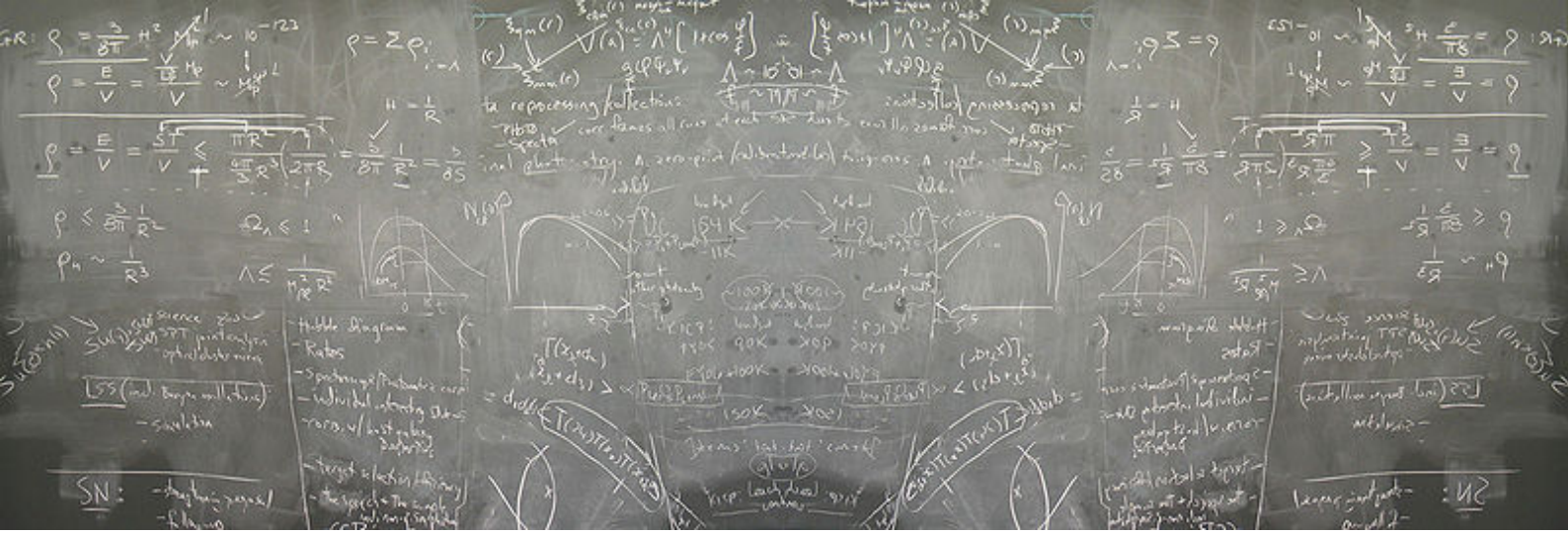
Aufgaben zur Mathematik für ADS

Formales Denken für angehende Algorithmiker

Sebastian Berndt

Stand: 30. Januar 2019, 17:43:44 Uhr



Inhaltsverzeichnis

1 Mengen und Beweise 7

2 Aussagenlogik, Prädikatenlogik 9

3 Folgen, Induktion, Rekurrenzen 11

4 Funktionen, O-Notation, Modulare Arithmetik 15

4.1 Funktionen und O-Notation 15

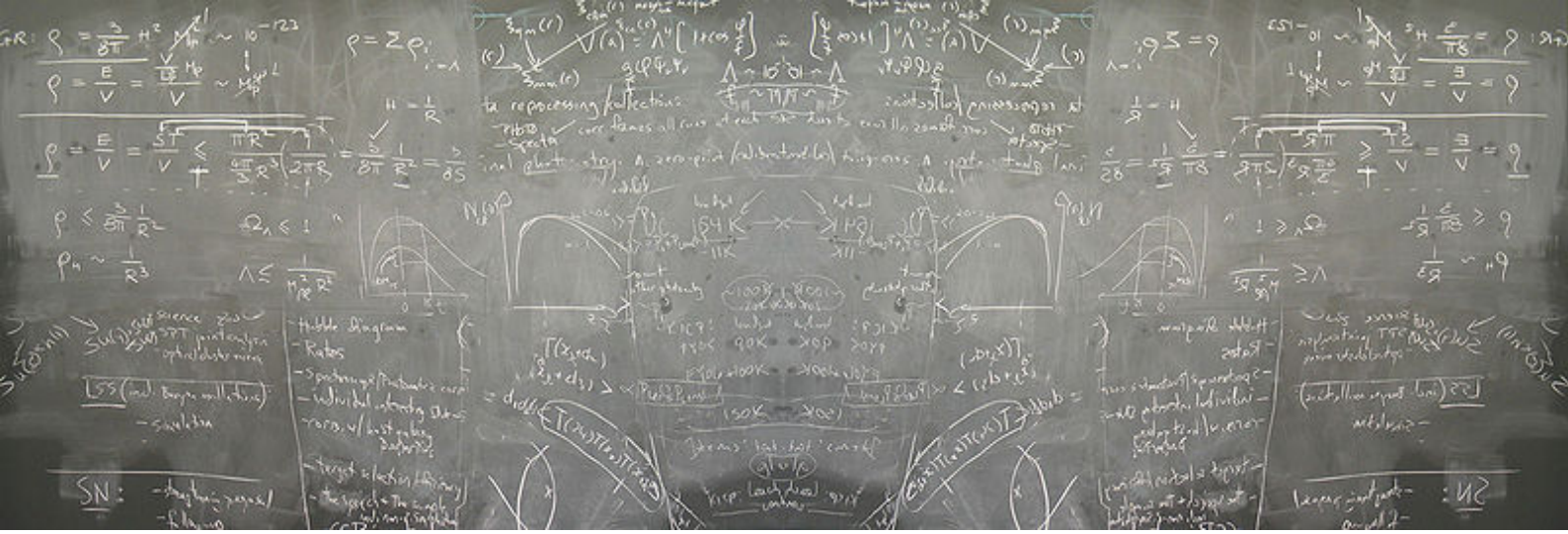
4.1.1 Wichtige Funktionen 16

4.1.2 O-Notation 17

5 Kombinatorik, Laufzeiten 23

5.1 Laufzeiten 23

Bibliographie 29



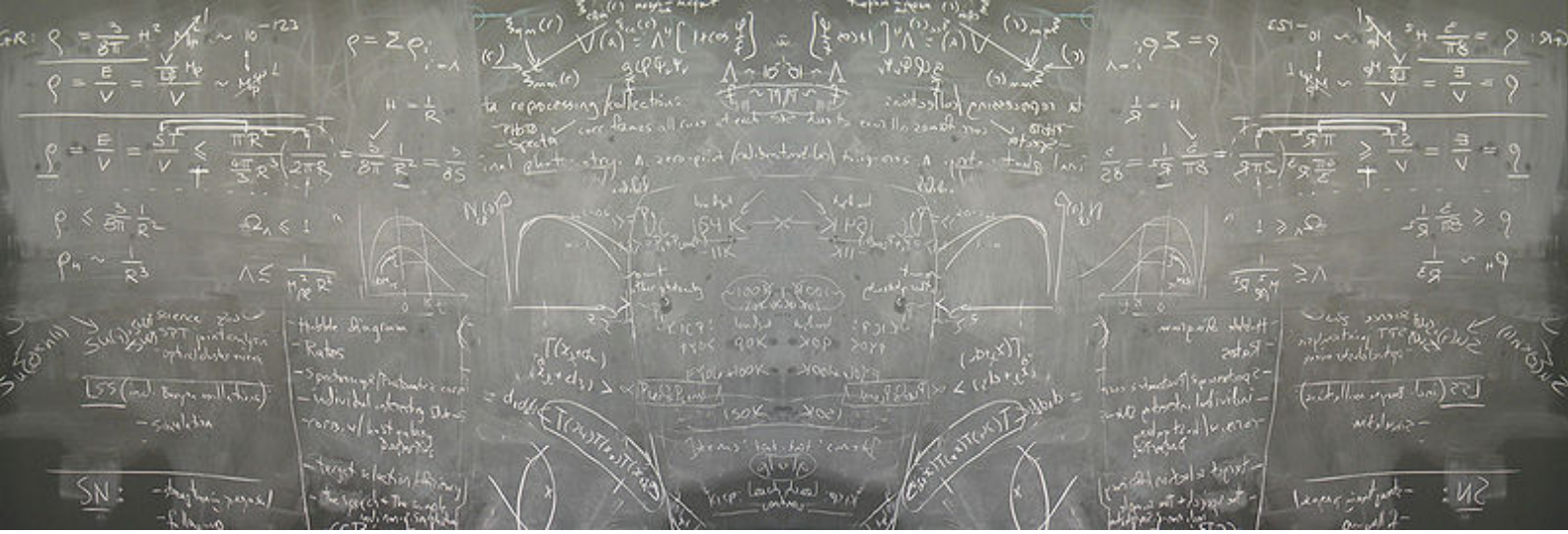
Herzlich Willkommen!

Herzlich Willkommen zur kurzen Mathe-Einführung, besonders geeignet für ~~Lehramts-Studierende~~ Zwei-Fach-Studierende[1], die im nächsten Semester *Algorithmen und Datenstrukturen (ADS)* hören werden. Wir werden versuchen, im Laufe des Semesters die dafür nötigen formalen Grundlagen der Hochschul-Mathematik zu entwickeln. Richtig Mathematik lernen kann man (wie auch Programmieren oder Schwimmen) nur durch reichlich Übung. Daher werden wir die kostbare Vorlesungszeit nicht dafür nutzen, Ihnen irgendwelche abstrakten Vorträge zu halten. Ganz im Gegenteil: Wir werden diese gemeinsame Zeit sinnvoll nutzen und versuchen, so viele Aufgaben wie möglich zu bearbeiten. Dafür müssen Sie sich natürlich im Vorhinein mit dem Stoff beschäftigt haben. Also gibt es zu jedem Themengebiet eine Kapitel-Auswahl, die *vor dem Vorlesungstermin* von Ihnen gelesen und verstanden werden sollte. Wir klären zwar zu Anfang der Vorlesung etwaige Fragen, wollen uns aber so schnell wie möglich auf die Aufgaben stürzen.

Über Hinweise und Verbesserungsvorschläge freuen wir uns natürlich immer sehr.

Im Text gibt es an einigen Stellen Kommentare. Diese sehen wie folgt aus und geben ergänzende Informationen.

[1] Hier wurden bisher die Nicht-Lehrämter ausgegrenzt. Das war nicht meine Absicht, entschuldigt!



1. Mengen und Beweise

Zum Lesen:
Kapitel 2.1, 2.2, 2.4, 2.5 in [Wol17].

