivgesetz}}{=} \\
& cx_{n+1} + c \sum_{k=0}^{n} (x_k) \overset{\text{Induktionsvoraussetzung}}{=} \\
& cx_{n+1} + \sum_{k=0}^{n} (cx_k) = \\
& \sum_{k=0}^{n+1} (cx_k) \,.
\end{aligned}
$$

Mit dem Prinzip der starken Induktion folgt somit die Richtigkeit der Aussage $\mathcal{A}(n)$ für alle $n \in \mathbb{N}$.

Kapitel 4

Rekursion

4.1 Fakultät

Drei (unterscheidbare) Personen A, B und C stellen sich in der Mensa in einer Warteschlange an. Wie viele verschiedene Warteschlangen sind möglich? In der vordersten Position der Warteschlange platzieren wir eine der drei Personen A, B oder C. Für die vorderste Position haben wir also drei Wahlmöglichkeiten. In der mittleren Position muss genau eine der verbleibenden zwei Personen positioniert werden (zwei Wahlmöglichkeiten). Die hinterste Position muss schliesslich von der noch verbleibenden Person besetzt werden (eine Möglichkeit). Insgesamt gibt es also (gemäss den Rechengesetzen der Kombinatorik)

$$3 \cdot 2 \cdot 1 = 6$$

verschiedene Möglichkeiten, drei Personen in einer Reihe anzuordnen. Analog sieht man ein, dass es

$$n \cdot (n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

Möglichkeiten gibt, $n \in \mathbb{N}$ Personen in einer Warteschlange anzuordnen. Es gibt eine Möglichkeit, 0 Personen anzuordnen, nämlich in der „leeren Warteschlange“.

Beispiel 4.1:
Es gibt $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ Möglichkeiten, fünf Personen in einer Reihe anzuordnen.

Da das Produkt $n \cdot (n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$ häufig in Erscheinung tritt, besitzt es einen eigenen Namen: Die *Fakultät* von n und wird $n!$ oder $fak(n)$ geschrieben. So ist zum Beispiel $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

Aufgabe 4.1

Wir gehen davon aus, dass Sie bereits das Produkt 12! berechnet haben. Wie können Sie dieses Vorwissen einsetzen, um 13! relativ schnell zu berechnen? Wie lässt sich für $n \in \mathbb{N}^{\times}$ die Fakultät $n!$ aus $(n-1)!$ berechnen?

Die in Aufgabe 4.1 gemachte Beobachtung ist zwar einfach, hat aber dennoch bedeutende Implikationen. Um 5! zu berechnen, müssen wir lediglich multiplizieren können und wissen, wie 4! berechnet wird. Das Problem der Berechnung von 5! lässt sich also auf eine Multiplikation und die Berechnung von 4! reduzieren. Doch genau gleich verhält es sich mit dem Problem der Berechnung von 4!. Diese Reduktion auf immer kleinere, aber gleichartige Probleme kann so lange fortgeführt werden, bis wir

bei 0! ankommen. Betrachten Sie die folgende Berechnung:

$$\begin{aligned}
5! &= 5 \cdot \underbrace{4 \cdot 3 \cdot 2 \cdot 1}_{4!} \\
4! &= 4 \cdot \underbrace{3 \cdot 2 \cdot 1}_{3!} \\
3! &= 3 \cdot \underbrace{2 \cdot 1}_{2!} \\
2! &= 2 \cdot \underbrace{1}_{1!} \\
1! &= 1 \cdot \underbrace{0!}_{1} = 1
\end{aligned}$$

Durch „Rückwärtseinsetzen“ erhalten wir nun den Wert für 5!:

$$\begin{aligned}
1! &= 1 \cdot 0! = 1 \\
2! &= 2 \cdot 1! = 2 \cdot 1 = 2 \\
3! &= 3 \cdot 2! = 3 \cdot 2 = 6 \\
4! &= 4 \cdot 3! = 4 \cdot 6 = 24 \\
5! &= 5 \cdot 4! = 5 \cdot 24 = 120.
\end{aligned}$$

Nach diesen Betrachtungen ist plausibel, dass Folgendes eine sinnvolle Definition der ***Fakultät*** ist:

Definition 4.1 (Fakultät)**:**
Es sei n eine natürliche Zahl. Dann definieren wir die Fakultät $n!$ von n durch:

$$n! := \begin{cases} 1, & \text{falls } n = 0, \text{ (Rekursionsanfang)} \\ n(n-1)!, & \text{falls } n \geq 1. \text{ (Rekursionsschritt)} \end{cases} \tag{4.1}$$

Beachten Sie, dass Definition 4.1 der Fakultät selbst die Fakultät verwendet! Eine auf diese Weise definierte Funktion wird ***rekursiv*** genannt.

4.2 Finde den Star! (konstruktive Induktion)

Das folgende Beispiel stammt aus [6].

Definition 4.2 (finde den Star)**:**
In einem Raum befinden sich $n \in \mathbb{N}$ Personen mit $n \geq 2$ ist. Wir wollen den *Star* unter diesen n Personen finden. Ein Star ist definiert als eine Person, welche niemanden anderen kennt, jedoch von allen gekannt wird. Die einzige erlaubte *Operation*, um einen Star zu identifizieren, ist, eine Person A zu fragen:

„Kennst Du Person B?“,

wobei $A \neq B$ gilt. Das Problem ***finde den Star*** besteht nun darin, den Star unter den n Personen in dem Raum zu identifizieren oder herauszufinden, dass es gar keinen Star unter diesen n Personen gibt.

Aufgabe 4.2

Begründen Sie, warum es unter n Personen in einem Raum nicht zwei verschiedene Stars geben kann.

Wir wollen das Problem *finde den Star* mit möglichst wenigen Fragen (Operationen) finden. Eine naive Lösung besteht darin, jede Person über jede andere zu befragen („alle mit allen“) . Für $n = 4$ könnte ein Resultat einer solchen Befragung wie folgt aussehen:

-	1	2	3	4
1	-	Ja	Nein	Nein
2	Nein	-	Nein	Nein
3	Ja	Ja	-	Nein
4	Ja	Ja	Ja	-

Dabei bedeutet der Eintrag Nein, dass Person 2 die Person 4 nicht kennt und somit die Frage „Kennst Du Person 4?“ mit „Nein“ beantwortet. Hier ist Person 2 der Star. Bei n Personen werden bei diesem Vorgehen offensichtlich genau $n(n-1)$ Fragen gestellt (jede der n Personen wird zu allen $n-1$ anderen befragt).

Wir wollen versuchen, die Anzahl der Fragen zu reduzieren. Dazu wenden wir ein Vorgehen an, welches manchmal *konstruktive Induktion* genannt wird. Wir zerlegen das Problem, den Star unter n Personen zu finden, in kleinere Probleme:

- Für $n = 2$ genügen zwei Fragen.
- Sei $n > 2$: Schicke eine Person A weg. Finde nun den Star unter $n-1$ Personen (kleineres Problem). Überprüfe danach A mit $2(n-1)$ Fragen.

Doch dieses Vorgehen können wir weiter fortsetzen und dieselbe Strategie auf das Problem mit $n-1$ Personen (das kleinere Problem) anwenden. Schliesslich gelangen wir zu dem Problem mit 2 Personen, für welches zwei Fragen genügen. Insgesamt stellen wir also fest:

$$
\begin{aligned}
F(n) = \\
&2(n-1) + F(n-1) = \\
&2(n-1) + 2(n-2) + F(n-2) = \\
&2(n-1) + 2(n-2) + 2(n-3) + F(n-3) = \\
&\qquad\qquad\vdots \\
&2(n-1) + 2(n-2) + 2(n-3) + 2(n-4) + \ldots + 2 = \\
&2(1 + 2 + 3 + \ldots + (n-2) + (n-1)) \overset{\text{kleiner Gauss}}{=} \\
&2\left(\frac{n(n-1)}{2}\right) = \\
&n(n-1).
\end{aligned}
$$

Somit haben wir gegenüber der ursprünglichen naiven Lösung (befrage alle zu allen) nichts gewonnen. Zum Glück ist es aber kein grosser Aufwand unseren Ansatz der konstruktiven Induktion stark zu verbessern und somit zu retten. Die Idee der Verbesserung besteht darin, sicherzustellen, dass die Person, welche wir aus dem Raum schicken, kein Star ist.

Aufgabe 4.3

Erklären Sie, warum eine einzige Frage an eine beliebige Person im Raum stets genügt, um eine Person zu identifizieren, welche sicherlich kein Star ist.

Zum Schluss bleiben zwei Personen, von denen möglicherweise eine Person X der Star ist. Wir überprüfen mit jeder Person, die draussen ist, ob X ein Star sein kann. Mit dieser Verbesserung erhalten wir $F(2) = 2$ und $F(n) = 1 + F(n-1) + 2$ für $n \geq 3$, also insgesamt:

$$F(n) = \begin{cases} 2, & \text{falls } n = 2, \\ 1 + F(n-1) + 2, & \text{falls } n \geq 3. \end{cases} \tag{4.2}$$

Ähnlich wie bei unserer Betrachtung der Fakultät, sehen wir auch hier, dass sich die Bestimmung der Anzahl benötigter Fragen $F(n)$ bei n Personen auf die Bestimmung des kleineren (aber analogen) Problems $F(n-1)$ reduzieren lässt.

Aufgabe 4.4

Wir haben für die benötigten Fragen $F(n)$ für einen Raum mit n Personen die Beziehung in Gleichung (4.2) gefunden. Wir vermuten, dass wir den Wert für $F(n)$ durch wiederholte Reduktion auf kleinere Probleme wie folgt „entpacken“ können:

$$F(n) = 3 + F(n-1) = 2 \cdot 3 + F(n-2) = \ldots = 3 \cdot (n-2) + 2 = 3n - 4.$$

Beweisen Sie durch vollständige Induktion, dass $F(n) := 3n - 4$ tatsächlich Gleichung (4.2) erfüllt. Berechnen Sie schliesslich, wie viele Fragen wir mit diesem Verfahren bei $n = 1000$ Personen benötigen.

4.3 Rekursion in Algorithmen

In diesem Abschnitt wollen wir beginnen zu verstehen, wie Probleme rekursiv mithilfe von Programmen gelöst werden können. Zum Einstieg betrachten wir nochmals die (rekursive) Definition der Fakultät in Gleichung (4.1). Lassen Sie uns an dieser Stelle wagemutig sein! Wir „übersetzen“ die Definition direkt in die Python-Programmiersprache und lassen uns von dem Ergebnis überraschen:

```
1 def factorial(n):
2     if n == 0:
3         return 1 # Rekursionsanfang
4     else:
5         return n * factorial(n-1) # Rekursionsschritt
```

Programm 4.1: rekursive Fakultäts-Funktion

Wir haben Definition 4.1 praktisch eins zu eins „abgetippt“. Beachten Sie, dass in der Definition der Funktion `factorial` die Funktion `factorial` selbst verwendet wird. Wie und warum funktioniert dieses Vorgehen? Listing 4.2 zeigt schematisch auf, wie der Funktionsaufruf `factorial(3)` abgearbeitet wird. Beachten Sie die Ähnlichkeit zu unserer Diskussion in Abschnitt 4.1.

```
1  ## anfängliche Aufrufe ##
2
3  # Aufruf 0:
4  factorial(3)
5  return 3 * factorial(2) # Rekursionsschritt (Zeile 5)
6                  |
7                  |
8                  v
9             # Aufruf 1:
10            factorial(2)
11            return 2 * factorial(1) # Rekursionsschritt (Zeile 5)
12                             |
13                             |
14                             v
15                       # Aufruf 2:
16                       factorial(1)
17                       return 1 * factorial(0) # Rekursionsschritt (Zeile 5)
18                                          |
19                                          |
20                                          v
21                                    # Aufruf 3:
22                                    factorial(0)
23                                    return 1 # Rekursionsanfang (Zeile 2)
24
25 ## Rückwärtseinsetzen ##
26 factorial(1) = 1 * factorial(0) = 1 * 1 = 1
27 factorial(2) = 2 * factorial(1) = 2 * 1 = 2
28 factorial(3) = 3 * factorial(2) = 3 * 2 = 6
```

Programm 4.2: rekursive Berechnung der Fakultät

Der anfängliche Aufruf `factorial(3)` löst (rekursiv) in seinem `return` auf (seiner) Zeile 5 den Aufruf `factorial(2)` aus. Dieser löst rekursiv einen weiteren Funktionsaufruf aus. Dies geht so lange weiter, bis zum ersten Mal ein Aufruf den Rekursionsanfang (Zeile 2) erreicht. Danach können die noch ausstehenden `return`-Statements endlich komplettiert werden (Rückwärtseinsetzen).

Aufgabe 4.5

Seien $a \in \mathbb{R}$ und n eine natürliche Zahl. Wir definieren den Potenzausdruck a^n rekursiv durch

$$a^n := \begin{cases} 1, & \text{falls } n = 0, \text{ (Rekursionsanfang)} \\ a \cdot a^{n-1}, & \text{falls } n \geq 1. \text{ (Rekursionsschritt)} \end{cases}$$

Implementieren Sie eine Python-Funktion `def potenz(a,n)`, welche a^n gemäss dieser rekursiven Definition berechnet.

Aufgabe 4.6

Implementieren Sie eine Python-Funktion `def factorial_loop(n)`, welche

$$n! = n \cdot (n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

nicht rekursiv, sondern mithilfe einer einzigen Schleife (`for`-loop) berechnet.

Aufgabe 4.7 The One

Sie arbeiten für eine respektable Softwareentwicklungsfirma. Ihr Arbeitskollege *Thomas A. Anderson* hat folgende Python-Funktion zu Ihrem Softwareprojekt hinzugefügt:

```
1 # m und n sind natürliche Zahlen.
2 def unbekannt(m,n):
3     if m == 0:
4         return 0
5     else:
6         return unbekannt(m-1,n) + n
```

(a) Handelt es sich bei `unbekannt` um eine rekursiv oder nicht rekursiv definierte Funktion? Begründen Sie Ihre Antwort.

(b) Thomas A. Anderson hat sich schon einige Tage nicht in der Firma blicken lassen und Sie können ihn telefonisch nicht erreichen. Abgesehen von dem Kommentar in Zeile 1 hat er die Funktion `unbekannt` nicht dokumentiert und der Name der Funktion hilft uns nicht weiter. Erklären Sie genau, was die Funktion tut und geben Sie ihr einen treffenden Namen.
Tipp: Berechnen Sie die Werte
- `unbekannt(0,4)`,
- `unbekannt(1,4)`,
- `unbekannt(2,4)`,
- und `unbekannt(3,4)`

„von Hand“ und leiten Sie daraus ab, was die Funktion tut.

