Mathematik IT-1

Diskrete Mathematik für Informatik-Studiengänge

29. November 2021

Prof. G. Averkov

Institut für Mathematik, Fakultät 1

Fachgebiet Algorithmische Mathematik

Brandenburgische Technische Universität Cottbus-Senftenberg

Inhaltsverzeichnis

Kapitel III

Algorithmische und Programmiergrundlagen

1 Berechnungen in der Theorie und Praxis

1.1. Viel mehr zum Thema dieses Kapitels findet man im Kurs Entwicklung von Softwaresystemen, vgl. auch das umfangreiche Buch [GS11].

1.2. Wer mit den Theorie und der Praxis der Computer-Berechnungen zu tun hat, muss sich in der Regel auch mit den folgenden Konzepten und Aspekten auseinandersetzen:

Algorithmische Theorie	Reale Berechnungen
Rechenmodell	Hardware Betriebsysteme
Rechenmodell	reale Programmiersprache
Algorithmen und deren Analyse	Umsetzung von Algorithmen
Klassifikation von Rechenproblemen, etwa nach dem Schwierigkeitsgrad	Analyse algorithmischer Fragestellungen aus der Praxis

Im Prinzip ist es möglich – aber nicht unbedingt sinnvoll – die theoretischen Aspekte komplett getrennt von der Praxis zu behandeln, denn Theorie ist für die Praxis entwickelt. Um die Theorie sinnvoll einsetzen zu können, braucht man auch praktisches Wissen.

2 Kodierung mit Strings

2.1 Def. Wir sagen, dass wir die Elemente der Menge X mit Hilfe der Elemente der Menge Y **kodieren**, wenn wir eine injektive Abbildung $c : X \to Y$ festlegen. Wir nennen dann c eine **Kodierung** von X mit Y und $y = c(x)$ die **Kodierung** von x als y bzgl. c.

2.2 Def. Für eine nichtleere Menge A und ein $n \in \mathbb{N}_0$ nennt man die Elemente von

$$A^n = \underbrace{A \times \ldots \times A}_{n \text{ mal}}$$

Strings der **Länge** n über dem **Alphabet** A. Es gibt nur einen String der Länge 0, diesen bezeichnen wir als $\oslash$ (diese Bezeichnung ist ähnlich zur Bezeichnung für die leere Menge steht aber für etwas anderes). Das heißt, $A^0 = \{\oslash\}$. Oft schreiben wir einen String $(x_1, \ldots, x_n) \in A^n$ auch kürzer als $x_1 \cdots x_n$ und wir nennen x_i das i-te **Zeichen** von x. Die Menge aller Strings über dem Alphabet A bezeichnen wir mit A^*, das heißt:

$$A^* := \bigcup_{n \in \mathbb{N}_0} A^n.$$

Der Wert n für einen String $x = x_1 \cdots x_n \in A^*$ heißt die **Länge** von x und wird als $|x|$ bezeichnet.

2.3 Def. Für zwei Strings $x = x_1 \cdots x_n$ und $y = y_1 \cdots y_k$ aus A^* heißt der String $xy = x_1 \cdots x_n y_1 \cdots y_k = (x_1, \ldots, x_n, y_1, \ldots, y_k) \in S^*$ die **Verkettung** oder **Konkatenation** von x und y.

2.4 Def. Strings über dem Alphabet $\{0, 1\}$ nennt man **Bit-Strings** und die Zeichen von Bit-Strings nennt man **Bits**.

2.5. Auf digitalen Speichermedien und in digitalen Rechengeräten werden Daten mit Hilfe von Bit-Strings kodiert. Das bedeutet, dass man für verschiedene Mengen X injektive Abbildungen $c : X \to \{0,1\}^*$ festlegt. Die Länge des Bit-Strings $c(x)$ für $x \in X$ ist somit die Länge der Darstellung von x in der Kodierung c.

2.6 Def. Für eine endliche Menge A und $k \in \mathbb{N}_0$, nennen wir eine injektive Abbildung $c : A \to \{0,1\}^k$ eine k-**Bit-Kodierung** von A.

2.7 Prop. Sei A endliche Menge und seien $k \in \mathbb{N}_0$. Dann sind die folgenden Bedingungen äquivalent:

(i) A besitzt eine k-Bit-Kodierung.

(ii) $|A| \leq 2^k$.

Beweis. Für die Existenz einer injektiven Abbildung von A nach $\{0,1\}^k$ ist es notwendig und hinreichend, dass X höchstens so viele Elemente wie $\{0,1\}^k$ hat. Nach Korollar **??** hat $\{0,1\}^k$ genau $|\{0,1\}|^k = 2^k$ Elemente. Das zeigt die gewünschte Äquivalenz. □

2.8 Bsp. Da die Menge

$$A = \{\text{'a'}, \ldots, \text{'z'}\}$$

26 Elemente hat und es $2^4 < 25 \leq 2^5$ gilt, besitzt A eine 5-Bit- aber keine 4-Bit-Kodierung.

2.9 Prop (Kodierung von Strings durch Bit-Strings)**.** Sei A endliche nichtleere Menge. Dann existiert eine Injektive Abbildung von A^* nach $\{0,1\}^*$. Mit anderen Worten: Strings über einem beliebigen endlichen Alphabet können als Bit-Strings kodiert werden.