Stichworte:

- Sachverhalt: Kontext; Wahrheitswert
- Definition: Begriffsbildung; *Textersetzung*
- Menge: Urelemente; Zusammenfassung von bestimmten, wohlunterschiedenen Objekten; Leere Menge $\emptyset$; Explizite Darstellung $\{0,1,2,3\}$; Deskriptive Darstellung $\{x \mid x \in \mathbb{N}, x \leq 3\}$; Mengen, die Mengen enthalten; $\in$-Symbol; Operatoren $\cup$, $\cap$, $\setminus$
- Beweis: Zwingende Argumentation; *Akt der Kommunikation*; Annahme und Beweisverpflichtung; Finden (Verfechter); Verifizieren (Skeptiker)

Definition 1.1 Wir definieren uns ein paar sehr nützliche Hilfsmengen:

- Die Menge der natürlichen Zahlen $\mathbb{N} = \{0,1,2,\ldots\}$[a].
- Die Menge der ganzen Zahlen $\mathbb{Z} = \{0,-1,1,-2,2\} = \mathbb{N} \cup \{-x | x \in \mathbb{N}\}$.
- die Menge der rationalen Zahlen $\mathbb{Q} = \{p/q \mid p,q \in \mathbb{Z}\}$.
- die Menge der reellen Zahlen $\mathbb{R}$.

[a] Ob die 0 eine natürliche Zahl ist, führt immer zu Diskussionen, insbesondere zwischen Mathematikern und Informatikern. Da wir hier in einem Informatik-nahen Kontext sind, gilt bei uns $0 \in \mathbb{N}$!

Aufgaben

Aufgabe 1.1 — Vorrechnen. Wir betrachten die beiden Mengen $M = \{x \mid x \in \mathbb{N}, 2^x \leq 10\}$ und $N = \{y \mid y \in \mathbb{N}, \text{Es gibt } z \in \mathbb{N} \text{ mit } y = z^2 \text{ und } z \leq 5\}$. Geben Sie die beiden Mengen M und N sowie $M \cup N$, $M \cap N$ und $M \setminus N$ explizit an, d. h. durch die in Mengenklammern eingeschlossene Aufzählung ihrer Elemente. ■

Aufgabe 1.2 — Vorrechnen. Zeigen Sie, dass $\sqrt{3} \notin \mathbb{Q}$, also irrational ist. ■

Aufgabe 1.3 Welche der folgenden Sätze sind Sachverhalte?

(i) Kiel liegt am 01.01.2019 in Schleswig-Holstein.
(ii) Kiel liegt am 01.01.2019 im Freistaat Bayern.
(iii) Mathe ist schwer.
(iv) Sebastian (Autor dieser Unterlagen) findet, dass Mathe schwer ist.
(v) $x^2 = 4$.
(vi) $42 = 2 \cdot 3 \cdot 7$.
(vii) $43 = 2 \cdot 3 \cdot 7$.
(viii) $43 \in 0$.

■

Aufgabe 1.4 Sei $n \in \mathbb{N}$. Spezifizieren Sie deskriptiv eine Menge T_n, die genau die positiven ganzzahligen Teiler von n enthält, d .h. jede Zahl $i \in \mathbb{N}$, für die $n/i \in \mathbb{N}$ gilt. ■

Aufgabe 1.5 Seien M, N und K Mengen mit $M \subseteq N$. Beweisen Sie, dass (i) $M \cup K \subseteq N \cup K$ und (ii) $M \cap K \subseteq N \cap K$ gilt. ■

Aufgabe 1.6 Beweisen Sie, dass jede durch 14 teilbare natürliche Zahl auch bereits durch 7 teilbar ist. ■

Aufgabe 1.7 Entscheiden Sie, ob die folgenden Aussagen wahr oder falsch sind, indem Sie entweder einen Beweis oder ein Gegenbeispiel angeben:

(i) Für alle Mengen M, N und K gilt $(M \setminus N) \setminus K = M \setminus (N \setminus K)$.
(ii) Für alle Mengen M, N und K mit $M \subseteq K$ und $N \subseteq K$ gilt $M \cup N \subseteq K$.

■

Aufgabe 1.8 Seien n und m natürliche Zahlen. Beweisen Sie, dass jede durch $n \cdot m$ teilbare natürliche Zahl auch durch n teilbar ist. ■

Aufgabe 1.9 Sei p eine Primzahl. Zeigen Sie, dass $\sqrt{p} \notin \mathbb{Q}$, also irrational ist. ■

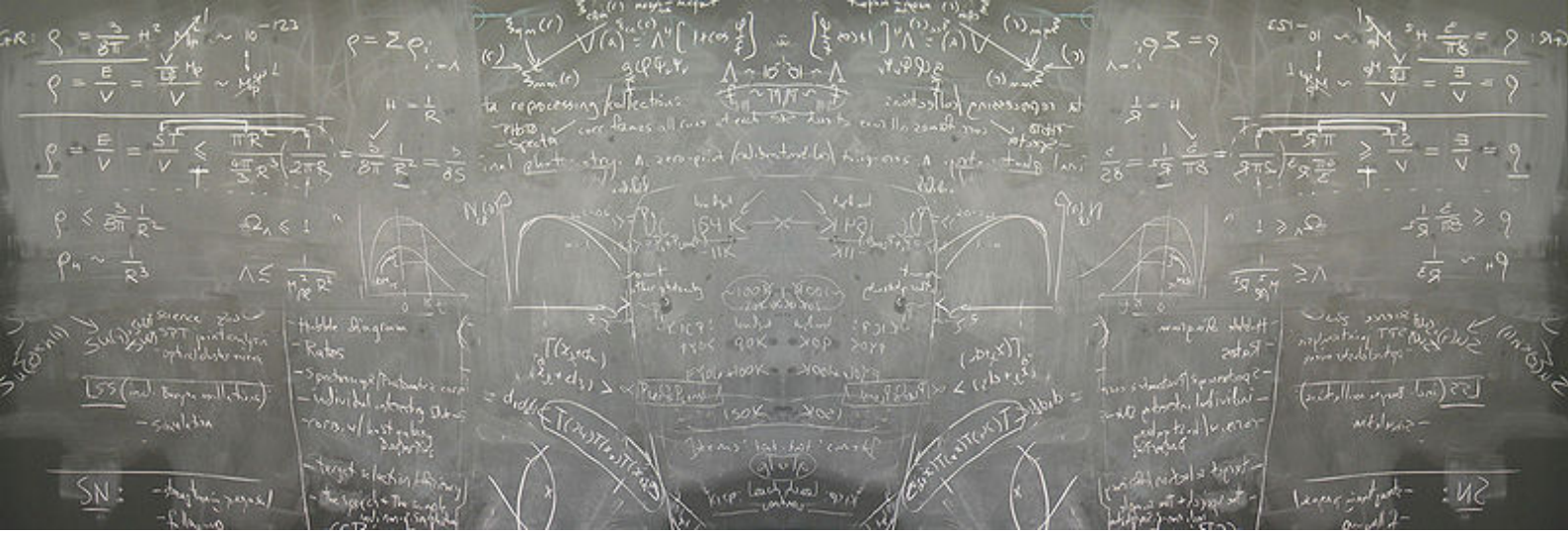
Aufgabe 1.10 Für eine Menge M ist $\mathscr{P}(M)$ die *Potenzmenge* von M, also die Menge, die alle Teilmengen von M enthält: $\mathscr{P}(M) = \{N \mid N \subseteq M\}$.

(i) Geben Sie $\mathscr{P}(\{1,2,3\})$ an.
(ii) Geben Sie $\mathscr{P}(\{\{1,2,3\}\})$ an.
(iii) Beweisen Sie, dass für alle Mengen M und N gilt, dass $\mathscr{P}(M \cap N) = \mathscr{P}(M) \cap \mathscr{P}(N)$.

■

Lernziele:

- Grundlegendes Verständnis für das Konzept einer Menge
- Sicherer Umgang mit expliziter und deskriptiver Mengendarstellung
- Sicherer Umgang mit grundlegenden Mengenoperationen wie $\cup$, $\cap$ und $\setminus$
- Beurteilen können, wann es sich um einen Sachverhalt handelt
- Das Prinzip eines Beweises verstanden haben
- Erste Beweise alleine führen können



2. Aussagenlogik, Prädikatenlogik

Zum Lesen:
Kapitel 2.3, 2.6, 3.3, 3.4 in [Wol17].

Stichworte:

- Zusammenfügung von Aussagen
- Kontext und Wahrheitsgehalt; Extensionalität; Verschiedene Welten
- Operatoren: *und*, *oder*, *nicht*, *entweder-oder*, *genau-dann-wenn*, *wenn-dann*
- Wahrheitstabellen
- Quantoren: *Für-alle*, *Es-gibt-ein*; Quantorenwechsel; Reihenfolge
- Unbestimmte Symbole vs. konkrete Werte

Definition 2.1 Die Operationen und Quantoren im Buch sind häufig auch durch andere Symbole dargestellt:

- Die *oder*-Verknüpfung "or" schreibt man auch als $\vee$, also
- Die *und*-Verknüpfung "and" schreibt man auch als $\wedge$.
- Die *nicht*-Verknüpfung "not" schreibt man auch als $\neg$.
- Die *entweder-oder*-Verknüpfung "xor" schreibt man auch als $\oplus$.
- Die *genau-dann-wenn*-Verknüpfung schreibt man auch als $\Leftrightarrow$.
- Die *wenn-dann*-Verknüpfung schreibt man auch als $\Rightarrow$.
- Der *Für-alle*-Quantor wird auch als $\forall$ geschrieben.
- Der *Es-gibt-ein*-Quantor wird auch als $\exists$ geschrieben.