(c) Was ist der Rückgabewert von `unbekannt(100,50)`?

Aufgabe 4.8

In Python ist eine Liste `L` von reellen Zahlen gegeben. Schreiben Sie eine Python-Funktion `sum_rek(L)`, welche rekursiv die Summe der Zahlen in der Liste berechnet. Es kann angenommen werden, dass die Länge der Liste ≥ 1 ist.

```
Beispiel 1:
Eingabe: L = [1,3,2,10]
Ausgabe: 16

Beispiel 2:
Eingabe: L = [1,-1,2,5,4]
Ausgabe: 11
```

Aufgabe 4.9

Ein Wort heisst *Palindrom*, falls das Wort vorwärts und rückwärts genau gleich gelesen wird. Beispiele:

- neben
- Rentner
- Otto
- Lagerregal.

Das *leere Wort*, also das Wort der Länge 0, ist ebenfalls ein Palindrom. Schreiben Sie ein rekursives Python-Programm, welches entscheidet, ob ein gegebenes Wort `w` ein Palindrom ist oder nicht.

Aufgabe 4.10

Schreiben Sie eine Python-Funktion `summe(n)`, welche die Summe

$$\sum_{k=0}^{n} k := 0 + 1 + \ldots + n$$

der ersten $n+1$ natürlichen Zahlen $0, 1, \ldots, n$ rekursiv berechnet.

Aufgabe 4.11

Schreiben Sie eine Python-Funktion `produkt(n)`, welche das Produkt

$$\prod_{k=1}^{n} k^3 := 1 \cdot 8 \cdot \ldots \cdot n^3$$

für $n \in \mathbb{N}^{\times}$ rekursiv berechnet.

4.4 Fibonacci-Folge

In der zweiten Fassung des Buches *Liber abbaci* („Buch der Rechenkunst“) beschrieb der italienische Mathematiker *Leonardo da Pisa*, bekannt als *Fibonacci*, das Wachstum einer Kaninchenpopulation.

- Jedes geschlechtsreife Paar Kaninchen wirft pro Monat genau ein Paar Kaninchen (ein Weibchen und ein Männchen). Die Austragungszeit (Dauer der Schwangerschaft) dauert bei Kaninchen also immer genau einen Monat. Jeden Monat kommt also eine neue Generation von Kaninchen zur Welt.
- Ein neugeborenes Paar von Kaninchen wird erst am Ende seines ersten Lebensmonats geschlechtsreif und wirft entsprechend erst Ende seines zweiten Lebensmonats sein erstes Paar Kaninchen.
- Kein Kaninchen stirbt, kein Kanninchen verlässt das betrachtete System und kein Kanninchen wird, ausser durch Geburt, in das System hineingebracht.

Sei G_n die Anzahl der geschlechtsreifen Kaninchenpaare und g_n die Anzahl der nicht geschlechtsreifen Kaninchenpaare der Generation n für $n \in \mathbb{N}$. Beachten Sie, dass die gesamte Anzahl der Kaninchenpaare der Generation n damit der Summe $F_n := G_n + g_n$ entspricht. Betrachten wir nun die obigen Regeln für das Wachstum einer Kaninchenpopulation für alle $n \in \mathbb{N}$. Es gilt

$$G_{n+2} = G_{n+1} + g_{n+1}, \tag{4.3}$$

da die geschlechtsreifen Kaninchen G_{n+1} der Generation $n+1$ auch einen Monat später noch geschlechtsreif sein werden und die nicht geschlechtsreifen Kaninchen g_{n+1} der Generation $n+1$ einen Monat später geschlechtsreif sein werden. Völlig analog begründet man die Gleichung

$$G_{n+1} = G_n + g_n. \tag{4.4}$$

Des Weiteren gilt offensichtlich

$$g_{n+2} = G_{n+1}. \tag{4.5}$$

Wir haben also drei Gleichungen für die Population:

$$G_{n+2} = G_{n+1} + g_{n+1} \tag{4.6}$$

$$G_{n+1} = G_n + g_n \tag{4.7}$$

$$g_{n+2} = G_{n+1} \tag{4.8}$$

Das Einsetzen von Gleichung 4.7 in Gleichung 4.6 liefert:

$$\begin{gathered} G_{n+2} = G_n + g_n + g_{n+1} \\ \Longleftrightarrow \\ \underbrace{G_{n+2} + g_{n+2}}_{F_{n+2}} = \underbrace{G_{n+1} + g_{n+1}}_{F_{n+1}} + \underbrace{G_n + g_n}_{F_n}. \end{gathered}$$

Für die Gesamtzahl der Population der Kaninchenpaare gilt also

$$F_{n+2} = F_{n+1} + F_n$$

für alle $n \in \mathbb{N}$. Für Generation $n = 0$ definieren wir $G_0 := 0$ und $g_0 := 0$. Zu Beginn, also in der Generation $n = 1$, wird ein erstes Paar von Kaninchen in das System eingeführt. Dieses

erste Paar wird erst nach einem Monat geschlechtsreif. Es gilt also $G_1 = 0$ und $g_1 = 1$. Somit besteht die Generation $n = 2$ immer noch aus nur einem Paar Kaninchen: $G_2 = 1$ und $g_2 = 0$. Die Generation $n = 3$ aus zwei Paaren: $G_3 = 1$ und $g_3 = 1$. Logisch fortgeführt findet man die sogenannte ***Fibonacci-Folge***:

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8,$$
$$F_7 = 13, F_8 = 21, F_9 = 34, F_{10} = 55, F_{11} = 89, \ldots$$

Beginnend mit den „Startwerten“ $F_0 := 0$ und $F_1 := 1$ ergibt sich F_{n+2} also für jedes $n \in \mathbb{N}$ aus der Summe der beiden unmittelbaren Vorgänger F_{n+1} und F_n, also

$$F_{n+2} = F_{n+1} + F_n.$$

Definition 4.3 (Fibonacci-Folge)**:**
Die Fibonacci-Folge ist rekursiv definiert durch

$$F_n := \begin{cases} 0, & \text{falls } n = 0, \text{ (Rekursionsanfang)} \\ 1, & \text{falls } n = 1, \text{ (Rekursionsanfang)} \\ F_{n-1} + F_{n-2}, & \text{falls } n \geq 2. \text{ (Rekursionsschritt)} \end{cases}$$

Abbildung 4.1: Leonardo da Pisa (1170-1240), auch Fibonacci genannt

Aufgabe 4.12

Implementieren Sie eine Python-Funktion `def fibonacci(n)`, welche für gegebenes $n \in \mathbb{N}$ das n-te Glied der Fibonacci-Folge rekursiv berechnet. Geben Sie schliesslich die ersten 25 Glieder der Fibonacci-Folge aus.

Aufgabe 4.13

Versuchen wir F_{40} mit der Python-Funktion aus Aufgabe 4.12 zu berechnen, stellen wir fest, dass die Berechnung bereits recht lange dauert. Erklären Sie, warum die Berechnung der Glieder der Fibonacci-Folge mithilfe der Definition 4.3 sehr aufwendig ist. Wie viele Funktionsaufrufe werden für die Berechnung von $F(5)$ benötigt? Wie viele für $F(10)$?

Aufgabe 4.14

Überlegen Sie sich, wie Sie die ersten 30 Glieder der Fibonacci-Folge „von Hand“ berechnen würden. Verwenden Sie diese Intuition um eine Python-Funktion

```
def fibonacci_fast(n)
```

zu schreiben, welche für gegebenes $n \in \mathbb{N}$ das n-te Glied der Fibonacci-Folge deutlich schneller und auf nicht rekursive Weise berechnet. Berechnen Sie mithilfe dieser Funktion das Folgeglied F_{100}.

Aufgabe 4.15

Wir bezeichnen für $n \in \mathbb{N}$ mit F_n die n-te Fibonacci-Zahl. Betrachten Sie die Gleichung

$$F_{n+2} = 1 + \sum_{k=0}^{n} F_k \tag{4.9}$$

für $n \in \mathbb{N}$.

(a) Beschreiben Sie die Aussage von Gleichung (4.9) in Ihren eigenen Worten.
(b) Beweisen Sie Gleichung (4.9).

4.5 Technische Umsetzung rekursiver Programme

In diesem Abschnitt werden wir den Übergang von der rein mathematischen Betrachtung rekursiver Algorithmen zu deren technischer Umsetzung vollziehen. Dazu betrachten wir ein Beispiel eines eleganten rekursiven Algorithmus, welcher für eine gegebene natürliche Zahl n sämtliche binären Strings der Länge n ausgibt. Insbesondere sollte der Algorithmus für die gegebenen Eingaben in den folgenden Testfällen die angegebenen Ausgaben erzeugen:

```
TESTFALL 0
Eingabe: n = 0
Ausgaben:

(Es wird eine leere Zeile (leerer String) ausgegeben.)
TESTFALL 1
Eingabe: n = 1
Ausgaben:
0
1
TESTFALL 2
Eingabe: n = 2
Ausgaben:
00
01
10
11
TESTFALL 3
Eingabe: n = 3
Ausgaben:
000
001
010
011
100
101
110
111
```

Programm 4.3: binäre Strings rekursiv ausgeben

Sie sind gerne eingeladen, an dieser Stelle vorerst nicht weiterzulesen und den Algorithmus selbst zu schreiben.

Unser Vorschlag für einen entsprechenden rekursiven Algorithmus ist in Listing 4.4 gegeben.

```
1 def binaryStrings(n, w = ''):
2     if n == 0:
3         print(w)
4         return
5     binaryStrings(n-1, w + '0')
6     binaryStrings(n-1, w + '1')
7     #return # optional
```

Programm 4.4: binaryStrings

Zur Vereinfachung und Konkretisierung der Beschreibung der technischen Realisation rekursiver Algorithmen werden wir sämtliche Betrachtungen dieses Abschnitts auf den Algorithmus in Listing 4.4 beziehen. Der Algorithmus beginnt mit dem leeren String und baut alle gesuchten Strings durch systematisches „Anhängen“ von Nullen und Einsen auf. Zur Abkürzung schreiben wir anstelle von `binaryString(...)` im Folgenden `f(...)`.

Betrachten wir den Aufruf `f(2,'')`, um alle Strings der Länge 2 auszugeben. Die Arbeitsschritte, welche der Funktionsaufruf `f(2,'')` einleitet, sind in Abbildung 4.2 dargestellt. Die grünen Knoten stellen Funktionsaufrufe (Aufrufe von `f`) dar. Die goldenen Knoten stellen Aufrufe der Python-Funktion `print` dar. Beim Schritt mit Nummer 0 erfolgt der anfängliche Aufruf `f(2,'')`. In diesem Aufruf wird in Programmzeile 5 der Funktionsaufruf `f(1,'0')` ausgelöst (Schritt 1). Beachten Sie, dass der ursprüngliche Aufruf `f(2,'')` seine Arbeit noch nicht abgeschlossen hat! In Programmzeile 5 des Funktionsaufrufs `f(1,'0')` wird nun der Aufruf `f(0,'00')` ausgelöst. In diesem Aufruf ist die `if`-Bedingung auf Programmzeile 2 erfüllt, sodass die Ausgabe `print('00')` erfolgt (Schritt 3) und der `return`-Aufruf auf Programmzeile 4 erfolgt. Der Aufruf `f(0,'00')` ist beendet und er springt zurück zum Aufruf `f(1,'0')` (Schritt 4). Der Funktionsaufruf `f(1,'0')` kann nun (endlich) zu Programmzeile 6 gelangen und den Aufruf `f(0,'01')` auslösen (Schritt 5).

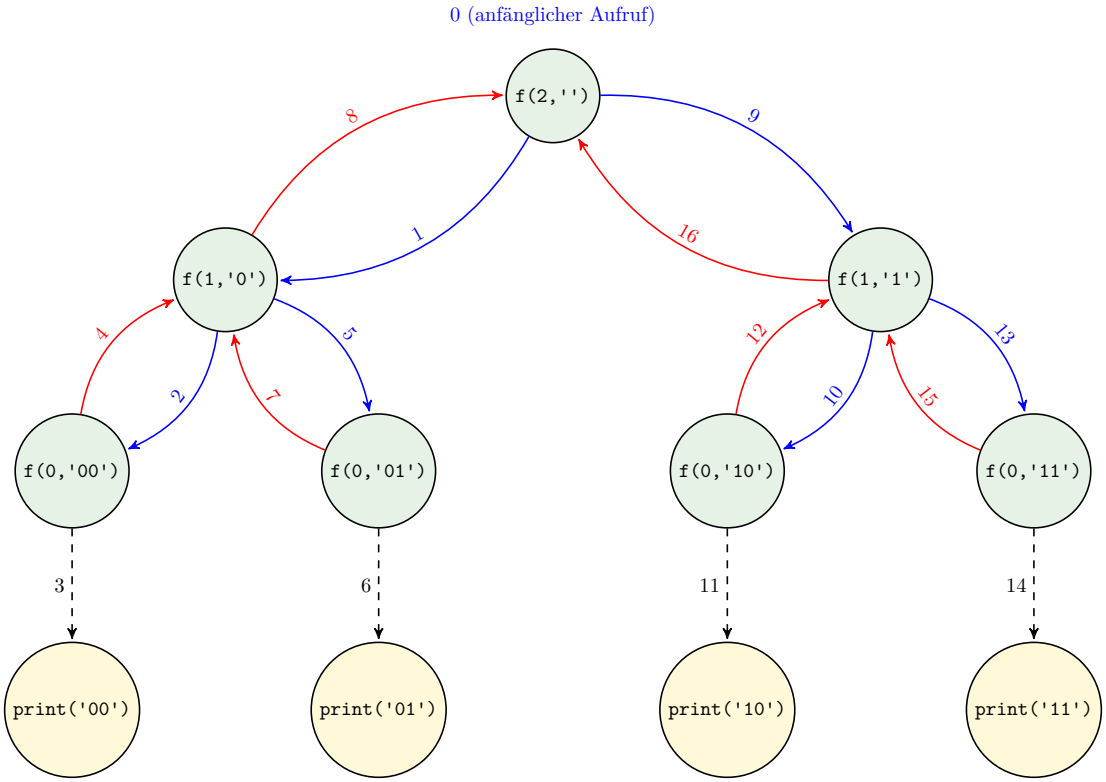


Abbildung 4.2: schematische Darstellung der Arbeitsschritte zur rekursiven Ausgabe aller binären Strings der Länge 2

Erst nachdem auch `f(0,'01')` seine Arbeit beendet hat (Schritte 6 und 7), erreicht der Aufruf `f(1,'0')` seine Programmzeile 7 und springt zurück zum ursprünglichen Aufruf `f(2,'')` (Schritt 7). Dieser erreicht nun Programmzeile 6 und ruft `f(1,'1')` auf (Schritt 9). Die Schritte in dieser „rechten“ Hälfte von Abbildung 4.2 lassen sich nun analog beschreiben. Erst nach der Rückgabe des Aufrufs `f(1,'1')` in Schritt 16 kann schliesslich der ursprüngliche Aufruf `f(2,'')` seine Arbeit beenden.