Beweis. Sei $k \in \mathbb{N}_0$ Wert mit $2^k \geq |A|$. Dann existiert eine Kodierung $c : A \to \{0,1\}^k$. Diese Kodierung erzeugt die Kodierung von A^* nach $\{0,1\}^*$ mit

$$(x_1, \cdots, x_n) \in A^n \mapsto (c(x_1), \ldots, c(x_n)) \in \{0,1\}^{nk}.$$

□

2.10 Prop (Kodierung von Paaren)**.** Besitzen Mengen X und Y Kodierungen mit Bit-Strings, so besitzt auch $X \times Y$ Kodierung mit Bit-Strings.

Beweis. Seien $a : X \to \{0,1\}^*$ und $b : Y \to \{0,1\}^*$ Kodierungen von X bzw. Y mit Bit-Strings. Dann können wir ein Trennzeichen $\# \notin \{0,1\}$ einführen und das Paar (x,y) mit $x \in X, y \in Y$ als ein String $a(x)\,\#\,b(y) \in \{0,1,\#\}^*$ über dem drei-elementigen Alphabet $\{0,1,\#\}$ kodieren. Nach Proposition 2.9 können Strings über einem beliebigen endlichen Alphabet als Bit-Strings kodiert werden, es gibt also eine Kodierung $c : \{0,1,\#\}^* \to \{0,1\}^*$.

Das ergibt die Kodierung

$$(x,y) \mapsto c(a(x)\#b(y))$$

von $X \times Y$ nach $\{0,1\}^*$. □

2.11 Kor (Kodierung von Tupeln)**.** Besitzen Mengen $X_1, \ldots, X_n$ Kodierungen mit Bit-Strings, so besitzt auch das Produkt $X_1 \times \cdots X_n$ eine Kodierung mit Bit-Strings.

Beweis. Die Konstruktion im Beweis von (2.10) direkt vom Fall $n = 2$ auf ein allgemeines n erweitern. Alternativ kann man die Behauptung durch die Induktion nach n, mit dem Induktionsanfang $n = 2$, aus Korollar 2.10 herleiten. □

2.12 Bsp. Eine mögliche Kodierung von Folgen/Tupeln nichtnegativer ganzer Zahlen als Bit-Strings kann durch die folgenden Schritte erfolgen:

$$\begin{aligned}(0,6,3,7) &\mapsto (0,110,11,111) && \text{(Binärdarstellung)}\\ &\mapsto 0\#110\#11\#111 && \text{(Trennzeichen)}\\ &\mapsto 00\,\mathbf{11}\,01\,01\,00\,\mathbf{11}\,01\,01\,\mathbf{11}\,010101.\end{aligned}$$

Im letzten Schritt nutzt man die Kodierung

$$0 \mapsto 00, \qquad 1 \mapsto 01, \qquad \# \mapsto 11$$

von $\{0,1,\#\}$ mit 2-Bit-Strings.

3 Stellenwertsysteme: Darstellung von Zahlen

3.1 Thm (Darstellung natürlicher Zahlen in einem Stellenwertsystem)**.** *Sei* $b \in \mathbb{Z}_{\geq 2}$*. Dann besitzt jedes* $z \in \mathbb{N}$ *eine eindeutige Darstellung als*

$$z = \sum_{i=0}^{k} z_i b^i \tag{III.1}$$

mit $k \in \mathbb{N}_0, z_0, \ldots, z_k \in \{0, \ldots, b-1\}$ *und* $z_k \neq 0$*. Mit anderen Worten ist die Relation* $z \mapsto (z_k, \ldots, z_0)$ *für* z *und* $z_0, \ldots, z_k$ *wie oben eine injektive Abbildung von* $\mathbb{N}$ *nach* $\{0, \ldots, b-1\}^*$*.*

Beweis. Die Existenz und Eindeutigkeit der gewünschten Darstellung für z zeigen wir durch Induktion über z. Bei $1 \leq z \leq b-1$ ist $k = 1$ und $z_0 = z$. Sei $z > b$ und sei die Existenz und Eindeutigkeit einer solchen Darstellung für die Zahlen $1, \ldots, z-1$ bereits gezeigt. Bei z ist z_0 notwendigerweise der Rest der Division von z durch b, weil die Terme $z_i b^i$ mit $i > 0$ alle durch b teilbar sind. Die Zahl $z - z_0$ ist positiv und durch b teilbar. Somit ist $(z - z_0)/b$ eine

natürliche Zahl, die echt kleiner als z ist; denn $(z - z_0)/b \leq z/b < z$. Die Darstellung von z, nach der wir suchen, kann als

$$(z - z_0)/b = \sum_{i=0}^{k-1} z_{i+1} b^i$$

umformuliert werden. Die Existenz und Eindeutigkeit von $z_1, \ldots, z_k \in \{0, \ldots, b-1\}$ mit $z_k \neq 0$ folgt durch die Anwendung der Induktionsvoraussetzung zu $(z - z_0)/b$. □