Aufgaben

Aufgabe 2.1 — Vorrechnen. Beweisen Sie: Es gibt eine Menge M, so dass für alle x gilt, dass $x \in M$ impliziert, dass $x < 4$. ∎

Aufgabe 2.2 — Vorrechnen. Beweisen Sie: Wenn A und B Aussagen sind, so gilt: WG(Wenn A, dann B) = WG(Nicht A oder B). ∎

Aufgabe 2.3 Beweisen Sie: Für alle Mengen M, N und K gilt, dass $(M \cup N) \cap K = (M \cap K) \cup (N \cap K)$. ■

Aufgabe 2.4 Beweisen Sie: Wenn A und B Aussagen sind, so gilt: WG(A genau dann, wenn B) = WG((Wenn A, dann B) und (Wenn B, dann A)). ■

Aufgabe 2.5 Beweisen Sie: Für alle $x \in \mathbb{N}$ gibt es ein $y \in \mathbb{N}$, so dass $y > x$. ■

Aufgabe 2.6 Beweisen Sie: Es gibt eine Menge M, so dass für alle Mengen N gilt, dass $M \cup N = N$. ■

Aufgabe 2.7 Widerlegen Sie: Es gibt eine Menge M, so dass für alle Mengen N gilt, dass $M \cup N = M$. ■

Aufgabe 2.8 Beweisen Sie: Es gibt eine natürliche Zahl $n_0 \in \mathbb{N}$, so dass für alle natürlichen Zahlen $n \in \mathbb{N}$ gilt: wenn $n \geq n_0$, dann $4n^3 + 1 \leq n^4$. ■

Aufgabe 2.9 Beweisen Sie: Für alle Mengen M und N gibt es eine Menge K mit $M \subseteq K$ und $N \subseteq K$. ■

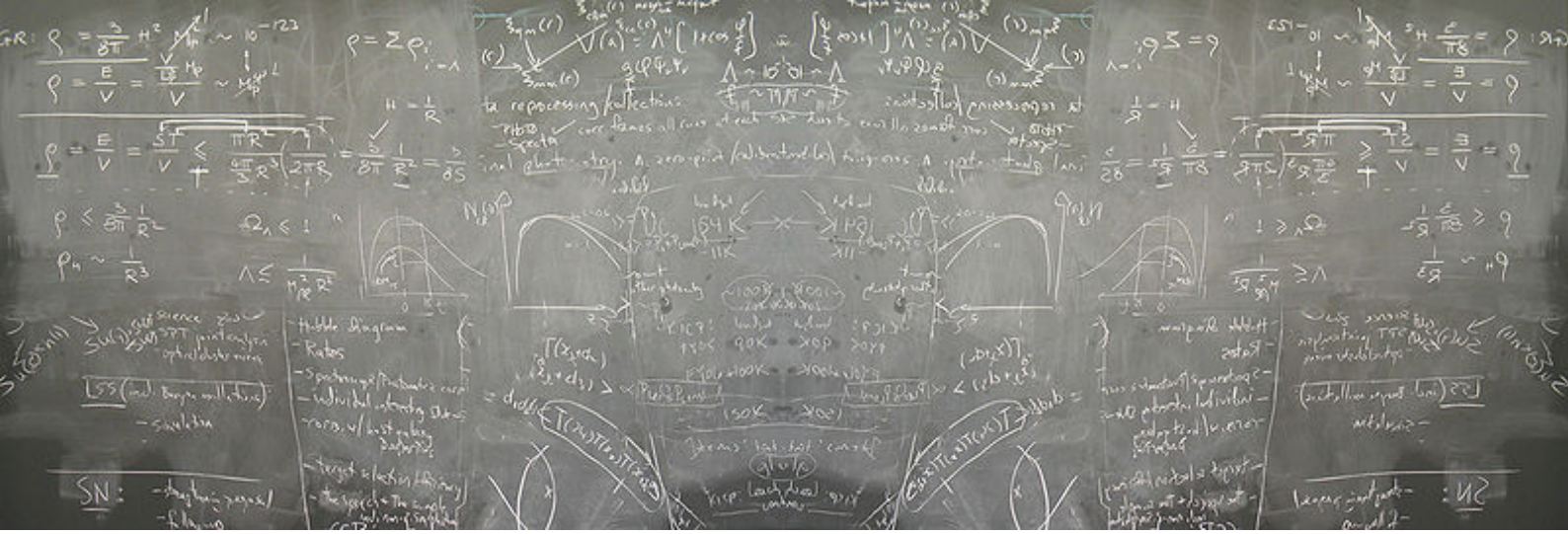
Aufgabe 2.10 Beweisen Sie: Für alle Zahlen $x \in \mathbb{N}$ gibt es eine natürliche Zahl $y \in \mathbb{N}$ so dass aus $x > 0$ folgt, dass $y < x$. ■

Aufgabe 2.11 Beweisen Sie: Für alle $x \in \mathbb{N}$ und alle $y \in \mathbb{N}$ gibt es eine Menge M, so dass für alle $z \in \mathbb{N}$ gilt: Ist $z \neq x$ und $z \neq y$, so gilt $x \in M$, $y \in M$ und $z \notin M$. ■

Aufgabe 2.12 Beweisen Sie: Für alle Mengen M und N gilt, dass $M \subseteq N$ genau dann wenn $M \cap N = M$. ■

Lernziele:

- Grundlegendes Verständnis für die elementaren Operatoren zum Verknüpfen von Aussagen
- Grundlegendes Verständnis für Quantoren und deren Reihenfolge
- Sicherer Umgang mit dem Beweisen von Aussagen, die Quantoren enthalten
- Wahrheitstabellen interpretieren können
- Die Welt der Aussagen und die Welt der Wahrheitswerte unterscheiden können



3. Folgen, Induktion, Rekurrenzen

Zum Lesen:
Kapitel 3.7 in [Wol17]
Kapitel 8.1, 8.2, 8.3, 8.4, 8.5 in [Bel18]

Stichworte:

- Aussagen über alle hinreichend große natürliche Zahlen
- Induktionsanfang, Induktionsvoraussetzung, Induktionsbehauptung, Induktionsschritt
- Fehlerhafte Argumentation bei "untypischen" Fällen
- Sequenzen von natürlichen Zahlen: Rekursive Definition / Rekurrenzgleichung vs. geschlossene Form
- Initiale Werte für Rekurrenzen
- Typischer Induktionsbeweis für Rekurrenzen
- Fibonacci-Zahlen F_n mit $F_0 = 1$, $F_1 = 1$ und $F_n = F_{n-1} + F_{n-2}$

Definition 3.1 In [Bel18] wird die *Summennotation* $\sum_{i=1}^{n} a_i$ genutzt. Dies ist einfach eine kompakte Schreibweise für $a_1 + a_2 + \ldots + a_n$. Zum Beispiel ist $\sum_{i=1}^{10} i = 1 + 2 + \ldots + 10$ oder $(1^2 - 1) + (2^2 - 1) + (3^2 - 1) + \ldots + (42^2 - 1) = \sum_{i=1}^{42} i^2 - 1$.

Definition 3.2 In beiden Büchern wird eine leicht vereinfachte Form der Induktion genutzt, die man manchmal auch *schwache Induktion* nennt. Um $H(k+1)$ zu beweisen, wird hierbei $H(k)$ vorausgesetzt. In der *starken Induktion*, nimmt man statt $H(k)$ *alle Aussagen* $H(0), H(1), \ldots, H(k)$ als Induktionsvoraussetzung. Häufig kann man dadurch seine Beweise etwas vereinfachen.

Aufgaben

Aufgabe 3.1 — Vorrechnen. Beweisen Sie: Für alle $n \in \mathbb{N}$ gilt, dass $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$. ■

Aufgabe 3.2 — Vorrechnen. Gegeben sei die Folge a_n mit $a_0 = 7$ und $a_n = a_{n-1} + 2$. Finden Sie eine geschlossene Form für a_n. ■

Der Bottom-Up-Ansatz gibt uns $a_1 = a_0 + 2 = 7 + 2$, $a_2 = a_1 + 2 = 7 + 2 + 2$, $a_3 = a_2 + 2 = 7 + 2 + 2 + 2$. Es sieht also nach $a_n = 7 + 2n$ aus.

Der Top-Down-Ansatz gibt uns

$$a_n = a_{n-1} + 2 = a_{n-2} + 2 + 2 = a_{n-3} + 2 + 2 + 2 = \ldots = a_0 + 2n = 7 + 2n.$$

Wir behaupten also $a_n = 2n + 7$.

Aufgabe 3.3 Beweisen Sie: Für alle $n \in \mathbb{N}$ und alle reellen Zahlen $q \in \mathbb{R}$ mit $q \neq 1$ gilt, dass $\sum_{i=0}^{n} q^i = \frac{q^{n+1}-1}{q-1}$. ∎

Aufgabe 3.4 Gegeben sei die Folge a_n mit $a_0 = 3$ und $a_n = 2a_{n-1}$. Finden Sie eine geschlossene Form für a_n. ∎

Der Bottom-Up-Ansatz gibt uns $a_1 = 2 \cdot a_0 = 2 \cdot 3$, $a_2 = 2 \cdot a_1 = 2 \cdot 2 \cdot 3$ und $a_3 = 2a_2 = 2 \cdot 2 \cdot 2 \cdot 3$, also vermuten wir $a_n = 3 \cdot 2^n$.

Der Top-Down-Ansatz gibt uns

$$a_n = 2a_{n-1} = 2 \cdot 2a_{n-2} = 2 \cdot 2 \cdot 2a_{n-3} = \ldots = 2^n a_0 = 3 \cdot 2^n.$$

Wir behaupten also, dass $a_n = 3 \cdot 2^n$.

Aufgabe 3.5 Beweisen Sie: Für alle $n \in \mathbb{N}$ gilt, dass $F_n \leq 2^n$. ∎

Aufgabe 3.6 Gegeben sei die Folge a_n mit $a_0 = 1$, $a_1 = 2$ und $a_n = a_{n-1} + 2a_{n-2}$. Finden Sie eine geschlossene Form für a_n. ∎

Der Bottom-Up-Ansatz gibt uns $a_2 = a_1 + 2a_0 = 2 + 2 \cdot 1 = 4 = 2^2$, $a_3 = a_2 + 2a_1 = 4 + 2 \cdot 2 = 8 = 2^3$, $a_4 = a_3 + 2a_2 = 8 + 2 \cdot 4 = 16 = 2^4$. Wir vermuten also $a_n = 2^n$.

Der Top-Down-Ansatz ist hier wenig hilfreich. Zumindest sehe ich keinen hilfreichen Ansatz.

Aufgabe 3.7 Beweisen Sie: Für alle $n \in \mathbb{N}$ gilt, dass $2n^3 + 3n^2 + n$ durch 6 teilbar ist. ∎

Aufgabe 3.8 Gegeben sei die Folge a_n mit $a_0 = 2$ und $a_n = 2a_{n-1} + 1$. Finden Sie eine geschlossene Form für a_n. ∎

Der Bottom-Up-Ansatz gibt uns $a_1 = 2a_0 + 1 = 2 \cdot 2 + 1$, $a_2 = 2a_1 + 1 = 2 \cdot (2 \cdot 2 + 1) + 1 = 2 \cdot 2 \cdot 2 + 2 + 1$, $a_3 = 2a_2 + 1 = 2(2 \cdot 2 \cdot 2 + 2 + 1) + 1 = 2 \cdot 2 \cdot 2 \cdot 2 + 2 \cdot 2 + 2 + 1$. Hier sehen wir nicht direkt ein, wie sich die Folge verändert.