Zusammenfassung 4.1:
Wie werden rekursive Programme in Computern realisiert? In Computern wird für jeden Funktionsaufruf (Prozedur) ein Abschnitt im Speicher angelegt. Dieser Speicherabschnitt wird *Stack-Frame* genannt. Darin darf die Funktion Speicherplatz zum Beispiel für lokale Variablen belegen. Deshalb benötigen rekursive Programme mit zahlreichen rekursiven Aufrufen viel Platz im Speicher.
Ruft eine Prozedur `A` eine andere Prozedur `B` auf, so wird im Stack-Frame von Prozedur `B` eine sogenannte *Rücksprung-Adresse* gespeichert. Diese Adresse gibt an, wo im Speicher die Prozedur `A` beginnt. Dadurch wird ermöglicht, dass Prozedur `B` zu Prozedur `A` „zurückspringen“ kann (*jump and link*). Mehr Details bezüglich der Funktionsweise von Computern und rekursiven Programmen finden Sie in den hervorragenden Texten [7] und [8].

4.6 Lösungen der Aufgaben

Lösungsvorschlag zu Aufgabe 4.1

Wir beobachten, dass $13! = 13 \cdot 12!$ gilt. Somit müssen wir $12!$ lediglich mit 13 multiplizieren um $13!$ zu erhalten. Für $n \in \mathbb{N}^\times$ gilt allgemein

$$n! = n \cdot (n-1)!.$$

Lösungsvorschlag zu Aufgabe 4.2

Angenommen es gäbe zwei Stars S_1 und S_2 in dem Raum. Da S_2 ein Star ist, wird S_2 von allen anderen gekannt. Insbesondere kennt S_1 die Person S_2. Damit kann aber S_1 selbst kein Star sein. Dies ist ein Widerspruch zu unserer Annahme, dass sowohl S_1 als auch S_2 Stars sind.

Lösungsvorschlag zu Aufgabe 4.3

Frage eine beliebige Person A im Raum, ob sie eine andere beliebige Person B kennt.

- Falls A die Person B kennt $\Rightarrow$ A ist kein Star.
- Falls A die Person B nicht kennt $\Rightarrow$ B ist kein Star.

In jedem der Fälle haben wir mit einer einzigen Frage eine Person identifiziert, welche sicherlich kein Star ist.

Lösungsvorschlag zu Aufgabe 4.4

Der Induktionsanfang ist klar, denn für $n = 2$ benötigen wir genau zwei Fragen. Sei nun die Behauptung für ein $n \in \mathbb{N}$ mit $n \geq 2$ wahr. Dann gilt

$$F(n+1) = F(n) + 3 = 3n - 4 + 3 = 3n - 1,$$

doch dies ist genau die Behauptung, denn $3(n+1) - 4 = 3n + 3 - 4 = 3n - 1$.

Gemäss der soeben bewiesenen Formel sind mit unserem Vorgehen genau $F(1000) = 3 \cdot 1000 - 4 = 2996$ Fragen notwendig. Beachten Sie, dass wir in keinster Weise behaupten, dass dieses Vorgehen optimal ist.

Lösungsvorschlag zu Aufgabe 4.5

```
def potenz(a,n):
    if n == 0:
        return 1
    else:
        return a * potenz(a,n-1)
```

Programm 4.5: rekursive Potenz-Funktion

Lösungsvorschlag zu Aufgabe 4.6

```
def factorial_loop(n):
    factorial = 1
    if n == 0:
        return factorial
    else:
        for k in range(1,n+1):
            factorial *= k
        return factorial
```

Programm 4.6: rekursive Potenz-Funktion

Lösungsvorschlag zu Aufgabe 4.7

(a) Die Python-Funktion ist rekursiv definiert, da sie sich selbst aufruft.
(b) Die unbekannte Funktion multipliziert die Zahlen m und n, realisiert also die Operation $m \cdot n$. In der Tat sieht man, dass die Funktion so lange n addiert, bis $m = 0$ ist. Ein treffender Name für die Funktion wäre zum Beispiel `mult(m,n)`.
(c) Der Rückgabewert ist $100 \cdot 50 = 5000$.

Lösungsvorschlag zu Aufgabe 4.8

```
def sum_rek(L):
    if len(L) == 1:
        return L[0]
    else:
        return sum_rek(L[:-1]) + L[-1]
```

Lösungsvorschlag zu Aufgabe 4.9

```
def check_palindrom(w):
    w = w.upper()
    if len(wort) <= 1:
        return True
    elif w[0] != w[-1]:
        return False
    return check_palindrom(w[1:-1])
```

oder noch kürzer:

```
def check_palindrom(w):
    return (True if len(w) <= 1 else (check_palindrom(w[1:-1]) if w[0].lower() == w[-1].lower()
    else False))
```

Lösungsvorschlag zu Aufgabe 4.10

```
def summe(n):
    if n == 0:
        return 0
    else:
        return n + summe(n-1)
```

Programm 4.7: rekursive Berechnung einer Summe

Lösungsvorschlag zu Aufgabe 4.11

```
def produkt(n):
    if n == 1:
        return 1
    else:
        return n**3 * produkt(n-1)
```

Programm 4.8: rekursive Berechnung eines Produkts

Lösungsvorschlag zu Aufgabe 4.12

```python
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Programm 4.9: rekursive Berechnung der Fibonacci-Folge

Die ersten 25 Glieder der Fibonacci-Folge lauten:

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
```

Programm 4.10: die ersten 25 Glieder der Fibonacci-Folge

Lösungsvorschlag zu Aufgabe 4.13

Die rekursiv definierte Python-Funktion aus Aufgabe 4.12 berechnet viele Folgeglieder mehrfach und arbeitet somit sehr „verschwenderisch“. Beispielsweise werden bei der Berechnung von F_{40} die Berechnungen von F_{39} und F_{38} aufgerufen. Doch zur Berechnung von F_{39} muss nochmals F_{38} berechnet werden und so weiter. In Abbildung 4.3 wird gezeigt, welche Funktionsaufrufe für die rekursive Berechnung von $F(5)$ benötigt werden. Beachten Sie, dass beispielsweise der Wert fib(2) dreimal berechnet wird. Es ist klar, dass die Anzahl der benötigten Funktionsaufrufe (und somit sicherlich auch die benötigten Rechenoperationen) zur Berechnung von $F(n)$ mit wachsendem n stark ansteigt. Jede Fibonacci-Zahl, ausser $F(1)$ und $F(2)$, löst jeweils zwei Funktionsaufrufe aus. Die genaue Anzahl $A(n)$ der benötigten Funktionsaufrufe (dies ist nicht offensichtlich) ist gegeben durch

$$A(n) = 2F(n+1) - 1.$$

In Kapitel 5 werden Sie sehen, dass die Fibonacci-Folge exponentiell wächst.

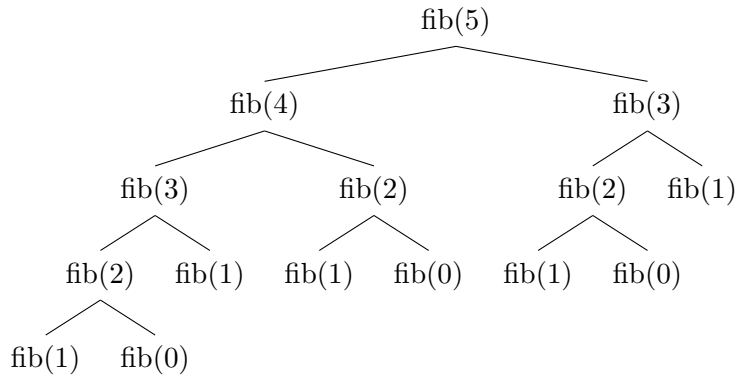


Abbildung 4.3: Die Anzahl Funktionsaufrufe für $F(5)$ ist 15.

Lösungsvorschlag zu Aufgabe 4.14

Die folgende Python-Funktion startet zur Berechnung von F_n bei den Startwerten F_0 und F_1 und berechnet „von unten her“ nacheinander (man spricht in diesem Zusammenhang von *iterativer Berechnung*) die Nachfolger. Dabei werden keine Berechnungen „verschwendet“.

```python
def fibonacci_fast(n):
    # Startwerte
    f0 = 0
    f1 = 1
    if n == 0:
        return f0
    elif n == 1:
        return f1

    for k in range(n-1):
        f2 = f1 + f0
        # Update
        f0 = f1
        f1 = f2
    return f2
```

Programm 4.11: iterative Berechnung der Fibonacci-Folge

Mithilfe dieser Funktion berechnen wir (in sehr kurzer Zeit)

$$F_{100} = 354224848179261915075.$$

Lösungsvorschlag zu Aufgabe 4.15

(a) Die $(n+2)$-te Fibonacci-Zahl ist um 1 grösser als die Summe der Fibonacci-Zahlen $F_0, \ldots, F_n$.

(b) Wir beweisen die Behauptung durch vollständige Induktion. Für $n = 0$ haben wir

$$1 + \sum_{k=0}^{0} F_k = 1 + F_0 = 1 + 0 = 1 = F_2.$$

Wir gehen nun von der Induktionsvoraussetzung

$$F_{n+1} = 1 + \sum_{k=0}^{n-1} F_k$$

aus und finden

$$\begin{aligned} &F_{n+2} = \\ &F_n + F_{n+1} = \\ &F_n + \left(1 + \sum_{k=0}^{n-1} F_k\right) = \\ &1 + \sum_{k=0}^{n} F_k. \end{aligned}$$

Kapitel 5

Binäre Strings ohne aufeinanderfolgende Einsen

In Listing 4.4 haben wir bereits eine Python-Funktion geschrieben, welche rekursiv alle binären Strings einer gegebenen Länge n ausgibt. Es wird sich ein interessanter und überraschender Zusammenhang zur Fibonacci-Folge ergeben, wenn wir nicht alle binären Strings der Länge n ausgeben, sondern nur die binären Strings der Länge n, welche nicht das Muster 11 (Eins-Eins) enthalten. Wir interessieren uns also nur für die binären Strings, welche nicht zwei aufeinanderfolgende Einsen enthalten.

Aufgabe 5.1

Ändern Sie Listing 4.4 dahingehend ab, dass für gegebenes $n \in \mathbb{N}$ genau die binären Strings der Länge n ohne aufeinanderfolgende Einsen ausgegeben werden. Beginnen Sie auch hier wieder mit dem leeren String und bauen Sie die gesuchten Strings rekursiv auf. Geben Sie Ihrer Funktion die Signatur

`print_binary_without_11(n, w = '').`

```
TESTFALL 0
Eingabe: n = 0, Ausgaben:

(Es wird eine leere Zeile (leerer String) ausgegeben.)
TESTFALL 1
Eingabe: n = 1, Ausgaben:
0
1
TESTFALL 2
Eingabe: n = 2, Ausgaben:
00
01
10
TESTFALL 3
Eingabe: n = 3, Ausgaben:
000
001
010
100
101
```

Programm 5.1: binäre Strings ohne 11 rekursiv ausgeben

5.1 Anzahl der binären Strings ohne 11

Sei n eine natürliche Zahl. Wir bezeichnen mit L_n die Menge aller binären Strings der Länge n, die nicht den Teilstring 11 enthalten, und setzen $N(n) = |L_n|$. Wir wollen $N(n)$, also die Anzahl der Elemente in L_n, rekursiv bestimmen. Offensichtlich gilt $N(0) = 1$, denn nur der leere String hat die Länge 0 und dieser enthält nicht den Teilstring 11. Des Weiteren gilt $N(1) = 2$, da die Strings 0 und 1 beide nicht den Teilstring 11 enthalten.

Aufgabe 5.2

Vervollständigen Sie Tabelle 5.1.

n	$N(n)$
0	1
1	2
2	3
3	5
4	
5	

Tabelle 5.1: Tabelle für $N(n)$

Betrachten wir nun einen binären String w der Länge $n+1$ mit $n \geq 1$. Angenommen w liegt in L_{n+1}, dann enthält offensichtlich auch keiner der Teilstrings von w das Muster 11. Dann können wir $w \in L_{n+1}$ schreiben als

$$w = xab,$$

wobei $x \in L_{n-1}$ und $a, b \in \{0, 1\}$.

Aufgabe 5.3

Wir nehmen an, dass die Anzahlen $N(n)$ und $N(n-1)$ für ein n mit $n \geq 1$ bereits bekannt sind. Wir haben soeben begründet, dass wir w in L_{n+1} schreiben können als

$$w = xab,$$

wobei $x \in L_{n-1}$ und $a, b \in \{0, 1\}$. Unterscheiden Sie zwei Fälle $b = 0$ und $b = 1$ für das Symbol b. Drücken Sie $N(n+1)$ durch $N(n)$ und $N(n-1)$ aus.

5.2 Explizite und rekursive Darstellungen von Folgen

Wir betrachten die rekursiv definierte Folge

$$a_0 = -1,$$
$$a_{n+1} = a_n + 4.$$

Ein Folgenglied mit grösserem Index, wie zum Beispiel a_{5000}, zu berechnen, ist recht mühsam. In dieser rekursiven Darstellung müssten für die Berechnung von a_{5000} nämlich alle Glieder $a_1, a_2, \ldots, a_{4999}$ zuerst bestimmt werden. Wie können wir späte Folgenglieder (mit hohen Indizes) effizienter berechnen? Die folgende Aufgabe 5.4 wird sich mit dieser Frage beschäftigen.

Aufgabe 5.4

Finden Sie eine „Formel“, welche a_n mit lediglich einer Multiplikation und einer Addition berechnet, ohne zuerst die Vorgänger $a_1, a_2, \ldots, a_n$ bestimmen zu müssen. Berechnen Sie mithilfe dieser Formel das Folgenglied a_{5000}.