Der Top-Down-Ansatz gibt uns

$$a_n = 2a_{n-1} + 1 = 2 \cdot (2a_{n-2} + 1) + 1 = 2 \cdot 2 \cdot a_{n-2} + 3 = 2 \cdot 2 \cdot (2a_{n-3} + 1) + 3 =$$
$$2 \cdot 2 \cdot 2a_{n-3} + 4 + 3 = 2 \cdot 2 \cdot 2 \cdot (2a_{n-4} + 1) + 7 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot a_{n-4} + 15 = \ldots =$$
$$2^n a_0 + 2^{n-1} - 1 = 2^{n+1} + 2^n - 1.$$

Wir vermuten also $a_n = 2^{n+1} + 2^n - 1 = 2^n(2+1) - 1 = 3 \cdot 2^n - 1$. Hier sieht man, dass wir fast die gleiche geschlossene Form wie oben haben, aber die Struktur viel schwerer aussieht.

Aufgabe 3.9 Beweisen Sie: Für alle $n \in \mathbb{N}$ gilt, dass $\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$. ∎

Aufgabe 3.10 Gegeben sei die Folge a_n mit $a_0 = 1$ und $a_n = a_{n-1} + 6n - 3$. Finden Sie eine geschlossene Form für a_n. ■

Der Bottom-Up-Ansatz gibt $a_1 = a_0 + 6 - 3 = 1 + 6 - 3 = 4$, $a_2 = a_1 + 12 - 3 = 4 + 12 - 3 = 13$, $a_3 = a_2 + 18 - 3 = 13 + 18 - 3 = 28$. Hier sieht man nicht so viel.

Der Top-Down-Ansatz ist hilfreicher und gibt

$$a_n = a_{n-1} + 6n - 3 = a_{n-2} + 6(n-1) - 3 + 6n - 3 = a_{n-2} + 6(n + (n-1)) - 2 \cdot 3 =$$
$$a_{n-3} + 6(n + (n-1) + (n-2)) - 3 \cdot 3 = \ldots = a_0 + 6(\sum_{i=1}^{n} i) - 3n = 1 + 6(\sum_{i=1}^{n} i) - 3n.$$

Wir wissen, dass $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, also gilt $1 + 6(\sum_{i=1}^{n} i) - 3n = 1 + 3n(n+1) - 3n = 3n^2 + 1$. Also vermuten wir, dass $a_n = 3n^2 + 1$ gilt.

Aufgabe 3.11 Beweisen Sie: Für alle $n \in \mathbb{N}$ mit $n \geq 6$ gilt, dass $F_n \geq (3/2)^n$. ■

Aufgabe 3.12 Gegeben sei die Folge a_n mit $a_0 = 2$ und $a_n = a_{n-1} + 21n^2 - 21n + 7$. Finden Sie eine geschlossene Form für a_n. ■

Der Bottom-Up-Ansatz gibt $a_1 = 2 + 21 - 21 + 7 = 9$, $a_2 = 9 + 21 \cdot 4 - 21 \cdot 2 + 7 = 58$, $a_3 = 58 + 21 \cdot 9 - 21 \cdot 3 + 7 = 191$. Hier sehen wir nicht allzuviel.

Beim Top-Down-Ansatz ergibt sich:

$$a_n = a_{n-1} + 21n^2 - 21n + 7 = a_{n-2} + 21(n-1)^2 - 21(n-1) + 7 + 21n^2 - 21n + 7 =$$
$$a_{n-3} + 21(n-2)^2 - 21(n-2) + 7 + 21(n-1)^2 - 21(n-1) + 7 + 21n^2 - 21n + 7 = \ldots =$$
$$a_0 + 21(\sum_{i=1}^{n} i^2) - 21(\sum_{i=1}^{n} i) + 7n = 2 + 21(\sum_{i=1}^{n} i^2) - 21(\sum_{i=1}^{n} i) + 7n.$$

Nun wissen wir, dass $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ und $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ gilt. Also können wir den obigen Term auch schreiben als

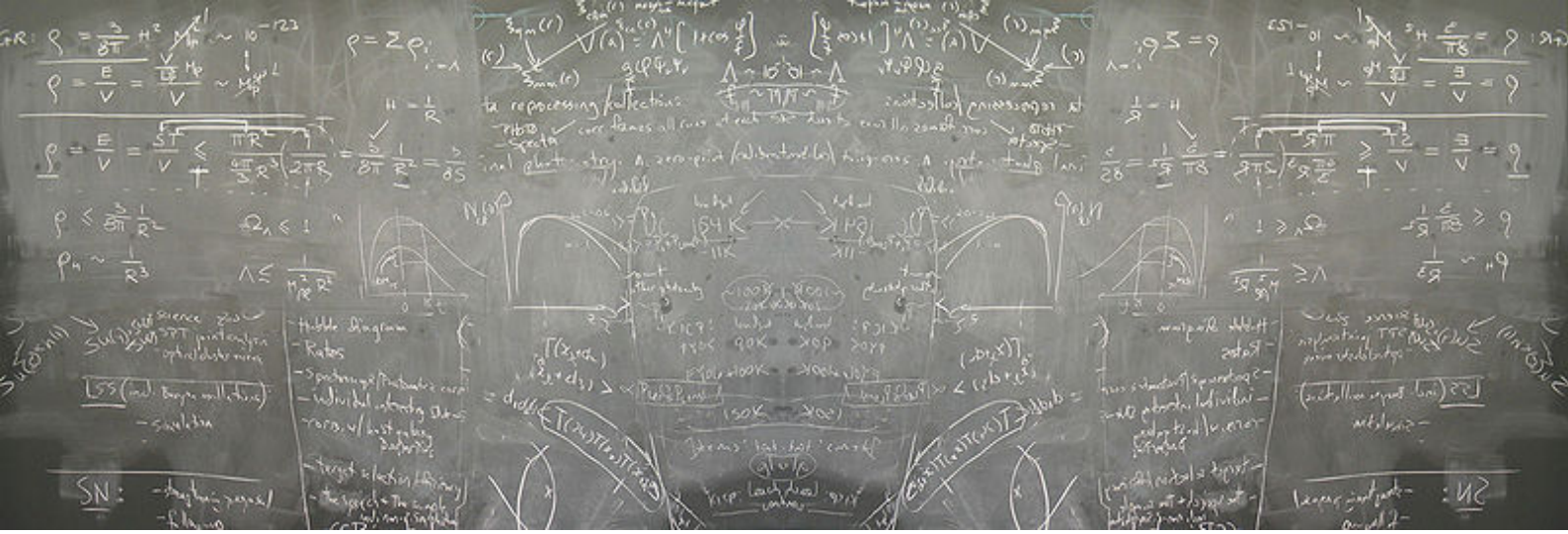
$$2 + 21(\sum_{i=1}^{n} i^2) - 21(\sum_{i=1}^{n} i) + 7n = 2 + 21 \cdot \frac{n(n+1)(2n+1)}{6} - 21 \cdot \frac{n(n+1)}{2} + 7n =$$
$$2 + 21 \cdot \frac{2n^3 + 3n^2 + n}{6} - 21 \cdot \frac{n^2 + n}{2} + 7n =$$
$$2 + \frac{42n^3 + 63n^2 + 21n}{6} - \frac{21n^2 + 21n}{2} + 7n =$$
$$2 + 7n^3 + (63/6)n^2 + (21/6)n - (21/2)n^2 - (21/2)n + 7n =$$

Nun gilt $63/6 = 21/2$, also fallen die quadratischen Terme weg. Weiterhin gilt $(21/6) + 7 = (21/6) + (42/6) = (63/6) = (21/2)$, also fallen auch die linearen Terme weg und es bleibt $7n^3 + 2$ übrig. Wir vermuten also $a_n = 7n^3 + 2$.

Lernziele:

- Sicherer Umgang mit Induktionsbeweisen
- Gutes Verständnis für die Unterteilung von Induktionsbeweisen in Anfang, Voraussetzung und Schritt
- Grundlegendes Verständnis von Folgen und Reihen
- Sicherer Umgang mit rekursiv definierten Folgen

- Rekursive Definition und geschlossene Form unterscheiden können
- Erste Übungen im systematischen Lösen von Rekurrenzgleichungen



4. Funktionen, O-Notation, Modulare Arithmetik

Zum Lesen:
Kapitel 3.1, 3.2, 5.3 in [Bel18]
Ausarbeitung von Sebastian zu Funktionen (also dieses Dokument)

Stichworte:

- Funktionen; Domain (Definitionsmenge, Urbildmenge); Target (Zielmenge); Range (Bildmenge)
- Eigenschaften von Funktionen: injektiv, surjektiv, bijektiv
- Grundlegende Funktionen: Polynome, Exponentialfunktion, Logarithmus; Rechenregeln
- Modulare Arithmetik; Teilbarkeit; Rest
- Äquivalenzklassen; Symmetrie; Transitivität; Reflexivität; Partition
- O-Notation zur Bestimmung des Wachstums von Funktionen

Definition 4.1 Für eine reelle Zahl $x \in \mathbb{R}$ ist $\lfloor x \rfloor$ die größte ganze Zahl kleiner oder gleich x und $\lceil x \rceil$ die kleinste ganze Zahl größer oder gleich x. Also gilt $\lfloor 4{,}5 \rfloor = 4$ und $\lceil 4{,}5 \rceil = 5$. Eselsbrücke: $\lceil \cdot \rceil$ sieht aus wie ein Dach und $\lfloor \cdot \rfloor$ wie der Fußboden.

Definition 4.2 Für die Modulo-Operation wird in [Bel18] die Notation $a \equiv b \pmod{m}$ benutzt, wenn a geteilt durch n den gleichen Rest hinterlässt wie b geteilt durch n. Dies ist die typische Schreibweise in der Mathematik, aber in der Informatik schreibt man auch $a \bmod m = b$, und interpretiert $\bmod m$ also als Funktion.

4.1 Funktionen und O-Notation

Im folgenden Abschnitt stellen wir zunächst die wichtigsten speziellen Funktionen vor, die wir so brauchen werden. Zusätzlich werden wir die wichtigsten Rechenregeln für diese Funktionen wiederholen. Danach geht es darum, das Wachstum dieser Funktionen zu vergleichen. Dieser Vergleich geschieht mithilfe der sogenannten *O-Notation*.

4.1.1 Wichtige Funktionen

Polynome

Eine einfache Klasse von Funktionen sind die sogenannten *Polynome*. Eine solche Funktion $f\colon \mathbb{R} \to \mathbb{R}$ lässt sich darstellen als $f(x) = \sum_{i=0}^{n} a_i x^i$, also haben wir $f(x) = a_0 + a_1 x^1 + a_2 x^2 + \ldots + a_n x^n$. Hierbei nennen wir die reellen Zahlen a_i die *Koeffizienten* des Polynoms.

▪ Beispiel 4.3

- Die Funktion $f\colon \mathbb{R} \to \mathbb{R}$ mit $f(x) = 2x+1$ ist ein Polynom. Hierbei ist $n = 1$, $a_0 = 1$ und $a_1 = 2$.
- Auch die Funktion $f\colon \mathbb{R} \to \mathbb{R}$ mit $f(x) = x^3 + 4x^2 - 7x + (1/2)$ ist ein Polynom mit $n = 3$, $a_0 = 1/2$, $a_1 = -7$, $a_2 = 4$ und $a_3 = 1$.

◇

Polynome sind die typischen Funktionen, mit denen in der Schule gerechnet wird. Der *Grad* eines Polynoms $f\colon \mathbb{R} \to \mathbb{R}$ mit $f(x) = \sum_{i=0}^{n} a_i x^i$ ist n und somit die höchste vorkommende Potenz von x. Multiplikation und Addition funktionieren genauso, wie wir es bereits aus der Schule gewohnt sind: Bei der Addition addieren wir einfach die Koeffizienten; bei der Multiplikation müssen wir für die Potenz x^i alle Koeffizienten a_j und b_{i-j} multiplizieren und dann aufaddieren.

▪ Beispiel 4.4 — Polynom-Multiplikation. Mit $f(x) = 2x+1$ und $g(x) = 7x^2 + x + 3$ gilt also $f(x) + g(x) = 7x^2 + 3x + 4$ und

$$f(x) \cdot g(x) = (2x+1)(7x^2+x+3) = 14x^3 + 2x^2 + 6x + 7x^2 + x + 3 = 14x^3 + 9x^2 + 7x + 3.$$

Der Term $9x^2$ ergibt sich also daraus, dass wir den konstanten Term 1 von f mit dem quadratischen Term $7x^2$ von g multiplizieren und den linearen Term $2x$ von f mit dem linearen Term x von g multiplizieren. Letztendlich addieren wir dann die beiden Terme $1 \cdot 7x^2 + 2x \cdot x = 7x^2 + 2x^2 = 9x^2$ und erhalten so den quadratischen Term von $f(x) \cdot g(x)$.

Der Term a_0 heißt *konstanter* Term, der Term $a_1 x$ heißt *linearer* Term, der Term $a_2 x^2$ heißt *quadratischer* Term und $a_3 x^3$ heißt *kubischer* Term. ◇

Polynome sind auch die wichtigsten Funktionen, die uns bei der Analyse von Algorithmen und Datenstrukturen interessieren werden. Ein sicherer Umgang mit ihnen ist also unerlässlich. Wer sich also noch nicht sicher im Umgang damit fühlt, der möge sich ein paar Polynome ausdenken und mit ihnen Rechnen üben. Auch findet man weitere interessante Beispiele und Informationen bei Wikipedia.

Zusätzlich zu den Polynomen gibt es jedoch noch zwei andere Arten von Funktionen, auf die wir immer wieder treffen werden, die *Exponentialfunktion* und ihr Inverses, der *Logarithmus*.

Exponentialfunktion

Für eine reelle Zahl $b \in \mathbb{R}$ (die sogenannte *Basis*) und eine beliebige Zahl $x \in \mathbb{R}$ (der *Exponent*) können wir die Zahl b^x berechnen. In der Informatik ist x meistens eine natürliche Zahl und wir können uns dann b^x als x-fache Multiplikation von b vorstellen, es gilt also

$$b^x = \underbrace{b \cdot b \cdot \ldots \cdot b}_{x\text{-mal}}.$$

Damit können wir sehr leicht eine der grundlegenden Regeln für die Exponentialfunktion sehen:

$$b^x \cdot b^y = \underbrace{b \cdot b \cdot \ldots \cdot b}_{x\text{-mal}} \cdot \underbrace{b \cdot b \cdot \ldots \cdot b}_{y\text{-mal}} = \underbrace{b \cdot b \cdot \ldots \cdot b}_{x+y\text{-mal}} = b^{x+y}.$$

Aber auch für nicht-natürliche x kann man den Ausdruck b^x definieren und es gelten folgende wichtige Rechenregeln:

(i) Es gilt $b^0 = 1$ für alle $b \neq 0$.
(ii) Es gilt $0^x = 0$ für all $x \neq 0$.
(iii) Der Wert 0^0 ist nicht sinnvoll definiert, also ignorieren wir ihn einfach.
(iv) Es gilt $b^x \cdot b^y = b^{x+y}$ für alle Zahlen x, y.
(v) Es gilt $(b^x)^y = b^{x \cdot y}$ für alle Zahlen x, y. Wichtig hierbei ist, dass $(b^x)^y$ etwas anderes als $b^{x^y} = b^{(x^y)}$ ist. Für $b = 2$, $x = 2$ und $y = 3$ ist $(b^x)^y = (2^2)^3 = 4^3 = 64$, aber $b^{x^y} = 2^{2^3} = 2^8 = 256$.

Logarithmus

Wenn man nun eine positive reelle Basis b und eine positive reelle Zahl y gegeben hat, stellt sich die Frage, ob es denn immer eine Zahl x gibt, so dass $b^x = y$ gilt. Und in der Tat gibt es immer eine solche Zahl, die wir als $\log_b(y)$ den *Logarithmus von y zur Basis b*. Es gilt also immer $b^{\log_b(x)} = x$. Behält man diese Definition im Kopf, kommt man schon ziemlich weit. Zum Beispiel können wir dann auch die wichtigste Rechenregel des Logarithmus sehr einfach herleiten:

$$\log_b(b^x \cdot b^y) = \log_b(b^{x+y}) = x + y = \log_b(b^x) + \log_b(b^y).$$

Also gilt $\log_b(r \cdot s) = \log_b(r) + \log_b(s)$ für alle Zahlen r, s. Daraus kann man dann auch leicht sehen, dass $\log_b(x^y) = y \cdot \log_b(x)$ gilt, Exponenten kann man also vor den Logarithmus ziehen.

Die wichtigste Basis in der Informatik ist natürlich $b = 2$, weshalb man auch häufig einfach log statt $\log_2$ schreibt. Diese Funktion nennt man den *binären Logarithmus* (selten auch: logarithmus dualis). Wie beschrieben, rechnen wir in der Informatik meist mit ganzen Zahlen. Das heißt, wir betrachten nicht den Wert $\log_2(15) \approx 3.9068905956085187$, sondern vielmehr $\lceil \log_2(15) \rceil = 4$. Diese Funktion hat zwei andere, einfache Interpretationen. Zunächst einmal beschreibt $\lceil \log_2(x+1) \rceil$ die Anzahl an Bits, die man braucht, um x in Binärdarstellung zu schreiben. Denn 15 in Binärdarstellung ist $(1111)_2$ und hat somit Länge $\lceil \log(15+1) \rceil = \lceil \log(16) \rceil = 4$. Die 16 hat hingegen die Binärdarstellung (10000) und somit Länge $\lceil \log(16+1) \rceil = \lceil \log(17) \rceil = 5$. Andererseits gibt $\lceil \log_2(x) \rceil - 1$ an, wie häufig man x ganzzahlig durch 2 teilen muss, bis man bei der 1 angekommen ist. Zum Beispiel gibt einem ganzzahliges Teilen von 15 durch 2 die Zahl 7. Ganzzahliges Teilen von 7 durch 2 gibt 3 und ganzzahliges Teilen von 3 durch 2 ergibt 1. Also mussten wir die 15 dreimal teilen. Es gibt sehr, sehr viele Algorithmen, in denen wir ein Problem in zwei gleichgroße Teilprobleme zerlegen und diese dann einzeln lösen. Die *binäre Suche* ist sicherlich das bekannteste solche Verfahren. Aufgrund der Häufigkeit dieses Teilen-und-Herrschen-Ansatzes taucht auch der Logarithmus in vielen Laufzeitabschätzungen auf.

4.1.2 O-Notation

Im Folgenden betrachten wir nur noch Funktionen, die natürliche Zahlen auf natürliche Zahlen abbilden. Diese sind für die Analyse von Algorithmen besonders interessant. Wenn wir einen Algorithmus A haben, der irgendein Problem löst, möchten wir gerne wissen, wieviele Schritte er dafür braucht. Natürlich hängt die Anzahl dieser Schritte von der Größe der Eingabe ab. Um ein Array der Länge 20 zu sortieren, wird man viel weniger Schritte brauchen als für ein Array der Länge 42000000000000000000. Somit weisen wir also einer Eingabegröße, beschrieben durch eine natürliche Zahl n, die Anzahl an Schritten $T_{\mathsf{A}}(n)$ zu, die der Algorithmus A bei der Lösung eines Problemes mit Eingabegröße n braucht. Dies nennen wir die *Laufzeit* von A und wir werden uns noch intensiver damit beschäftigen. Wenn wir nun zwei Algorithmen A und B gegeben haben, möchten wir natürlich gerne entscheiden, welcher der beiden Algorithmen schneller ist, also ob $T_{\mathsf{A}}(n)$ kleiner als $T_{\mathsf{B}}(n)$ gilt. Häufig ist die Antwort darauf jedoch nicht so leicht. Für einige Fälle ist das völlig klar: Gilt $T_{\mathsf{A}}(n) = 3n+1$ und $T_{\mathsf{B}}(n) = 2n$, so gilt $T_{\mathsf{B}}(n) \leq T_{\mathsf{A}}(n)$ für alle $n \in \mathbb{N}$, also ist B schneller als A. Aber wie sieht es aus mit den Laufzeiten $T_{\mathsf{A}}(n) = n^2$ und $T_{\mathsf{B}}(n) = 4n+7$? Für $n \in \{0, 1, \ldots, 5\}$ gilt $T_{\mathsf{A}}(n) < T_{\mathsf{B}}(n)$ und für $n \geq 6$ gilt $T_{\mathsf{B}}(n) \leq T_{\mathsf{A}}(n)$. Aus einem intuitiven Standpunkt heraus betrachten wir T_{B} als die kleinere Funktion, denn der höchste auftretende

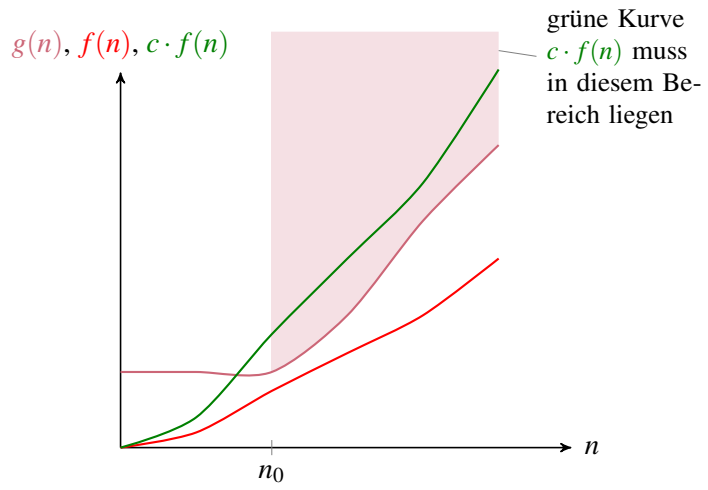


Abbildung 4.1: Skizze zur Definition der O-Notation

Exponent ist kleiner und ab einem bestimmten Zeitpunkt n_0 ist T_B auch wirklich immer kleiner als T_A. Die Einsicht, dass es einen solche Zeitpunkt n_0 gibt, führt uns zur Definition der O-Notation. Wir werden jedoch sogar noch etwas mehr Flexibilität einführen. Da die O-Notation für allgemeine Funktionen funktioniert (nicht nur für solche, die die Laufzeit von Algorithmen messen), werden wir uns im Folgenden wieder auf beliebige Funktionen, die natürliche Zahlen auf natürliche Zahlen abbilden, beschäftigen.