Die in Aufgabe 5.4 gefundene (nicht rekursive) Darstellung von a_n wird ***explizite Darstellung*** von a_n genannt.

Aufgabe 5.5

Beweisen Sie durch vollständige Induktion, dass $a_n = n^2 - 2n$ eine explizite Darstellung der rekursiv definierten Folge

$$a_0 = 0, \quad a_n = a_{n-1} + 2n - 3$$

für $n \in \mathbb{N}^\times$ ist.

Die rekursive Darstellung Gleichung (5.3) ist nicht geeignet, um $N(n)$ für grosse Werte von n zu berechnen. Die Fibonacci-Folge wurde bereits im Jahr 1202 von Leonardo da Pisa verwendet, um das Wachstum einer Kaninchenpopulation zu beschreiben. Dennoch gelang es (höchstwahrscheinlich) erst in der ersten Hälfte des 18. Jahrhunderts, eine explizite Darstellung dieser wichtigen Folge zu finden. Diese Darstellung zu finden ist also alles andere als einfach. Diese explizite Darstellung ist als *Formel von Moivre-Binet* bekannt und besagt

$$F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}, \tag{5.1}$$

wobei $\varphi := \frac{1+\sqrt{5}}{2}$.

Gleichung (5.1) enthält die irrationale Zahl $\sqrt{5}$. Ist es nicht erstaunlich, dass $F(n)$ für alle $n \in \mathbb{N}$ eine natürliche Zahl ist? In der anspruchsvollen Aufgabe 5.7 haben Sie die Gelegenheit zu beweisen, dass die Formel von Moivre-Binet tatsächlich die Fibonacci-Zahlen berechnet (und somit ausschliesslich natürliche Zahlen generiert).

Aufgabe 5.6

Berechnen Sie $F(0)$ und $F(1)$ mithilfe von Gleichung (5.1). Überzeugen Sie sich, dass $F(0)$ und $F(1)$ die ersten beiden Fibonacci-Zahlen und somit insbesondere natürliche Zahlen sind.

Aufgabe 5.7

(!) Beweisen Sie durch starke Induktion, dass die n-te Fibonacci-Zahl F_n durch den Ausdruck $F(n)$ in Gleichung (5.1) gegeben ist.

Hinweis:

- Definieren Sie zuerst

$$\alpha := 1 - \varphi$$

und beachten Sie, dass

$$\alpha = 1 - \varphi = 1 - \frac{1+\sqrt{5}}{2} = \frac{2-1-\sqrt{5}}{2} = \frac{1-\sqrt{5}}{2}.$$

- Beweisen und verwenden Sie nun die beiden Gleichungen

$$\varphi^2 = 1 + \phi,$$
$$\alpha^2 = 1 + \alpha.$$

Bemerkung 5.1 (Moivre-Binet)**:**
Die Formel von Moivre-Binet enthält Terme, welche Computer aufgrund ihrer (nur) endlichen Arithmetik nicht exakt darstellen können. Eine einfache Beobachtung macht die Formel aber auch für die Berechnung in endlicher Arithmetik gut zugänglich. Wir schreiben zuerst

$$F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} = \frac{\varphi^n}{\sqrt{5}} - \frac{(1-\varphi)^n}{\sqrt{5}}.$$

Die Fibonacci-Zahl F_n unterscheidet sich von der Zahl $\frac{\varphi^n}{\sqrt{5}}$ also lediglich um den Term $\frac{(1-\varphi)^n}{\sqrt{5}}$. Doch wie gross ist dieser Term? Für den Exponenten $n = 0$ reduziert er sich auf

$$\left|\frac{1}{\sqrt{5}}\right| < \frac{1}{2}.$$

Für den Exponenten $n = 1$ reduziert sich der Term auf

$$\left|\frac{1-\varphi}{\sqrt{5}}\right|.$$

Wegen $|1 - \varphi| < 1$ gilt

$$\left|\frac{1-\varphi}{\sqrt{5}}\right| < \frac{1}{\sqrt{5}} < \frac{1}{2}.$$

Was ändert sich aber, wenn wir allgemeine Exponenten $n \in \mathbb{N}$ zulassen? Tatsächlich gilt: Je grösser der Exponent ist, desto kleiner ist der Term! Aufgrund der Abschätzung $|1 - \varphi| < 1$ wissen wir nämlich, dass die geometrische Folge

$$\left|\frac{(1-\varphi)^n}{\sqrt{5}}\right| = \frac{1}{\sqrt{5}}\left|(1-\varphi)^n\right| = \frac{1}{\sqrt{5}}\left|1-\varphi\right|^n$$

strikt monoton fallend ist. Damit gilt also für alle natürlichen Exponenten $n \in \mathbb{N}$

$$\left|\frac{(1-\varphi)^n}{\sqrt{5}}\right| < \frac{1}{2}.$$

Wir haben somit nachgewiesen, dass

$$\frac{\varphi^n}{\sqrt{5}} - \frac{1}{2} < \underbrace{\frac{\varphi^n}{\sqrt{5}} - \frac{(1-\varphi)^n}{\sqrt{5}}}_{F_n} < \frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \iff$$

$$-\frac{1}{2} < \quad F_n - \frac{\varphi^n}{\sqrt{5}} \quad < \frac{1}{2},$$

oder anders geschrieben:

$$\left| F_n - \frac{\varphi^n}{\sqrt{5}} \right| < \frac{1}{2}.$$

Der Abstand von $\varphi^n/\sqrt{5}$ zu der ganzen Zahl F_n ist also kleiner als $1/2$. Es liegt damit keine ganze Zahl so nahe bei $\varphi^n/\sqrt{5}$ wie F_n. Die n-te Fibonacci-Zahl F_n entspricht somit $\varphi^n/\sqrt{5}$, gerundet auf die nächste ganze Zahl:

$$F_n = \left[\frac{\varphi^n}{\sqrt{5}} \right] \quad \text{für alle } n \in \mathbb{N}. \tag{5.2}$$

Damit ist nun auch klar ersichtlich, dass die Fibonacci-Folge exponentiell wächst!

Aufgabe 5.8

Verwenden Sie die Formel von Moivre-Binet, um eine explizite Darstellung von $N(n)$ zu finden und schreiben Sie eine Python-Funktion `def N(n)`, welche $N(n)$ für gegebenes $n \in \mathbb{N}$ berechnet. Verwenden Sie dazu Gleichung (5.2).

5.3 Lösungen der Aufgaben

Lösungsvorschlag zu Aufgabe 5.1

```python
def print_binary_without_11(n, w = ''):
    if n == 0:
        print(w)
        return

    if len(w) == 0 or  w[-1] != '1':
        print_binary_without_11(n-1, w + '0')
        print_binary_without_11(n-1, w + '1')
    else:
        print_binary_without_11(n-1, w + '0')
```

Programm 5.2: binäre Strings ohne 11

Lösungsvorschlag zu Aufgabe 5.2

n	$N(n)$
0	1
1	2
2	3
3	5
4	8
5	13
6	21
7	34
8	55

Tabelle 5.2: ausgefüllte Tabelle für $N(n)$

Lösungsvorschlag zu Aufgabe 5.3

Wir unterscheiden zwei mögliche Fälle für das Symbol b.

- Falls $b = 0$, dann hat w die Form $w = xab = xa0$. Da $b = 0$ ist, kann a sowohl 0 als auch 1 sein (es gibt keine Einschränkung für a). Dann hat also w die Form

$$w = \underbrace{xa}_{=:y} 0 = y0,$$

  wobei y ein beliebiger String aus L_n sein darf, und L_n enthält $N(n)$ Elemente.
- Falls $b = 1$, dann muss $a = 0$ gelten. Für x ist dann aber ein beliebiger String aus L_{n-1} möglich und L_{n-1} enthält $N(n-1)$ Elemente.

Da für b genau zwei Fälle möglich sind und diese Fälle keine „Überlappung“ aufweisen, können wir die jeweiligen Anzahlen addieren und erhalten die Gleichung

$$N(n+1) = N(n) + N(n-1). \tag{5.3}$$

Dies sieht aber genauso aus wie die rekursive Definition der Fibonacci-Folge! Aufgrund der bereits gefundenen Anfangsbedingungen $N(0) = 1$ und $N(1) = 2$ muss die gewöhnliche Fibonacci-Folge „verschoben“ werden und wir sehen

$$N(n) = F(n+2).$$

Lösungsvorschlag zu Aufgabe 5.4

Wir bemerken, dass das nachfolgende Folgenglied um 4 grösser ist als sein Vorgänger. Damit ist a_n genau $4n$ grösser als a_0 und wir erhalten

$$a_n = a_0 + 4n = 4n - 1.$$

Nun finden wir $a_{5000} = 4 \cdot 5000 - 1 = 19999$.

Lösungsvorschlag zu Aufgabe 5.5

Wegen $a_0 = 0^2 - 2 \cdot 0 = 0$ ist der Induktionsanfang gezeigt. Angenommen die Aussage gilt für n. Wir zeigen, dass sie auch für $n+1$ gilt, dass also $a_{n+1} = (n+1)^2 - 2(n+1) = n^2 - 1$. In der Tat ist

$$a_{n+1} = a_n + 2(n+1) - 3 = n^2 - 2n + 2(n+1) - 3 = n^2 - 1.$$

Lösungsvorschlag zu Aufgabe 5.6

Wir berechnen

$$F(0) = \frac{\varphi^0 - \alpha^0}{\sqrt{5}} = 0 = F_0,$$
$$F(1) = \frac{\varphi^1 - \alpha^1}{\sqrt{5}} = \frac{\varphi - (1-\varphi)}{\sqrt{5}} = \frac{2\varphi - 1}{\sqrt{5}} = \frac{1 + \sqrt{5} - 1}{\sqrt{5}} = 1 = F_1.$$

Lösungsvorschlag zu Aufgabe 5.7

Wir beweisen zuerst die Gleichungen aus dem Hinweis:

$$\varphi^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{6+2\sqrt{5}}{4} = 1 + \frac{1+\sqrt{5}}{2} = 1 + \varphi,$$
$$\alpha^2 = \left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{6-2\sqrt{5}}{4} = 1 + \frac{1-\sqrt{5}}{2} = 1 + \alpha.$$

In Aufgabe 5.6 haben wir bereits die Korrektheit der Aussage für $n = 0$ und $n = 1$ gezeigt. Sei nun die Aussage für ein $n \in \mathbb{N}^\times$ wahr.

$$\begin{aligned} F_{n+1} &\overset{\text{Definition von } F_{n+1}}{=} F_n + F_{n-1} = \\ &\frac{\varphi^n - \alpha^n}{\sqrt{5}} + \frac{\varphi^{n-1} - \alpha^{n-1}}{\sqrt{5}} = \\ &\frac{\varphi^n + \varphi^{n-1} - \left(\alpha^n + \alpha^{n-1}\right)}{\sqrt{5}} = \\ &\frac{(\varphi+1)\,\varphi^{n-1} - \left((\alpha+1)\,\alpha^{n-1}\right)}{\sqrt{5}} = \\ &\frac{\varphi^2\varphi^{n-1} - \alpha^2\alpha^{n-1}}{\sqrt{5}} = \\ &\frac{\varphi^{n+1} - \alpha^{n+1}}{\sqrt{5}} = \frac{\varphi^{n+1} - (1-\varphi)^{n+1}}{\sqrt{5}} = F(n+1) \end{aligned}$$

Lösungsvorschlag zu Aufgabe 5.8

Wir haben bereits festgestellt, dass

$$N(n) = F(n+2) = \frac{\varphi^{n+2} - (1-\varphi)^{n+2}}{\sqrt{5}}$$

mit $\varphi := \frac{1+\sqrt{5}}{2}$ gilt. Das entsprechende Programm lautet:

```
import math

def N(n):
    sqrt5 = math.sqrt(5)
    phi = (1 + sqrt5)/2
    return int(round(phi**(n+2)/sqrt5))
```

Kapitel 6

Sortieren (*)

Eines der häufigsten algorithmischen Probleme ist das *Sortieren*.

Definition 6.1 (Sortierproblem)**:**
mydefinition:Sortierproblem Sei S eine Menge, deren Elemente durch die Relation $\leq$ verglichen werden können ($\leq$ heisst dann *totale Ordnung* auf S). Das ***Sortierproblem*** lautet dann:
Eingabe: Eine endliche Folge von $n \in \mathbb{N}^{\times}$ Elementen $a_1, a_2, \ldots, a_n$ aus S.
Ausgabe: Eine Umordnung $a'_1, a'_2, \ldots, a'_n$ der Eingabe, sodass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$ (die Elemente der Eingabe sind nun aufsteigend sortiert).

Abbildungen 6.1 und 6.2 zeigen zwei einfache Beispiele von Eingaben und entsprechenden Ausgaben des Sortierproblems.

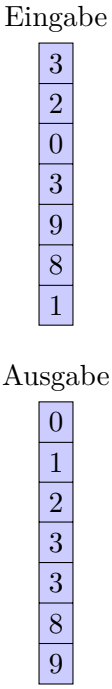


Abbildung 6.1: Eingabe und Ausgabe eines kleinen Sortierproblems mit ganzen Zahlen

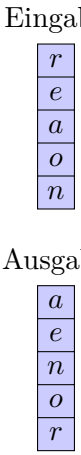


Abbildung 6.2: Eingabe und Ausgabe eines kleinen Sortierproblems mit Kleinbuchstaben des lateinischen Alphabets. Dabei wurde die übliche alphabetische Ordnung $a \leq b \leq \ldots \leq z$ verwendet.

6.1 Sortieren mit dem *merge sort* Algorithmus

Natürlich ist das Bedürfnis nach schnellen Algorithmen besonders gross, falls diese Probleme lösen, die zeitkritisch sind und häufig auftreten. Mittlerweile sind viele Sortieralgorithmen bekannt. Ein berühmter Vertreter dieser Algorithmen ist *merge sort*. Der Algorithmus *merge sort* arbeitet rekursiv. Dabei macht er insbesondere von der Tatsache Gebrauch, dass sich zwei sortierte Listen recht effizient zu einer ebenfalls sortierten Liste zusammenfügen lassen. Es seien also zwei jeweils **bereits sortierte** Listen `L` und `R` gegeben. Diese Listen besitzen die Längen (Anzahl von Elementen) `len(L)` und `len(R)`. Es ist nun nicht besonders aufwändig, die beiden Listen `L` und `R` zu einer neuen **sortierten** Liste `M` der Länge `len(L) + len(R)` zu vereinen (englisch: *to merge*).