Definition 4.5 — O-Notation. Sei $f\colon \mathbb{N} \to \mathbb{N}$ eine Funktion. Die Menge $O(f)$ ist definiert als die Menge aller Funktionen $g\colon \mathbb{N} \to \mathbb{N}$, so dass es eine positive Zahl $c \in \mathbb{R}_{>0}$ und eine natürliche Zahl $n_0 \in \mathbb{N}$ gibt mit $g(n) \leq c \cdot f(n)$ für alle $n \geq n_0$.

In Formeln ausgedrückt, gilt also

$$O(f) = \{g\colon \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}_{>0}\ \exists n_0 \in \mathbb{N}\ \forall n \in \mathbb{N}, n \geq n_0 : g(n) \leq c \cdot f(n)\}.$$

Dies ist wahrscheinlich eine der kompliziertesten Formeln, die uns begegnen wird. Es ist aber sehr wichtig, sie genau zu verstehen!

Somit heißt $g \in O(f)$, dass es eine Konstante c gibt, so dass $c \cdot f(n)$ für genügend große n immer größer als $g(n)$ ist. Das Prinzip wird im Abbildung 4.1 noch einmal verdeutlicht: Es kann durchaus sein, dass $f(n) \leq g(n)$ gilt, solange $g(n) \leq c \cdot f(n)$ gilt.

Gucken wir uns zunächst einmal ein paar positive Beispiele an:

■ **Beispiel 4.6 —** $g \in O(f)$.

- Für $g(n) = 2n$ und $f(n) = 3n+1$ gilt $g \in O(f)$, denn wir können $c = 1$ und $n_0 = 0$ wählen und somit gilt $g(n) = 2n \leq 3n+1 = f(n) = c \cdot f(n)$ für alle $n \geq n_0 = 0$.
- Für $g(n) = 4n$ und $f(n) = n^2$ gilt $g \in O(f)$, denn wir können $c = 4$ und $n_0 = 0$ wählen und somit gilt $g(n) = 4n \leq 4 \cdot n^2 = c \cdot f(n)$ für alle $n \geq n_0 = 0$.
- Für $g(n) = 5n+2$ und $f(n) = n^2$ gilt $g \in O(f)$, denn wir können $c = 7$ und $n_0 = 1$ wählen und somit gilt $g(n) = 5n+2 \leq 5n+2n = 7n \leq 7n^2 = c \cdot f(n)$ für alle $n \geq n_0 = 0$.
- Für $g(n) = 100000000000n^2$ und $f(n) = n^3$ gilt $g \in O(f)$, denn wir können $c = 100000000000$ und $n_0 = 1$ wählen und somit gilt $g(n) = 100000000000n^2 \leq 100000000000n^3 = c \cdot f(n)$.
- Für $g(n) = n^2$ und $f(n) = 2^n$ gilt $g \in O(f)$. Wähle $c = 1$ und $n_0 = 4$. Nun müssen wir also zeigen, dass $n^2 \leq 2^n$ für alle $n \geq n_0$ gilt. Dies werden wir mithilfe von vollständiger Induktion beweisen:

Induktionsanfang: Für $n = 4$ gilt $n^2 = 4^2 = 16 \leq 16 = 2^4 = 2^n$, also $n^2 \leq 2^n$.
Induktionsvoraussetzung: Für ein festes $k \in \mathbb{N}$ mit $k \geq 4$ gelte $k^2 \leq 2^k$.
Induktionsschritt: Wir müssen nun zeigen, dass $(k+1)^2 \leq 2^k$. Es gilt $(k+1)^2 = k^2 + 2k + 1$. Aus den Aufgaben der letzten Woche wissen wir, dass $2k+1 \leq k^2$ für $k \geq 3$ gilt. Also folgt $(k+1)^2 \leq k^2 + k^2 \leq 2k^2$. Nach Induktionsvoraussetzung gilt $k^2 \leq 2^k$ und somit $2k^2 \leq 2 \cdot 2^k = 2^{k+1}$. Insgesamt gilt also $(k+1)^2 \leq 2^{k+1}$.

◇

Nun betrachten wir ein paar Beispiele, bei denen $g \notin O(f)$ liegt. Dazu müssen wir uns zunächst einmal überlegen, was dass denn bedeutet. Betrachten wir formal die Definition der O-Notation und negieren sie, gilt $g \notin O(f)$ genau dann, wenn

$$\neg[\exists c \in \mathbb{R}_{>0}\ \exists n_0 \in \mathbb{N}\ \forall n \in \mathbb{N}, n \geq n_0 : g(n) \leq c \cdot f(n)] =$$
$$\forall c \in \mathbb{R}_{>0}\ \forall n_0 \in \mathbb{N}\ \exists n \in \mathbb{N}, n \geq n_0 : g(n) > c \cdot f(n).$$

Also müssen wir zeigen, dass es *für alle* positiven Zahlen c und *für alle* natürlichen Zahlen n_0 *ein* $n \geq n_0$ gibt, so dass $g(n) > c \cdot f(n)$.

■ **Beispiel 4.7 —** $g \notin O(f)$**.**

- Für $g(n) = n^2$ und $f(n) = n$ gilt $g \notin O(f)$. Sei c eine beliebige positive Zahl und n_0 eine beliebige natürliche Zahl. Wähle $n = n_0 + c$. Dann gilt $g(n) = n^2 = (n_0+c)^2 > c \cdot (n_0+c) = c \cdot n = c \cdot f(n)$.
- Für $g(n) = 2n^2$ und $f(n) = n+7$ gilt $g \notin O(f)$. Sei c eine beliebige positive Zahl und n_0 eine beliebige natürliche Zahl. Wähle $n = n_0 + c + 7$. Dann gilt $g(n) = 2n^2 = 2(n_0+c+7)^2 > 2c(n_0+c+7) = 2cn = 2c \cdot f(n) > c \cdot f(n)$.
- Für $g(n) = n^3$ und $f(n) = 100000000000n^2$ gilt $g \notin O(f)$. Sei c eine beliebige positive Zahl und n_0 eine beliebige natürliche Zahl. Wähle $n = n_0 + 100000000000 \cdot c + 1$. Dann gilt $g(n) = n^3 = (n_0 + 100000000000 \cdot c + 1)^3 > 100000000000 \cdot c \cdot (n_0 + 100000000000 \cdot c + 1)^2 > 100000000000 \cdot c \cdot (n_0 + 100000000000 \cdot c)^2 >= 100000000000 \cdot c \cdot n^2 = c \cdot f(n)$.

◇

Nach unserer Intuition sollte ein Polynom vom Grad k immer kleiner als ein Polynom vom Grad $k+1$ sein. Mithilfe unserer Definition der Größe durch die O-Notation können wir dies nun auch formal beweisen.

Theorem 4.8 Sei $k \in \mathbb{N}$ und $g\colon \mathbb{N} \to \mathbb{N}$ ein Polynom mit $g(n) = \sum_{i=0}^{k} a_i n^i$ vom Grad k mit natürlichen Koeffizienten $a_0, \ldots, a_k \in \mathbb{N}$ und $a_k > 0$. Für jedes Polynom $f\colon \mathbb{N} \to \mathbb{N}$ vom Grad $k+1$ mit $f(n) = \sum_{i=0}^{k+1} b_i n^i$ und natürlichen Koeffizienten $b_0, \ldots, b_{k+1} \in \mathbb{N}$ mit $b_{k+1} > 0$ gilt $g \in O(f)$.

Ein sehr wichtiger Spezialfall tritt dann ein, wenn $g \in O(f)$ und $f \in O(g)$ gilt. In diesem Fall wachsen also f und g etwas gleich schnell.

Definition 4.9 — Θ-Notation. Sei $f\colon \mathbb{N} \to \mathbb{N}$ eine Funktion. Die Menge $\Theta(f)$ ist die Menge aller Funktionen $g\colon \mathbb{N} \to \mathbb{N}$, so dass $g \in O(f)$ und $f \in O(g)$ gilt, also

$$\Theta(f) = \{g \in O(f) \mid f \in O(g)\}.$$

Diese Definition ist symmetrisch: Gilt also $g \in \Theta(f)$, so gilt auch $f \in \Theta(g)$.

■ **Beispiel 4.10 —** $g \in \Theta(f)$**.**

- Für $g(n) = 2n$ und $f(n) = 3n+1$ gilt $g \in \Theta(f)$.
- Für $g(n) = 43n^3 + 27n^2 + (n/21) + 9$ und $f(n) = n^3$ gilt $g \in \Theta(f)$.

◇

Besonders praktisch an der O-Notation ist es, dass es egal ist, ob wir $f(n) = 43n^3 + 23n^2 + 7n + 1$ oder $f'(n) = n^3$ haben, denn $O(f) = O(f')$. Um also knapp anzugeben, dass ein Term $T(n)$ maximal quadratisch in n wächst, können wir einfach $T \in O(n^2)$ schreiben, wobei wir n^2 als Funktion verstehen, die n auf n^2 abbildet. In der Literatur wird manchmal auch "$g = O(f)$" statt "$g \in O(f)$" geschrieben.