Aufgabe 6.1

Angenommen wir haben zwei jeweils **bereits sortierte** Listen `L` und `R` gegeben. Schreiben Sie eine Python-Funktion `merge(L, R)`, welche `L` und `R` zu einer einzigen sortierten Liste vereint. Ihre Funktion soll unbedingt Gebrauch von der Tatsache machen, dass `L` und `R` bereits sortiert sind.

```
Testfall 0
Eingabe: L = [8, 12, 17], R = []
Ausgabe: [8, 12, 17]

Testfall 1
Eingabe: L = [2, 5, 13], R = [1, 2, 3, 7, 15, 20]
Ausgabe: [1, 2, 2, 3, 5, 7, 13, 15, 20]

Testfall 2
Eingabe: L = [1,10,100,1000], R = [0,5,50]
Ausgabe: [0, 1, 5, 10, 50, 100, 1000]
```

Programm 6.1: Testfälle für die Funktion `merge`

Ihre Funktion braucht nicht zu überprüfen, ob `L` und `R` tatsächlich sortiert sind.

Aufgabe 6.2

Analysieren Sie den Algorithmus, welchen Sie in Aufgabe 6.1 geschrieben haben. Welche Laufzeit hat dieser Algorithmus in Abhängigkeit von `n = len(L) + len(R)`?

Der *merge sort* Algorithmus teilt eine zu sortierende Liste rekursiv so lange in zwei Teile, bis nur noch Listen der Länge 1 vorliegen. Offensichtlich ist jede Liste der Länge 1 bereits sortiert. Schliesslich werden bereits sortierte Listen mithilfe des *merge* Algorithmus zu der gesuchten sortierten Liste zusammengefügt. Mithilfe der bereits in Aufgabe 6.1 erstellten Routine *merge* lässt sich der berühmte *merge sort* Algorithmus in nur wenigen Zeilen in Python beschreiben. Eine mögliche Implementation ist in Listing 6.2 gegeben.

```
1  def merge_sort(A):
2      # sortiert die Liste A
3      if len(A) == 1: # Rekursionsanfang
4          return A # Listen der Länge 1 sind schon sortiert.
5
6      # teile A in linke Hälfte und rechte Teile
7      if len(A) % 2 == 0: # falls len(A) gerade ist
8          middle = len(A) // 2
9      else: # falls len(A) ungerade ist
10         middle = (len(A) // 2) + 1
11     L = merge_sort(A[:middle])
12     R = merge_sort(A[middle:])
13     return merge(L, R)
```

Programm 6.2: Implementation der Funktion `merge_sort`

Den *merge sort* Algorithmus zu verstehen ist nicht einfach! Erfahrungsgemäss ist es sehr hilfreich, die einzelnen Schritte dieses Sortieralgorithmus für ein kleines Beispiel durchzugehen. Dazu haben wir die Schritte von *merge sort* (`ms`) beim Sortieren der Liste `[3,2,1]` in Abbildung 6.3 schematisch dargestellt. Mit `m` wird natürlich der Algorithmus *merge* bezeichnet. Die schematische Darstellung ist analog zu den Ausführungen in Abschnitt 4.5 zu verstehen.

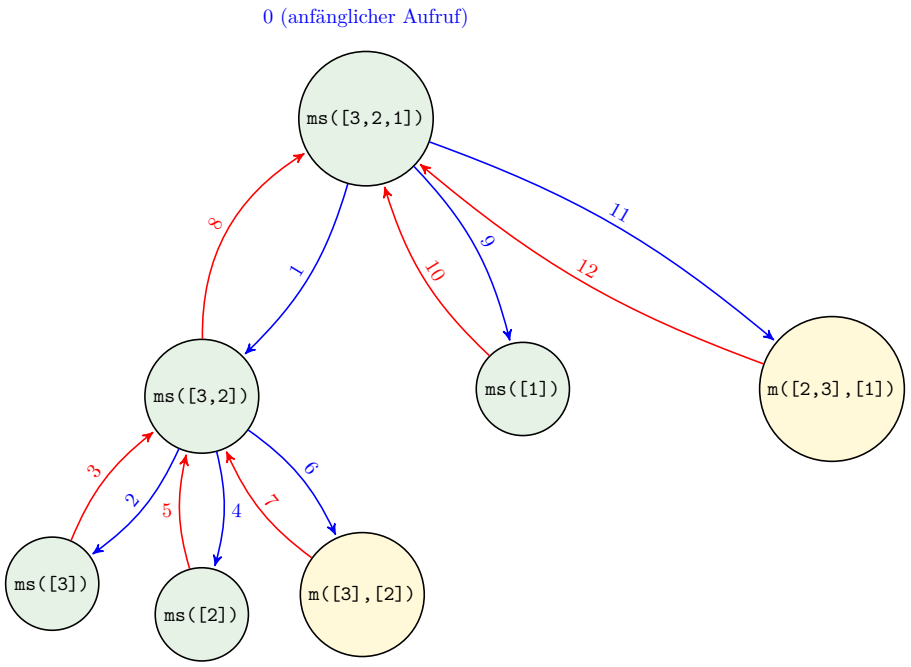


Abbildung 6.3: schematische Darstellung der Arbeitsschritte von *merge sort*

6.2 Analyse der Laufzeit von *merge sort*

Beachten Sie, dass der in Listing 6.2 gegebene Algorithmus durchaus in der Lage ist, Listen beliebiger Längen $n \in \mathbb{N}$ zu sortieren. Für die nachfolgenden Untersuchungen wollen wir aber annehmen, dass die Problemgrösse n stets eine Zweierpotenz ist, also $n = 2^k$ für ein $k \in \mathbb{N}$. Ohne diese Vereinfachungen werden in den Untersuchungen diverse Auf- und Abrundeoperationen notwendig sein und die Darstellung wird technischer und weniger instruktiv. Zusätzlich möchten wir den trivialen Fall $n = 0$ als Problemgrösse ausschliessen. Die folgenden Betrachtungen sind inspiriert durch die entsprechenden Abschnitte in dem herausragenden Buch [9].

Die Laufzeit des *merge sort* Algorithmus setzt sich aus drei einzelnen Teilen zusammen:

Divide: Dieser Teil berechnet lediglich die Mitte der zu sortierenden Liste. Dazu ist offensichtlich nur eine konstante Zeit $\Theta(1)$ notwendig.

Conquer: Beim Aufruf des *merge sort* Algorithmus mit Problemgrösse n werden rekursiv zwei Teilprobleme (derselben Art) mit jeweils halber Grösse $n/2$ aufgerufen. Dies trägt $2T(n/2)$ zur Laufzeit bei.

Combine: Wie wir in Aufgabe 6.2 bereits festgestellt haben, benötigt *merge sort* die lineare Zeit $\Theta(n)$ für die Vereinigung zweier Listen mit summierter Länge `n = len(L) + len(R)`.

Zusammengefasst ist die Laufzeit $T(n)$ von *merge sort* im schlimmsten Fall also gegeben durch

$$T(n) = \begin{cases} \Theta(1), & \text{falls } n = 1, \\ 2T(n/2) + \Theta(n), & \text{falls } n \geq 2. \end{cases} \tag{6.1}$$

Gleichung (6.1) lässt sich natürlich schreiben als

$$T(n) = \begin{cases} c_0, & \text{falls } n = 1, \\ 2T(n/2) + c_1 n, & \text{falls } n \geq 2. \end{cases} \tag{6.2}$$

Diesen Ausdruck können wir mit der Definition $c := \max\{c_0, c_1\}$ sogar nochmals vereinfachen zu

$$T(n) = \begin{cases} c, & \text{falls } n = 1, \\ 2T(n/2) + cn, & \text{falls } n \geq 2, \end{cases} \tag{6.3}$$

da wir uns im Moment lediglich für eine obere Schranke für die Laufzeit $T(n)$ interessieren. Beachten Sie, dass Gleichung (6.3) die Funktion T (Laufzeit) rekursiv definiert.

Wir wollen nun die eindeutige Lösung von Gleichung (6.3) durch intuitive Überlegungen finden. Betrachten Sie Abbildung 6.4.

- In der obersten „Etage“ fallen die Kosten cn für den *merge* zu einer Liste der Länge n an.
- In der zweitobersten „Etage“ fallen die Kosten $cn/2 + cn/2 = cn$ für zwei *merges* zu Listen der Längen $n/2$ an.
- Dies geht rekursiv so weiter.
- In der untersten „Etage“ wird überall der Rekursionsanfang erreicht und wir haben n-mal die Kosten $T(1) = c$, also cn.

Diese Überlegung zeigt, dass jede „Etage“ genau cn zu den Gesamtkosten von *merge sort* beiträgt. Nun stellt sich lediglich noch die Frage, wie viele „Etagen“ der Baum in Abbildung 6.4 hat. Diese Frage ist aber mit der Frage verwandt, wie häufig eine Zweierpotenz 2^k halbiert werden muss, bis das

Resultat der Division durch 2 identisch zu 1 ist. Dies ist aber genau das, was uns der Logarithmus zur Basis zwei beantwortet. Der Faktor 2 ist

$$\log_2(n) = \log_2\left(2^k\right) = k$$

Male in n enthalten. Das ist die Anzahl Teilungen. Der Baum hat somit $\log_2(n) + 1$ viele „Etagen“. Die Laufzeit von *merge sort* ist also

$$cn\left(\log_2(n) + 1\right)$$

und somit $\Theta\left(n \log(n)\right)$.

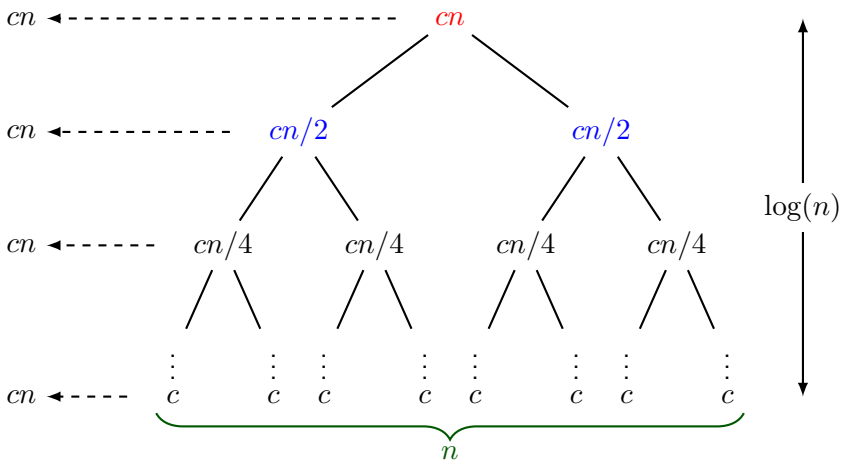


Abbildung 6.4: Laufzeitanalyse von *merge sort*

Aufgabe 6.3

Zeigen Sie mithilfe der vollständigen Induktion, dass

$$T(n) = n \log_2(n)$$

die rekursive Gleichung (Rekurrenz)

$$T(n) = \begin{cases} 2, & \text{falls } n = 2, \\ 2T(n/2) + n, & \text{falls } n = 2^k \text{ für ein } k \in \mathbb{N}, k \geq 2 \text{ erfüllt.} \end{cases}$$

6.3 Lösungen der Aufgaben

Lösungsvorschlag zu Aufgabe 6.1

```python
def merge(L,R):
    # L und R sind jeweils bereits sortierte Listen.

    # M wird am Ende len(L) + len(R) viele Elemente haben
    # und eine sortierte Liste sein.
    M = [] # M ist zu Beginn eine leere Liste.

    l = 0 # Laufindex innerhalb der Liste L
    r = 0 # Laufindex innerhalb der Liste R

    # solange weder L noch R vollständig durchlaufen wurden
    while (l < len(L)) and (r < len(R)):
        if L[l] <= R[r]:
            M.append(L[l])
            l = l + 1
        else:
            M.append(R[r])
            r = r + 1

    while l < len(L):
        M.append(L[l])
        l = l + 1
    while r < len(R):
        M.append(R[r])
        r = r + 1
    return M
```

Programm 6.3: Implementation der Funktion `merge`

Lösungsvorschlag zu Aufgabe 6.2

Wir beziehen uns hier auf den Algorithmus in der Musterlösung zu Aufgabe 6.1. Der Algorithmus besteht im Wesentlichen in einer Schleife über die summierte Länge

```
n = len(L) + len(R)
```

der beiden gegebenen Listen. Somit ist die Laufzeit gegeben durch $\Theta(n)$.

Lösungsvorschlag zu Aufgabe 6.3

Sei also $T(n) := n \log_2(n)$. Es ist zu zeigen, dass diese Wahl von T die Rekurrenz in der Aufgabenstellung erfüllt. Wir untersuchen den Fall $T(2) = T\left(2^1\right)$ separat. Wegen $2\log_2(2) = 2$ ist die Behauptung für $T\left(2^1\right)$ korrekt. Nun zeigen wir durch Induktion, dass für alle $k \in \mathbb{N}$ mit $k \geq 2$ die Aussage

$$\mathcal{A}(k) : \Longleftrightarrow\ T\left(2^k\right) = 2T\left(2^{k-1}\right) + 2^k$$

korrekt ist. Mit den Logarithmengesetzen finden wir

$$T\left(2^2\right) = 2^2 \log_2\left(2^2\right) = 2^2(\log_2(2) + \log_2(2)) = 2 \cdot 2\log_2(2) + 2^2 = 2T\left(2^1\right) + 2^2$$

und damit ist $\mathcal{A}(2)$ gezeigt. Sei $\mathcal{A}(n)$ nun für ein $n \in \mathbb{N}$ mit $n \geq 2$ wahr, dann finden wir

$$
\begin{aligned}
&T\left(2^{k+1}\right) = \\
&2^{k+1} \log_2\left(2^{k+1}\right) = \\
&2^{k+1}\left(\log_2\left(2^k\right) + \log_2(2)\right) = \\
&2 \cdot 2^k \log_2\left(2^k\right) + 2^{k+1} = \\
&2T\left(2^k\right) + 2^{k+1}.
\end{aligned}
$$

Dies zeigt den Induktionsschritt.