Aufgaben

Aufgabe 4.1 — Vorrechnen. Beweisen oder widerlegen Sie die folgenden Aussagen über die Funktion $f\colon \mathbb{Q} \to \mathbb{Q}$ mit $f(x) = 2x$:
(i) Die Funktion f ist injektiv.
(ii) Die Funktion f ist surjektiv. ■

Aufgabe 4.2 — Vorrechnen. Sei $g\colon \mathbb{N} \to \mathbb{N}$ mit $g(n) = 3n + 2$ und $f\colon \mathbb{N} \to \mathbb{N}$ mit $f(n) = (1/2)n^2$. Zeigen Sie, dass $g \in O(f)$. ■

Aufgabe 4.3 Beweisen Sie, dass für alle $a, b > 0$ und alle $x \in \mathbb{R}$ mit $x > 0$ gilt, dass $\log_a(x) = \log_b(x)/\log_b(a)$.
Hinweis: Betrachten Sie den Ausdruck $\log_b(a^{\log_a(x)})$. ■

Aufgabe 4.4 Beweisen oder widerlegen Sie die folgenden Aussagen über die Funktion $f\colon \mathbb{N} \to \mathbb{N}$ mit $f(n) = n + 1$:
(i) Die Funktion f ist injektiv.
(ii) Die Funktion f ist surjektiv. ■

Aufgabe 4.5 Sei $g\colon \mathbb{N} \to \mathbb{N}$ mit $g(n) = n^4$ und $f\colon \mathbb{N} \to \mathbb{N}$ mit $f(n) = n^5$. Zeigen Sie, dass $g \in O(f)$. ■

Aufgabe 4.6 Beweisen Sie, dass für alle geraden natürlichen Zahlen $n \geq 1$ gilt, dass

$$(n/2)[\log(n) - 1] \leq \sum_{i=1}^{n} \log(i) \leq n \cdot \log(n).$$

■

Aufgabe 4.7 Beweisen Sie: Für alle natürlichen Zahlen $a, m, b \in \mathbb{N}$ mit $a \equiv b \pmod{m}$ gibt es eine Zahl $q \in \mathbb{Z}$, so dass $b = a + qm$. ■

Aufgabe 4.8 Sei $g\colon \mathbb{N}_{>0} \to \mathbb{N}_{>0}$ mit $g(n) = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ (das schreibt man auch als $n!$) und $f\colon \mathbb{N}_{>0} \to \mathbb{N}_{>0}$ mit $f(n) = n^n$. Beweisen Sie $g \in O(f)$. ■

Aufgabe 4.9 Sei $g\colon \mathbb{N} \to \mathbb{N}$ mit $g(n) = 3^n$ und $f\colon \mathbb{N} \to \mathbb{N}$ mit $f(n) = 2^n$. Beweisen Sie $g \notin O(f)$. ■

Aufgabe 4.10 Sei $a \in \mathbb{R}$ eine nicht-negative reelle Zahl und $b \in \mathbb{R}$. Zeigen Sie, dass die Funktion $f\colon \mathbb{R} \to \mathbb{R}$ mit $f(x) = ax + b$ bijektiv ist. ■

Aufgabe 4.11 Wenn zwei ganze Zahlen x, y den gleichen Betrag haben, also $|x| = |y|$ gilt, so schreiben wir $x \sim y$.

(i) Beweisen Sie, dass $\sim$ eine Äquivalenzrelation ist.

(ii) Bestimmen Sie die Äquivalenzklassen von $\sim$.

■

Aufgabe 4.12 Sei $f\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ eine Funktion mit $f(n,m) = n + (1/2)(n+m)(n+m+1)$. Zeigen Sie:

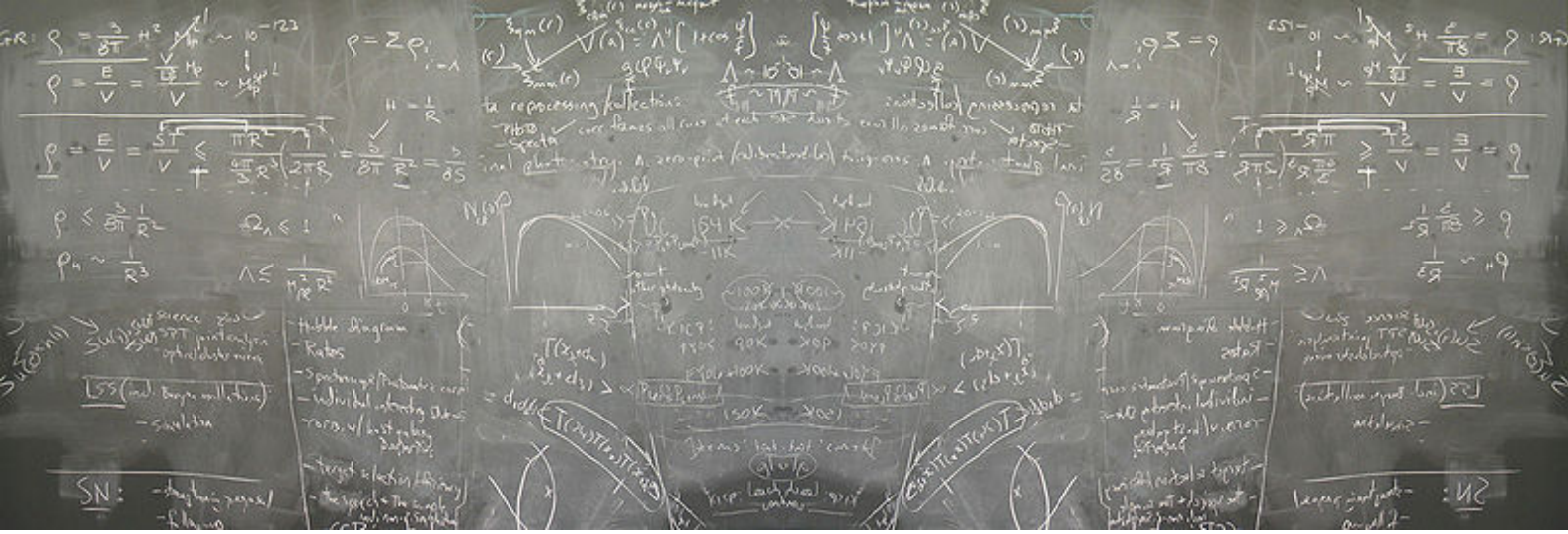
(i) Die Funktion f ist surjektiv.

(ii) Die Funktion f ist injektiv.

Hinweis: Was ist der kleinste Wert, den $f(n,m)$ annehmen kann, wenn $n+m = K$ gilt? Was ist der größte Wert? ■

Lernziele:

- Sicherer Umgang mit Funktionen und deren Eigenschaften
- Beherrschen der Rechenregeln für die wichtigsten Funktionen
- Grundlegendes Verständnis für Modulare Arithmetik, den mod-Operator und Teilbarkeitsregeln
- Äquivalenzklassen verstehen
- Verständnis der O-Notation
- Sicherer Umgang mit dem Vergleich zweier Funktionen durch die O-Notation



5. Kombinatorik, Laufzeiten

Zum Lesen:
Kapitel 6.1, 6.2, 6.5, 6.7, 6.8 in [Bel18]
Ausarbeitung von Sebastian zu Laufzeiten (also dieses Dokument)

Stichworte:

- Binomialkoeffizienten $\binom{n}{k}$; Interpretationen; Identitäten
- Permutationen; $n!$
- Binomischer Lehrsatz; Produkt von Binomen
- k-to-one correspondence
- Kombinatorische Beweise mittels Bijektionen
- Laufzeit von Algorithmen; Konkatenation; for-Schleife; while-Schleife

5.1 Laufzeiten

Wie bereits zu Anfang des letzten Kapitels erklärt, möchten wir jedem Algorithmus A eine *Worst-Case-Laufzeit* T_A zuordnen. Anders ausgedrückt: Wenn A eine Eingabe der Länge n erhält, rechnet er für höchstens $T_A(n)$ Schritten. Nehmen wir uns beispielhaft den folgenden Algorithmus, der das maximale Element in einem Array X der Länge n sucht:

```
FindMax(X)
1 :  n := X.length();
2 :  max := X[0];
3 :  for i = 1,...,n-1 :
4 :      if X[i] > max :
5 :          max := X[i];
6 :  if max < 0 :
7 :      max := 0
8 :  return max
```

Nun ist es ersteinmal sehr schwer, eine konkrete Aussage über die wirkliche Laufzeit T_{FindMax} zu machen: Wir wissen nicht, wieviele Operationen der Rechner wirklich für die Zeile » **if** $X[i] >$

max «benötigt. Es könnte sogar so sein, dass diese Anzahl vom konkreten Rechner abhängt, denn verschiedene Prozessoren können verschiedene Operationen verschieden schnell ausführen. Selbst wenn wir die Zahl auf unserem Rechner genau wüssten, hängt diese Zahl natürlich auch vom Compiler ab, der die Zeile in Maschinencode verwandelt. Somit ist es sehr schwer, die genaue Laufzeit zu bestimmen. Wenn wir jedoch realistisch annehmen, dass jede der Zeilen in konstanter Laufzeit auszuführen ist, können wir eine sinnvolle obere Schranke geben. Sei c die maximale Ausführungszeit einer einzelnen Zeile im obigen Code. Offenbar werden die Zeilen 1, 2, 6, 7 und 8 genau einmal ausgeführt. Die Zeilen 3 und 4 werden jeweils $n-1$-mal ausgeführt, wenn das Eingabearray die Länge n hat. Wie häufig Zeile 5 genau ausgeführt wird, ist nicht klar und hängt von der Eingabe ab: Beim Array $X = [1,2,3,4,\ldots,n]$ wird die Zeile auch $n-1$-mal ausgeführt, beim Array $X = [n,n-1,\ldots,1]$ jedoch niemals. Aber man sieht leicht, dass im schlimmsten Fall nur $n-1$ Wiederholungen auftreten können. Insgesamt können wir also abschätzen, dass $T_{\mathsf{FindMax}}(n) \leq 3c(n-1)+5c$ ist. Da wir jedoch im Wesentlichen nur am groben Wachstum der Funktion interessiert sind, können wir das in eine einfachere Form bringen. Da $3c(n-1)+3c \in O(n)$, haben wir also $T_{\mathsf{FindMax}} \in O(n)$. Die Laufzeit ist also linear. Wenn wir also die Eingabelänge verdoppeln, erwarten wir, dass wir ungefähr doppelt so lange rechnen müssen. Hätten wir zum Beispiel das folgende Verfahren $\mathsf{FindMax}_2$, welches ebenso das Minimum eines Arrays X berechnet, so sehen wir, dass die Zeile 6 hier $(n-1)\cdot(n-1)$-mal ausgeführt wird. Also gilt $T_{\mathsf{FindMax}_2} \in O(n^2)$ und $T_{\mathsf{FindMax}_2} \notin O(n)$.

$\mathsf{FindMax}_2(X)$

```
1 :  n := X.length();
2 :  max := X[0];
3 :  for i = 1,...,n-1 :
4 :      maxi := X[i];
5 :      for j = 1,...,n-1 :
6 :          if X[j] > X[i] :
7 :              maxi = X[j];
8 :      if maxi > max :
9 :          max := maxi
10 :  return max
```

Wenn wir also die Länge X des Arrays verdoppeln, erwarten wir hier, dass wir ungefähr viermal so lange rechnen müssen. Zusammengefasst wollen wir also die Worst-Case-Laufzeit eines Algorithmus mithilfe der O-Notation nach oben abschätzen. Es gibt drei wichtige Grundoperationen, wie man Algorithmen miteinander verbinden kann, die wir uns im Folgenden genauer anschauen wollen.