Kapitel 7

Sudoku und Backtracking (*)

Sudokus sind Rätsel, welche sich seit den frühen 2000er Jahren international grosser Beliebtheit erfreuen. Seit vielen Jahren gibt es Sudoku-Apps und man findet Sudokus auch oft abgedruckt in Zeitschriften und Gratiszeitungen. Eines der zentralen Ziele dieses Kapitels wird sein, einen rekursiven Lösungsalgorithmus für Sudokus zu entwickeln. Zuerst müssen wir kurz erklären, was Sudokus überhaupt sind und wie die Spielregeln aussehen.

7.1 Spielregeln

Die Spielregeln für Sudokus lassen sich ganz einfach erklären. Wir illustrieren sie anhand eines konkreten (sehr schwierigen) Sudokus. Ein Sudoku besteht immer aus einem 9×9-Gitter. In Abbildung 7.1 ist links das noch ungelöste Sudoku gezeigt und rechts das gelöste (ausgefüllte). Bei einem Sudoku sind zu Beginn einige Zahlen (Felder) vorgegeben. Diese, zu Beginn vorgegebenen Zahlen, dürfen nicht geändert werden. Das Sudoku ist gelöst, wenn jedes der 81 Felder genau eine der 9 Ziffern $1, 2, \ldots, 9$ enthält und zusätzlich die folgenden zwei Bedingungen erfüllt sind:

1. In jeder Zeile und jeder Spalte muss jede der 9 Ziffern $1, 2, \ldots, 9$ genau einmal vorkommen.
2. Beachten Sie, dass das 9×9-Gitter (siehe die fetten Linien) wiederum in 9 verschiedene 3×3-Blöcke aufgeteilt ist. In jedem dieser 3×3-Blöcke müssen ebenfalls alle 9 Ziffern $1, 2, \ldots, 9$ genau einmal vorkommen.

		7			3	9		2
			8					
9	4	3						7
6	9							
3			5	2	7			
						8	4	
				4	8			
2	6							
						1	2	9

ungelöstes Sudoku Nr. 0

8	*1*	7	*4*	*5*	3	9	*6*	2
5	*2*	*6*	8	*7*	*9*	*3*	*1*	*4*
9	4	3	*1*	*6*	*2*	*5*	*8*	7
6	9	*1*	*3*	*8*	*4*	*2*	*7*	*5*
3	*8*	*4*	5	2	7	*6*	*9*	*1*
7	*5*	*2*	*9*	*1*	*6*	8	4	*3*
1	*3*	*9*	*2*	4	8	*7*	*5*	*6*
2	6	*5*	*7*	*9*	*1*	*4*	*3*	*8*
4	*7*	*8*	*6*	*3*	*5*	1	2	9

gelöstes Sudoku Nr. 0

Abbildung 7.1: Sudoku Nr. 0

Normale 9×9-Sudokus sind mit ihren 81 Feldern recht gross und für didaktische Betrachtungen eher unhandlich. Wir werden deshalb zuerst kleinere Sudokus der Grösse 4×4 untersuchen. Lassen

Sie uns zunächst die Regeln für Sudokus für allgemeine Dimensionen festhalten.

Bemerkung 7.1 (Spielregeln des (verallgemeinerten) Sudokus)**:**
myremark:Regeln Ein (verallgemeinertes) Sudoku ist ein $n^2 \times n^2$-Gitter, wobei n eine natürliche Zahl ist. Bei einem Sudoku sind zu Beginn einige Zahlen (Felder) vorgegeben. Diese zu Beginn vorgegebenen Zahlen dürfen nicht geändert werden. Das Sudoku ist gelöst, wenn jedes der $n^2 \cdot n^2 = n^4$ Felder genau eine der Ziffern $1, 2, \ldots, n^4$ enthält und zusätzlich die folgenden zwei Bedingungen erfüllt sind:

1. In jeder Zeile und in jeder Spalte muss jede der n^2 Ziffern $1, 2, \ldots, n^2$ genau einmal vorkommen.
2. Beachten Sie, dass das $n^2 \times n^2$-Gitter (siehe die fetten Linien) wiederum in n^2 verschiedene $n \times n$-Blöcke aufgeteilt ist. In jedem dieser $n \times n$-Blöcke müssen ebenfalls alle n^2 Ziffern $1, 2, \ldots, n^2$ genau einmal vorkommen.

Das übliche Sudoku (mit 81 Feldern) erhalten wir für $n = 3$. Wir werden zunächst kleine Sudokus mit $n = 2$, also mit nur $2^4 = 16$ Feldern anschauen.

Aufgabe 7.1

Finden Sie die (eindeutige) Lösung des Sudokus in Abbildung 7.2. Dieses Sudoku wird als *einfach* eingestuft.

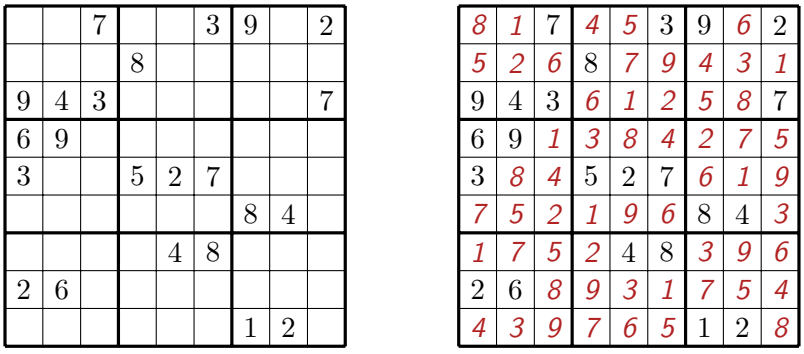
	9	5					7	
	7		1	2		8		
	2	4	6	7		3	9	1
					1	2	5	3
	1		3		4			9
7	3	9	8					4
			9	8	6			
				4			1	6
2	4							

ungelöstes Sudoku Nr. 2

Abbildung 7.2: Sudoku Nr. 2

7.2 Sudokus können mehr als eine Lösung haben!

Betrachten Sie nochmals Abbildung 7.1 und vergewissern Sie sich, dass die Lösung auf der rechten Seite tatsächlich alle geforderten Bedingungen erfüllt. Sudoku Nr. 0 erlaubt übrigens nur diese eine Lösung. Die in Zeitschriften und Sudoku-Apps aufgeführten Sudokus sind meist absichtlich so konstruiert, dass sie eine eindeutige Lösung besitzen. Entfernen wir aber beispielsweise die 9 in der rechten unteren Ecke von Sudoku Nr. 0, so erhalten wir ein neues Rätsel:



ungelöstes Sudoku Nr. 1 gelöstes Sudoku Nr. 1

Abbildung 7.3: Sudoku Nr. 1

Sudoku Nr. 1 besitzt ebenfalls die Lösung, welche bereits Lösung von Sudoku Nr. 0 war (siehe Abbildung 7.1). Neben dieser Lösung besitzt Sudoku Nr. 1 aber noch weitere Lösungen. Eine davon ist rechts in Abbildung 7.3 gezeigt.

In Abbildung 7.4 ist ein Beispiel eines 4×4-Sudokus („Mini-Sudoku“) gegeben. Dieses besitzt übrigens 4 verschiedene Lösungen.

ungelöstes Mini-Sudoku Nr. 0 eine Lösung des Mini-Sudokus Nr. 0

Abbildung 7.4: Mini-Sudoku Nr. 0

Aufgabe 7.2

Finden Sie die drei weiteren Lösungen des Sudokus in Abbildung 7.4.

7.3 Darstellung von Sudokus in Python

In Python können wir ein Sudoku durch eine „zweidimensionale“ Liste (oder Array) angeben. Beispielsweise kann Sudoku Nr. 2 aus Aufgabe 7.1 in der folgenden Form in Python gespeichert werden:

```
sudokuNo2 = [
[0,9,5,0,0,0,0,7,0],
[0,7,0,1,2,0,8,0,0],
[0,2,4,6,7,0,3,9,1],
[0,0,0,0,0,1,2,5,3],
[0,1,0,3,0,4,0,0,9],
[7,3,9,8,0,0,0,0,4],
[0,0,0,9,8,6,0,0,0],
[0,0,0,0,4,0,0,1,6],
[2,4,0,0,0,0,0,0,0]
]
```

Programm 7.1: Abspeichern von Sudoku-Gittern in Python

Das Mini-Sudoku in Abbildung 7.4 kann folgendermassen in Python abgespeichert werden:

```
mini_sudokuNo0 = [
[0,1,0,0],
[0,0,0,0],
[0,3,0,2],
[0,0,3,0]
]
```

Programm 7.2: Abspeichern von Mini-Sudoku-Gittern in Python

Um ein $n^2 \times n^2$-Gitter ansehnlich in Python auszugeben („pretty-print“), empfehlen wir, am Anfang des Programms die Bibliothek `numpy` durch den Befehl `import numpy as np` einzubinden. Danach kann das Gitter durch `print(np.array(gitter))` übersichtlich ausgegeben werden.

7.4 Erlaubte Felder

In Abbildung 7.5 ist nochmals das Sudoku vom Anfang des Kapitels gezeigt. Betrachten Sie das mit einem blauen Fragezeichen ? markierte Feld. Wir wissen (aufgrund der Eindeutigkeit der Lösung dieses konkreten Sudokus), dass in das markierte Feld die Ziffer 8 gesetzt werden muss. Nach den Spielregeln von Sudoku würde jedoch im Moment nichts dagegen sprechen, eine 1 in dieses Feld zu schreiben — auch wenn sich diese Wahl im Verlauf des Spiels als falsch herausstellen wird. Wir sagen, dass die Wahl der Ziffer 1 für das blau markierte Feld (bei dem aktuellen Spielstand) *erlaubt* ist. Die Wahl der Ziffer 6 ist beispielsweise nicht erlaubt, da diese Ziffer in der entsprechenden Zeile bereits vorkommt. Ebenso wäre die Wahl der Ziffer 5 nicht erlaubt, da diese Ziffer bereits im entsprechenden 3×3-Block enthalten ist.

Es sei also ein konkretes Gitter gegeben. Wir sagen, dass das Setzen einer Ziffer Z auf ein leeres (unbesetztes) Feld F des Gitters ***erlaubt*** ist, falls diese Ziffer Z weder in der entsprechenden Zeile noch der Spalte noch dem entsprechenden $n \times n$-Block von F in dem gegebenen Gitter vorkommt.

		7			3	9		2
			8					
9	4	3						7
6	9			?				
3			5	2	7			
						8	4	
				4	8			
2	6							
						1	2	9

ungelöstes Sudoku Nr. 0

8	*1*	7	*4*	*5*	3	9	*6*	2
5	*2*	*6*	8	*7*	*9*	*3*	*1*	*4*
9	4	3	*1*	*6*	*2*	*5*	*8*	7
6	9	*1*	*3*	*8*	*4*	*2*	*7*	*5*
3	*8*	*4*	5	2	7	*6*	*9*	*1*
7	*5*	*2*	*9*	*1*	*6*	8	4	*3*
1	*3*	*9*	*2*	4	8	*7*	*5*	*6*
2	6	*5*	*7*	*9*	*1*	*4*	*3*	*8*
4	*7*	*8*	*6*	*3*	*5*	1	2	9

gelöstes Sudoku Nr. 0

Abbildung 7.5: nochmals Sudoku Nr. 2

Aufgabe 7.3

Wir befassen uns in dieser Aufgabe nur mit 4×4-Gittern. Entwickeln Sie eine Python-Funktion

`def erlaubt(zeile, spalte, ziffer, gitter)`,

welche für ein gegebenes noch **leeres** (markiert durch die Zahl 0) Feld mit Zeilenindex `zeile` (von 0 bis 3) und Spaltenindex `spalte` (von 0 bis 3) prüft, ob das Platzieren einer gegebenen Ziffer `ziffer` (von 1 bis 4) in einem gegebenen Gitter erlaubt ist. Hier sind einige Testfälle:

```
print(erlaubt(0,3,4,mini_sudokuNo0)) # True

print(erlaubt(1,0,1,mini_sudokuNo0)) # False
# (bereits eine 1 in dem ersten 4x4 Block)

print(erlaubt(1,1,3,mini_sudokuNo0)) # False
# (bereits eine 3 in der entsprechenden Spalte)
```

Programm 7.3: erlaubt oder nicht

Testen Sie Ihre Funktion genau!

Aufgabe 7.4

Verallgemeinern Sie die Funktion

`def erlaubt(zeile, spalte, ziffer, gitter)`

auf $n^2 \times n^2$-Gitter.

7.5 Backtracking

Wir wollen uns nun an die Entwicklung eines rekursiven Algorithmus zum Lösen von Sudokus herantasten. Dazu beginnen wir mit der einführenden Aufgabe 7.5, in der Sie einige gedankliche Vorarbeiten leisten.

Aufgabe 7.5

Betrachten Sie das folgende Sudoku.

	3		
	1		4
3		1	

(a) Füllen Sie die noch leeren Felder Zeile für Zeile von links nach rechts und von oben nach unten aus. Beginnen Sie also mit dem Feld mit den Koordinaten $(0, 0)$ (oben links) und beenden Sie Ihre Arbeit mit dem Feld $(3, 3)$ (unten rechts). Zeichnen Sie dabei die einzelnen „Stationen“ (Gitter) auf, indem Sie das Diagramm in Abbildung 7.6 systematisch vervollständigen.

(b) Warum ist unser Vorgehen in Teil (a) für das gegebene Sudoku suboptimal? Schlagen Sie eine effizientere Strategie vor.

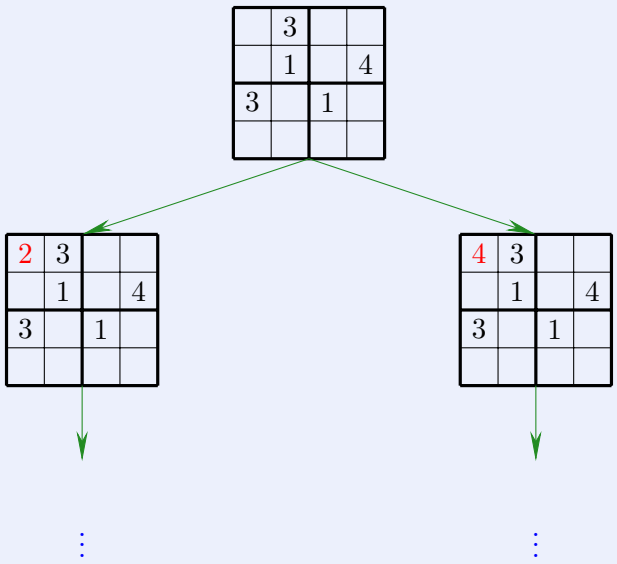
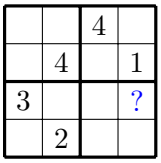


Abbildung 7.6: einzelne Stationen

Nun wollen wir die in Aufgabe 7.5 gemachten Beobachtungen etwas vertiefen. Betrachten Sie dazu das neue Mini-Sudoku in Abbildung 7.7.



ungelöstes Mini-Sudoku Nr. 1

1	*3*	4	*2*
2	4	*3*	1
3	*1*	*2*	*4*
4	2	*1*	*3*

Lösung des Mini-Sudoku Nr. 1

Abbildung 7.7: Mini-Sudoku Nr. 1

Beim Lösen eines Sudokus dürfen wir die Reihenfolge, in der wir die (noch) leeren Felder auffüllen, beliebig wählen. Um ganz konkret ein mögliches Vorgehen zu untersuchen, nehmen wir an, dass wir die 11 leeren Felder des Sudokus in Abbildung 7.7 in der Reihenfolge

$$(2,3) \to (3,3) \to (0,3) \to (1,2) \to (1,0) \to (2,2) \to (3,2) \to (2,1) \to (3,0) \to (0,0) \to (0,1)$$

auszufüllen versuchen.

Wir betrachten also zuerst das leere Feld mit den Koordinaten $(2,3)$, welches mit einem blauen Fragezeichen ? markiert ist. Sie werden schnell erkennen, dass in diesem leeren Feld genau das Setzen der beiden Ziffern 2 und 4 erlaubt ist. Wir stehen hier also vor einer Wahl. Die Situation ist in Abbildung 7.8 dargestellt. Die Wahl der Ziffer 2 führt uns zu Station (1). Hier haben wir für das Feld $(3,3)$ ebenfalls eine Wahl, und zwar zwischen den Ziffern 3 und 4. Die Wahl der Ziffer 3 führt uns zu Station (2). Bei Station (3) stellen wir fest, dass für das leere Feld $(0,3)$ keine der vier Ziffern gesetzt werden darf. Somit sind wir in eine Sackgasse geraten. Mindestens an einer „Abzweigung“ müssen wir also eine falsche Wahl getroffen haben! Wir gehen deshalb so weit den „Pfad“ entlang zurück, bis wir zur jüngst angetroffenen Abzweigung gelangen. Dies war in diesem Fall die Station (1). Nun folgen wir von Station (1) aus der Wahl der Ziffer 4 zu Station (4). Bei Station (7) erkennen wir, dass auch die zweite Wahl (die der Ziffer 4) bei Station (1) uns nicht weiterbringt. Erst jetzt können wir sicher sein, dass die ursprüngliche Wahl der Ziffer 2 bei Station (0) falsch gewesen war. Dies führt uns zur korrekten Wahl der Ziffer 4 für das Feld $(2,3)$ und somit zu Station (8).

Der Vorgang des „Zurückgehens“ entlang der gegangenen Wege wird ***Backtracking*** genannt. Im nächsten Abschnitt werden wir einen eleganten Algorithmus vorstellen, welcher mithilfe von Backtracking auf rekursive Weise die Lösungen eines Sudokus findet.

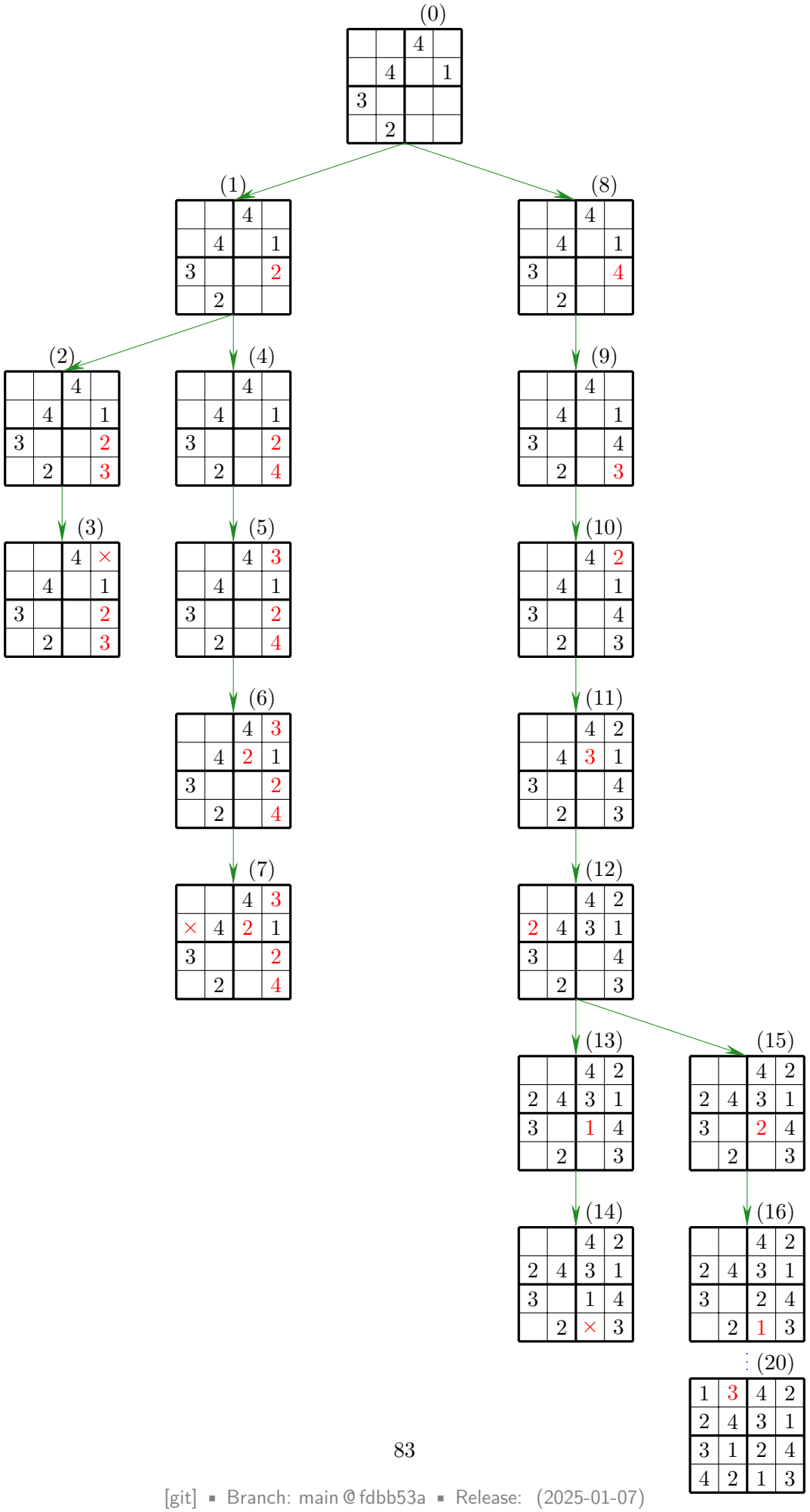


Abbildung 7.8: Visualisierung von Backtracking bei Sudoku

Bemerkung 7.2 (Backtracking in einem Labyrinth)**:**
Die Situation beim Backtracking in Sudokus kann vage mit der (rekursiven) Suche eines Ausgangs (oder aller Ausgänge) aus einem Labyrinth verglichen werden. Wir suchen, beginnend bei *Start*, einen der **A**usgänge aus einem Labyrinth. Dazu gehen wir so lange die Gänge entlang, bis wir entweder herausgefunden haben oder in einer **S**ackgasse angelangt sind. In Abbildung 7.9 ist die Situation veranschaulicht. Der Weg *Start* $\to 0 \to 1$ ist eine Sackgasse. Wir gehen darum einen Gang zurück (also zu 0). Von 0 aus gibt es keine weitere Abzweigung und wir gehen nochmals einen Gang zurück, also zum *Start*. Nun gehen wir den neuen Weg *Start* $\to 2 \to 3 \to 4$. Doch auch 4 ist eine Sackgasse. Deshalb gehen wir einen Gang zurück zu 2. Von 2 aus nehmen wir den neuen Weg $2 \to 5 \to 6$ und haben einen Ausgang gefunden. Diesen gefundenen Weg *Start* $\to 2 \to 5 \to 6$ können wir als Lösung ausgeben. Falls wir (wie beim Sudoku) alle Lösungen (Ausgänge) finden wollen, würden wir hier nicht bereits abbrechen, sondern (rekursiv) weitersuchen.

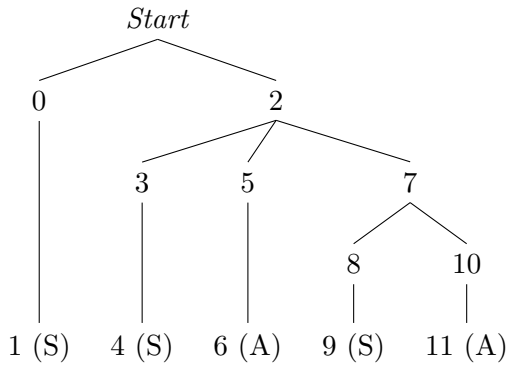


Abbildung 7.9: Navigation in einem Labyrinth

7.6 Lösungsalgorithmus für Sudoku

Unser Lösungsalgorithmus verwendet die Funktion `erlaubt` sowie Backtracking in Kombination mit Rekursion. Diese drei Komponenten sind Ihnen inzwischen bekannt. Der vollständige Algorithmus zum Lösen von Sudokus ist in Listing 7.5 gegeben. Mit der Definition

```
sudokuNo0 = [
  [0,0,7,0,0,3,9,0,2],
  [0,0,0,8,0,0,0,0,0],
  [9,4,3,0,0,0,0,0,7],
  [6,9,0,0,0,0,0,0,0],
  [3,0,0,5,2,7,0,0,0],
  [0,0,0,0,0,0,8,4,0],
  [0,0,0,0,4,8,0,0,0],
  [2,6,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,1,2,9]
 ]
```

Programm 7.4: Abspeichern von Sudoku-Gittern in Python

wird Ihnen der Aufruf `sudoku(sudokuNo0)` alle (es gibt hier nur eine) Lösungen von Sudoku Nr. 0 ausgeben. Die Funktion in Listing 7.5 verwendet die Funktion `erlaubt`, welche Sie in Aufgabe 7.4 geschrieben haben.

```
1  def sudoku(gitter):
2    # Gehe durch alle Felder im Gitter.
3    for zeile in range(9):
4      for spalte in range(9):
5        # Schaue, ob das Feld noch leer ist.
6        if gitter[zeile][spalte] == 0:
7          # leeres Feld gefunden
8
9          # Gehe durch alle 10 Ziffern 1 bis 9.
10         for ziffer in range(1,10):
11           # Prüfe, ob die Ziffer für dieses Feld erlaubt ist.
12           if erlaubt(zeile,spalte,ziffer,gitter):
13             # Die betrachtete Ziffer ist erlaubt.
14
15             # Schreibe diese Ziffer in das Feld.
16             gitter[zeile][spalte] = ziffer
17
18             # Die Ziffer wurde ins Gitter geschrieben.
19             # Arbeite nun rekrusiv mit dem
20             # neuen Gitter weiter.
21             sudoku(gitter)
22
23             # Entferne die gesetzte Ziffer wieder.
24             gitter[zeile][spalte] = 0
25
26         # leeres Feld, für welches keine Ziffer passt
27         # => Sackgasse gefunden => Backtracking
28         return
29   # gültige Lösung gefunden
30   print(np.array(gitter))
```

```
31    return
```

Programm 7.5: Implementation der Funktion `sudoku`

Aufgabe 7.6

Betrachten Sie die `sudoku`-Funktion in Listing 7.5. Es gibt genau zwei verschiedene Möglichkeiten, die Zeile 24 in dieser Funktion zu erreichen. Nennen Sie diese beiden Möglichkeiten und erklären Sie jeweils die Bedeutung des Entfernens der gesetzten Ziffer in dem entsprechenden Fall.

Aufgabe 7.7

(!) Ändern Sie die `sudoku`-Funktion in Listing 7.5 dahingehend ab, dass genau eine Lösung ausgegeben wird, falls das Sudoku mindestens eine Lösung besitzt. Falls das Sudoku keine Lösung besitzt, so soll auch nichts ausgegeben werden.

Aufgabe 7.8

(!) Dies ist eine besonders schwierige Aufgabe. Wir betrachten das bekannte *Damenproblem*. Das Problem besteht darin, 8 Damen auf einem 8×8-Schachbrett so zu platzieren, dass sich keine zwei Damen gegenseitig bedrohen. Finden Sie Inspiration an unserem Sudoku-Löser und schreiben Sie ein rekursives Programm, welches alle Lösungen des Damenproblems ausgibt. Es gibt genau 92 unterschiedliche Lösungen für den 8×8-Fall. Betrachten Sie auch https://en.wikipedia.org/wiki/Eight_queens_puzzle#Counting_solutions_for_other_sizes_n. Gelingt es Ihnen, das allgemeine $n \times n$-Problem für $n \in \mathbb{N}$ zu lösen?