Konkatenation

Sind A und B Algorithmen, so erhalten wir den Algorithmus $\mathsf{A} \circ \mathsf{B}$, indem wir zuerst A und dann B ausführen. Betrachten wir das folgende Beispiel:

A

$x := 42$

$y := x^2$

B

$z := 2 \cdot y$

Dementsprechend erhalten wir den folgenden Algorithmus $\mathsf{A} \circ \mathsf{B}$ als *Konkatenation* oder *Hintereinanderausführung* der beiden Algorithmen.

```
A ∘ B
x := 42
y := x^2
z := 2 · y
```

Wie man sehr leicht einsieht, gilt $T_{\mathsf{A}\circ\mathsf{B}}(n) = T_{\mathsf{A}}(n) + T_{\mathsf{B}}(n)$, also brauchen wir nur die beiden Laufzeiten zu addieren, um die Gesamtlaufzeit zu erhalten.

for-Schleife

Ist A ein Algorithmus, der eine natürliche Zahl i als Eingabe erhält, so können wir mittels einer for-Schleife den Algorithmus wiederholt ausführen:

```
B(n)
for i = 1,...,n :
    A(j)
```

Auch hier kann man sich leicht ableiten, dass für die Laufzeit $T_{\mathsf{B}}(n) \in O(\sum_{i=1}^{n} T_{\mathsf{A}}(i))$ gilt. Wir müssen uns also nur überlegen, wie lange die Summe der Ausführungszeiten von A ist.

Ein relativ häufig auftretender Fall ist, dass man k ineinander geschachtelte Schleifen hat, die jeweils ungefähr n Schritte durchlaufen. Die Gesamtlaufzeit ist in diesem Fall einfach $O(n^k)$ (siehe auch unser obiges Beispiel mit $\mathsf{FindMax}_2$ mit $k = 2$). Die Analyse wird deutlich komplizierter, wenn die Schleifen voneinander abhängen: Läuft zum Beispiel der Schleifenzähler i der äußeren Schleife von 1 bis n und der Schleifenzähler j der inneren Schleife von i bis n, ist die Laufzeit weiterhin $O(n^2)$, aber die Analyse wird deutlich aufwändiger. Es gibt natürlich auch Schleifen, die nur konstant häufig durchlaufen. Diese vergrößern unsere Laufzeit entsprechend nicht.

while-Schleife

Ist A ein Algorithmus, der eine natürliche Zahl i als Eingabe erhält und update ein Algorithmus, der ebenfalls i als Eingabe erhält und eine natürliche Zahl zurückgibt, so können wir mittels einer while-Schleife den Algorithmus auch wiederholt ausführen:

```
B(n)
i := n
while i > 1 :
    A(i)
    i := update(i)
```

Die Laufzeit T_{B} hier zu bestimmen, ist schon schwieriger und hängt enorm vom Verhalten von $\mathsf{update}(i)$ ab. Die einfachsten und häufigsten Fälle sind zum Beispiel, dass $\mathsf{update} = i+1$ oder $\mathsf{update} = i/2$ gilt. Im ersteren Fall haben wir nur eine for-Schleife anders geschrieben. Im zweiten Fall (und allgemein) können wir wie folgt argumentieren: Wir bekommen ja nun Werte $i_1 = n$, $i_2 = \mathsf{update}(i_i) = \mathsf{update}(n)$, $i_3 = \mathsf{update}(i_2) = \mathsf{update}(\mathsf{update}(n))$, und so weiter. Allgemein gilt $i_k = \mathsf{update}^k(n)$, wobei $\mathsf{update}^k(n) = \underbrace{\mathsf{update}(\mathsf{update}(\ldots(\mathsf{update}}_{k\text{-mal}}(n)$ ist. Nun möchten wir das kleinste k bestimmen, so dass $\mathsf{update}^k(n) \leq 1$ gilt. Dementsprechend häufig wird nämlich unsere Schleife ausgeführt. Im Falle $\mathsf{update}(i) = i/2$ kennen wir diesen Wert übrigens schon: Es ist $\lceil \log_2(n) \rceil - 1$. So häufig müssen wir nämlich n durch 2 teilen, bis wir bei der 1 angekommen sind. Im Allgemeinen kann es aber durchaus schwerer sein, die Laufzeit zu analysieren. Haben wir jedoch solch ein k bestimmt, so können wir einfach $T_{\mathsf{B}}(n) \leq k \cdot T_{\mathsf{A}}(n)$ benutzen, um die Laufzeit von B abzuschätzen. Wir werden uns in »Algorithmen und Datenstrukturen «noch sehr intensiv mit der Abschätzung komplizierterer update-Algorithmen auseinandersetzen.

Aufgaben

Aufgabe 5.1 — Vorrechnen. Beweisen Sie: Für alle $n,k \in \mathbb{N}$ gilt

$$\sum_{m=0}^{n} \binom{m}{k} = \binom{n+1}{k+1}.$$

■

Aufgabe 5.2 — Vorrechnen. Geben Sie eine möglichst gute obere Schranke für die Laufzeit des folgenden Algorithmus A, der eine natürliche Zahl n als Eingabe erhält:

```
A(n)
1 : s := 0;
2 : for i = 1,...,n :
3 :     s := s + i;
4 : return s
```

■

Aufgabe 5.3 Beweisen Sie: Für alle $n,k \in \mathbb{N}$ gilt

$$\binom{n}{k} \cdot k = \binom{n-1}{k-1} \cdot n.$$

■

Aufgabe 5.4 Geben Sie eine möglichst gute obere Schranke für die Laufzeit des folgenden Algorithmus A, der eine natürliche Zahl n als Eingabe erhält:

```
A(n)
1 : s := 0;
2 : for i = 1,...,n :
3 :     for j = 1,...,n :
4 :         s := s + i · j;
5 : return s
```

■

Aufgabe 5.5 Beweisen Sie: Für alle $n,k,m \in \mathbb{N}$ gilt:

$$\binom{n}{m} \cdot \binom{m}{k} = \binom{n}{k} \cdot \binom{n-k}{m-k}.$$

■

Aufgabe 5.6 Beweisen Sie: Für alle $n,k \in \mathbb{N}$ gilt:

$$\sum_{k=0}^{n} \binom{n}{k} \cdot (-1)^k = 0.$$

Hinweis: Benutzen Sie den binomischen Lehrsatz. ■

Aufgabe 5.7 Geben Sie eine möglichst gute obere Schranke für die Laufzeit des folgenden Algorithmus A, der eine natürliche Zahl n als Eingabe erhält:

```
A(n)
1 : i := n;
2 : s := 0;
3 : while i > 0 :
4 :     s := s + 1;
5 :     i := ⌊i/2⌋;
6 : return s
```

■

Aufgabe 5.8 Beweisen Sie, dass für alle $n \in \mathbb{N}$ gilt:

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}.$$

■

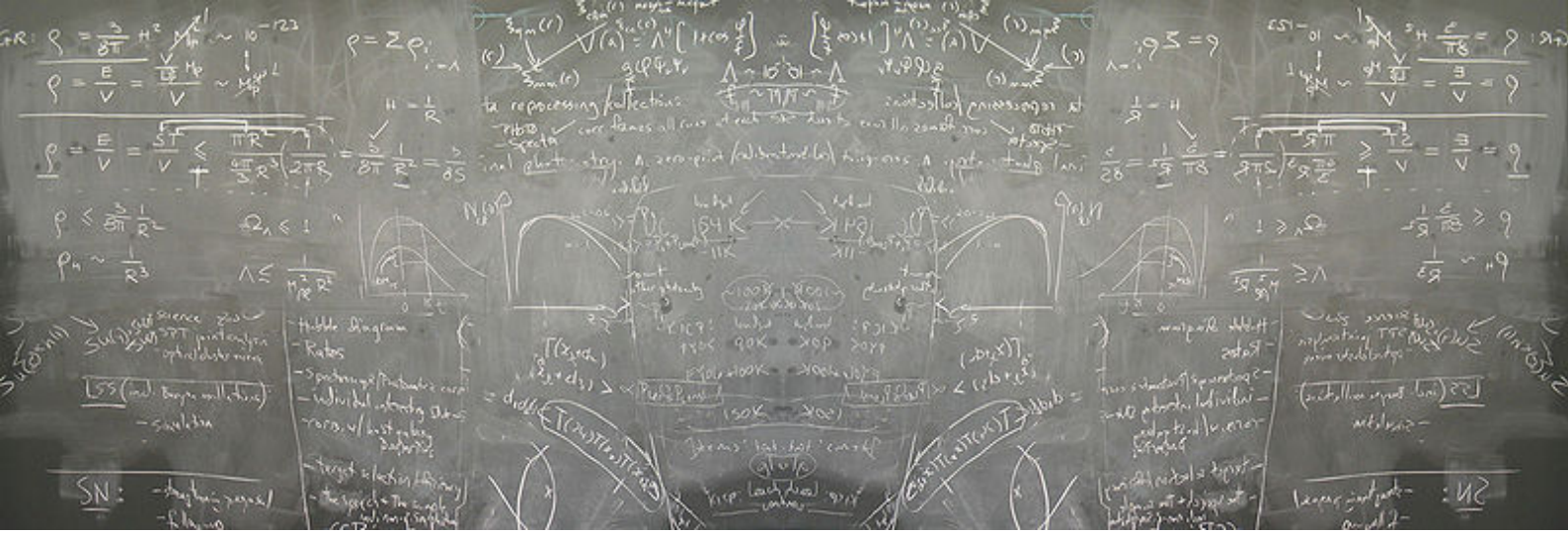
Aufgabe 5.9 Beweisen Sie, dass für alle $n \in \mathbb{N}$ gilt:

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

Hierbei ist F_n die $n$te Fibonacci-Zahl. ■

Lernziele:

- Die Begriffe Binomialkoeffizient und Fakultät kennen
- Grundlegende Identitäten des Binomialkoeffizient kennen
- Einfache kombinatorische Beweise mittels Abzählargumenten entwickeln können
- Den Binomischen Lehrsatz anwenden können
- Die Begriffe Teilmenge und geordnetes Tupel unterscheiden können
- Erste Laufzeitabschätzungen tätigen können



Bibliographie

[Bel18] Sarah-Marie Belcastro. *Discrete mathematics with ducks*. CRC Press, 2018. ISBN: 9781466504998.

[Wol17] Karsten Wolf. *Präzises Denken für Informatiker*. Springer-Verlag, 2017. ISBN: 3662549727.