Lösungsvorschlag zu Aufgabe 7.1

	9	5					7	
	7		1	2		8		
	2	4	6	7		3	9	1
					1	2	5	3
	1		3		4			9
7	3	9	8					4
			9	8	6			
				4			1	6
2	4							

ungelöstes Sudoku Nr. 2

1	9	5	*4*	*3*	*8*	*6*	7	*2*
6	7	*3*	1	2	*9*	8	*4*	*5*
8	2	4	6	7	*5*	3	9	1
4	*6*	*8*	*7*	*9*	1	2	5	3
5	1	*2*	3	*6*	4	*7*	*8*	9
7	3	9	8	*5*	*2*	*1*	*6*	4
3	*5*	*1*	9	8	6	*4*	*2*	*7*
9	*8*	*7*	*2*	4	*3*	*5*	1	6
2	4	*6*	*5*	*1*	*7*	*9*	*3*	*8*

gelöstes Sudoku Nr.2

Abbildung 7.10: Sudoku Nr. 2 (mit Lösungen)

Lösungsvorschlag zu Aufgabe 7.2

2	1	4	3
3	4	2	1
4	3	1	2
1	2	3	4

3	1	2	4
2	4	1	3
1	3	4	2
4	2	3	1

3	1	2	4
4	2	1	3
1	3	4	2
2	4	3	1

4	1	2	3
3	2	1	4
1	3	4	2
2	4	3	1

Abbildung 7.11: alle Lösungen des Sudokus in Abbildung 7.4

Lösungsvorschlag zu Aufgabe 7.3

```python
def erlaubt(zeile, spalte, ziffer, gitter):
  # Prüfe, ob 'ziffer' bereits in der gegeben Zeile vorkommt.
  # (Fixiere die Zeile und gehe durch alle Spalten.)
  for j in range(4):
    if gitter[zeile][j] == ziffer:
      return False

  # Prüfe, ob 'ziffer' bereits in der gegeben Spalte vorkommt.
  # (Fixiere die Spalte und gehe durch alle Zeilen.)
  for i in range(4):
    if gitter[i][spalte] == ziffer:
      return False

  # Berechne Indizes der "linken oberen Ecke" des relevanten 2x2-Blocks.
  zeile_block = zeile - (zeile % 2)
  spalte_block = spalte - (spalte % 2)
  # Prüfe, ob 'ziffer' bereits in dem relevanten 2x2-Block vorkommt.
  for i_block in range(2):
    for j_block in range(2):
      if gitter[zeile_block+i_block][spalte_block+j_block] == ziffer:
        return False
  return True
```

Programm 7.6: Implementation der Funktion `erlaubt`

Lösungsvorschlag zu Aufgabe 7.4

```python
def erlaubt(zeile, spalte, ziffer, gitter):
  n = int(np.sqrt(len(gitter[0])))
```

```
3
4   for j in range(n**2):
5     if gitter[zeile][j] == ziffer:
6       return False
7
8   for i in range(n**2):
9     if gitter[i][spalte] == ziffer:
10      return False
11
12  zeile_block = zeile - (zeile % n)
13  spalte_block = spalte - (spalte % n)
14  for i_block in range(n):
15    for j_block in range(n):
16      if gitter[zeile_block+i_block][spalte_block+j_block] == ziffer:
17        return False
18  return True
```

Programm 7.7: Funktion `erlaubt` für allgemeine Sudokus

Lösungsvorschlag zu Aufgabe 7.5

(a) Die Stationen sind in Abbildung 7.12 gegeben.

(b) Beim Zeichnen des Diagramms in Teilaufgabe (a) werden Sie festgestellt haben, dass die Arbeit mit Feldern, die das Setzen von mehr als einer Ziffer erlauben, den Aufwand erheblich erhöht. Es ist deshalb intuitiv klar, dass man zuerst alle Felder besetzen möchte, die nur eine Ziffer erlauben. Danach sollte man die Felder besetzen, welche nur zwei Ziffern erlauben und so weiter. Unser Vorgehen in Teilaufgabe (a) ist suboptimal, da wir beispielsweise als Erstes ein Feld zu füllen versuchen, welches zwei Ziffern erlaubt.

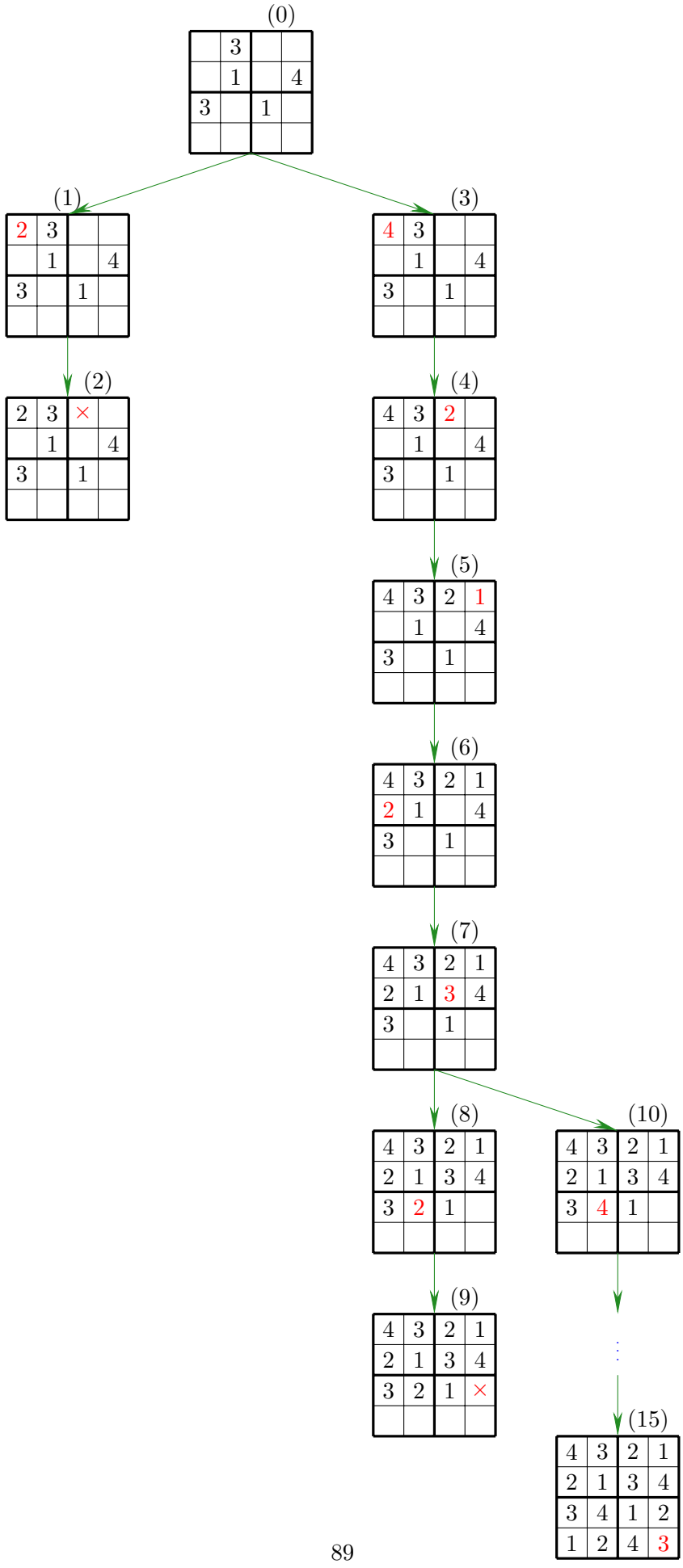


Abbildung 7.12: alle Stationen

Lösungsvorschlag zu Aufgabe 7.6

Zeile 24 wird genau dann erreicht, wenn der rekursive Aufruf auf Zeile 21 abgeschlossen werden konnte. Dies ist genau dann der Fall, wenn der rekursive Aufruf einen `return` lieferte. Dazu gibt es exakt zwei Möglichkeiten:

return auf Zeile 28: Der `return` auf Zeile 28 wird genau dann ausgelöst, wenn der rekursive Aufruf auf Zeile 21 in eine „Sackgasse“ gelangt ist. Dies geschieht genau dann, wenn dieser Aufruf ein noch leeres Feld gefunden hat, in welches keine der 10 Ziffern gesetzt werden darf (Widerspruch). Die gesetzte Ziffer muss also wieder entfernt werden und wir versuchen es mit einer anderen Ziffer.

return auf Zeile 31: Der `return` auf Zeile 31 wird genau dann ausgelöst, wenn der rekursive Aufruf auf Zeile 21 in kein leeres Feld gefunden hat. Das bedeutet aber, dass dieser Aufruf eine gültige Lösung gefunden (und ausgegeben) hat. Durch das Entfernen der gesetzten Ziffer können rekursiv (falls vorhanden) weitere Lösungen gefunden werden (siehe auch Bemerkung 7.2).

Lösungsvorschlag zu Aufgabe 7.7

Diese Funktion bricht ihre Berechnung ab, sobald eine Lösung gefunden wurde.

```
1  def sudoku(gitter):
2    for zeile in range(9):
3      for spalte in range(9):
4        if (gitter[zeile][spalte] == 0):
5          for ziffer in range(1,10):
6            if erlaubt(zeile,spalte,ziffer,gitter):
7              gitter[zeile][spalte] = ziffer
8              solution_found = sudoku(gitter)
9              if solution_found:
10               return True
11             gitter[zeile][spalte] = 0
12         return
13   print(np.array(gitter))
14   return True
```

Programm 7.8: Implementation der Funktion `sudoku`

Lösungsvorschlag zu Aufgabe 7.8

Wir geben eine Lösung in `Python` und eine alternative Lösung in `C++` an. Das Python-Programm verwendet eine globale Variable, was sehr unschön ist. Diese könnte man beispielsweise durch *Python-Klassen* vermeiden. Wir wollen an dieser Stelle aber nicht auf diese eingehen.

Lösung in Python

```
1  import numpy as np
2
3  def check_square(row, column, board, n):
4      # check if column is already occupied by a queen in one of the rows above
5      i = row - 1
6      while (i >= 0):
7          if board[i][column] == 1:
8              return False
9          i -= 1
10
```

```
    # check north-western diagonal
    i = row - 1
    j = column - 1
    while ((i >= 0) and (j >= 0)):
        if board[i][j] == 1:
            return False
        i -= 1
        j -= 1

    # check north-eastern diagonal
    i = row - 1
    j = column + 1
    while ((i >= 0) and (j < n)):
        if board[i][j] == 1:
            return False
        i -= 1
        j += 1

    return True

number_of_solutions = 0
def solve_queen_problem(board, n, row = 0):
    global number_of_solutions # can be done way prettier! for example with classes
    if (row == n):
        number_of_solutions += 1
        print(board)
        return

    for j in range(n):
        if (check_square(row, j, board, n)):
            board[row][j] = 1
            solve_queen_problem(board, n, row + 1)
            board[row][j] = 0

def run():
    n = 8
    board = np.zeros((n,n), dtype = int)
    solve_queen_problem(board, n)
    print("for n = {0:>2}".format(n), ", there are exactly", "{0:>15}".format(number_of_solutions
    ), " solutions", sep = '')

run()
```

Lösung in C++

```
#include "n_queen_problem.h"
#include <iostream>
int main() {
    Queen_Problem Q = Queen_Problem(8);
    Q.solve_queen_problem();
    std::cout << Q.get_number_of_solutions() << std::endl;
    return 0;
}
```

```
// n_queen_problem.h

class Queen_Problem
{
  public:
    Queen_Problem(const unsigned int);
    ~Queen_Problem(void)
    {
```

```
      for (unsigned int i = 0; i < n; ++i)
      {
        delete[] board[i];
      }
      delete[] board;
    }
    void print_solution(void) const;
    bool check_square(const unsigned int, const unsigned int) const;
    void solve_queen_problem(const unsigned int = 0);
    unsigned long long get_number_of_solutions(void) const
    {
      return number_of_solutions;
    }

  private:
    unsigned long long number_of_solutions;
    unsigned int n;
    bool** board;
};
```

```
// n_queen_problem.cpp

#include <iostream>
#include <cstring>
#include "n_queen_problem.h"

Queen_Problem::Queen_Problem(const unsigned int board_size) : number_of_solutions{0}, n{
    board_size}
{
  board = new bool*[n];
  for (unsigned int i = 0; i < n; ++i)
  {
    board[i] = new bool[n];
    memset(board[i], false, n * sizeof(bool));
  }
}

void Queen_Problem::print_solution(void) const
{
  std::cout << "\n";
  for (unsigned int i = 0; i < n; ++i)
  {
    for (unsigned int j = 0; j < n; ++j)
    {
      std::cout << board[i][j] << " ";
    }
    std::cout << "\n";
  }
  std::cout << "\n";
}

bool Queen_Problem::check_square(const unsigned int row, const unsigned int column) const
{
  int i;
  int j;

  // check if column is already occupied by a queen in one of the rows above
  for (i = row - 1; i >= 0; --i)
  {
    if (board[i][column] == 1)
    {
      return false;
```

```
42      }
43    }
44
45    // check north-western diagonal
46    for (i = row - 1, j = column - 1; (i >= 0) and (j >= 0); --i, --j)
47    {
48      if (board[i][j] == 1)
49      {
50        return false;
51      }
52    }
53
54    // check north-eastern diagonal
55    for (i = row - 1, j = column + 1; (i >= 0) and (j < static_cast<int>(n)); --i, ++j)
56    {
57      if (board[i][j] == 1)
58      {
59        return false;
60      }
61    }
62    return true;
63  }
64
65  void Queen_Problem::solve_queen_problem(const unsigned int row)
66  {
67    if (row == n)
68    {
69      print_solution();
70      ++number_of_solutions;
71      return;
72    }
73
74    for (unsigned int j = 0; j < n; ++j)
75    {
76      if ( check_square(row, j) )
77      {
78        board[row][j] = 1;
79        solve_queen_problem(row + 1);
80        board[row][j] = 0;
81      }
82    }
83  }
```

Literatur

[1] Armin P. Barth. *Mathematik fürs Gymnasium (Band 1, 1. Auflage)*. Mathematik fürs Gymnasium (Band 1 von 4). hep, 2021. ISBN: 978-3-0355-1786-6.

[2] Terence Tao. *Analysis I (Third Edition)*. Texts and Readings in Mathematics. Springer Singapore, 2016. ISBN: 978-981-10-1789-6.

[3] Herbert Amann und Joachim Escher. *Analysis I (3. Auflage)*. Amann-Escher Analysis (Band 1 von 3). Birkhäuser Verlag, 2006. ISBN: 3 7643 7755 0.

[4] Edmund Landau. *Grundlagen der Analysis (Das Rechnen mit ganzen, rationalen, irrationalen, komplexen Zahlen)*. Ergänzung zu den Lehrbüchern der Differential- und Integralrechnung. Akademische Verlagsgesellschaft M.B.H. Leipzig, 1992.

[5] Piotr Łukowski. *Paradoxes*. TRENDS IN LOGIC Studia Logica Library. Dordrecht Heidelberg London New York, 2011. ISBN: 978-94-007-1475-5.

[6] Felix Friedrich. *Vorlesung: Datenstrukturen und Algorithmen an der ETH Zürich*. 2018. URL: https://lec.inf.ethz.ch/DA/2018/slides/daLecture1.handout.2x2.pdf (besucht am 04. 12. 2023).

[7] Albert Paul Malvino und Jerald A. Brown. *Digital Computer Electronics (Third Edition)*. McGraw-Hill Inc.,US, 1930.

[8] Markus Sauter. *Design eines programmierbaren Rechners*. Bildungsportal der ETH Zürich, 2006.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.