3.2 Def. Im Kontext von Theorem 3.1 heißt das Tupel $(z_k, \ldots, z_0)$ die **Darstellung** von $z \in \mathbb{N}$ im **Stellenwertsystem zur Basis** b. Die Zahl z mit dieser Darstellung wird als

$$z_k \cdots z_0 \; _b$$

bezeichnet. Bzgl. des Stellenwertsystems zur Basis b wird z eine $(k+1)$-**stellige Zahl genannt**. Hierbei nennt man z_k die **höchste Stelle** und z_0 die **niedrigste Stelle**. Stellenwertsysteme zu Basen $2, 10$ und 16 nennt man **binär**, **dezimal** bzw. **hexadezimal**.

Die Darstellung der natürlichen Zahlen wird erweitert, indem man die Darstellung von 0 als 0 festgelegt und die Darstellung von $\pm z$ als $(\pm, z_k, \ldots, z_0) \in \{+, -\} \times \{0, \ldots, b-1\}^*$ mit Hilfe der Darstellung $(z_k, \ldots, z_0)$ von $z \in \mathbb{N}$ definiert.

3.3 Prop. Sei $b \in \mathbb{Z}_{\geq 2}$ und $t \in \mathbb{N}_0$. Dann ist die Abbildung

$$(z_t, \ldots, z_0) \mapsto z := \sum_{i=0}^{t} z_i b^i \tag{III.2}$$

eine Bijektion $\{0, \ldots, b-1\}^t \to \{0, \ldots, b^{t+1}-1\}$. Insbesondere ist die Umkehrung dieser Abbildung eine Kodierung von $\{0, \ldots, b^{t+1}-1\}$ mit t-Bit-Strings.

Beweis. Sei $(z_t, \ldots, z_0) \in \{0, \ldots, b-1\}^t$. Dann gilt $0 \leq z_i \leq b-1$ für jedes $i \in \{0, \ldots, b-1\}$, sodass die Abschätzungen

$$0 \leq z \leq \sum_{i=0}^{t} (b-1) b^i = (b-1) \sum_{i=0}^{t} b^i = b^{t+1} - 1$$

erfüllt sind. Das zeigt, dass die Angabe des Wertebereichs $\{0, \ldots, b^{t+1}-1\}$ korrekt ist. Der Definitionsbereich sowie der Wertebereich haben je b^t Elemente. Um zu zeigen, dass die Abbildung

bijektiv ist, reicht es zu zeigen, dass die Abbildung injektiv ist. Seien $(z_t, \ldots, z_0), (z'_t, \ldots, z'_0) \in \{0, \ldots, b-1\}^t$ verschieden. Dann gibt es das größte $k \in \{0, \ldots, t\}$ mit $z_k \neq z'_k$. Sei ohne Beschränkung der Allgemeinheit $z_k > z'_k$. Dann ist

$$\begin{aligned}
\sum_{i=0}^{t} z_i b^i - \sum_{i=0}^{t} z'_i b^i &= (z_k - z'_k) b^k + \sum_{i=0}^{k-1} (z_i - z'_i) b^i \\
&\geq b^k + \sum_{i=0}^{k-1} (0 - (b-1)) b^i \\
&= b^k - (b-1) \sum_{i=0}^{k-1} b^i \\
&= b^k - (b^k - 1) \\
&= 1.
\end{aligned}$$

Es folgt: $\sum_{i=0}^{t} z_i b^i > \sum_{i=0}^{t} z'_i b^i$. Also ist die Abbildung tatsächlich injektiv. □

3.4. Die Intuition hinter den Stellenwertsystemen ist wie folgt: man führt die Ziffernbezeichnungen für die b Ziffern des Stellenwertsystems ein und rechnet dann im Rahmen dieses Stellenwertsystems schriftlich, so wie es für das System mit der Basis $b = 10$ in der Schule gelernt haben. Hier ein Vergleich von $b = 10$ und $b = 2$:

Bezeichnungen bzgl. der Basis 10	Bezeichnungen bzgl. Basis 2
0 ist die Null	0 ist die Null
1 ist die Eins	1 ist die Eins
$2 := 1 + 1$	$10 := 1 + 1$
$3 := 2 + 1$	
$4 := 3 + 1$	
$5 := 4 + 1$	
$6 := 5 + 1$	
$7 := 6 + 1$	
$8 := 7 + 1$	
$9 := 8 + 1$	
$10 := 9 + 1$	

Stellen Sie sich einfach vor, Sie würden die Ziffern des Dezimalsystems nicht kennen. Denn

würde die linke Seite der Tabelle Ihnen Erklären, was für eine Bedeutung die Ziffern haben $1, \ldots, 9$ sowie die Zahl 10 haben. Danach könnte man Ihnen auch erklären, dass $100 = 10 \cdot 10$, $1000 = 10 \cdot 10 \cdot 10$ ist usw. und dass z.B. $1425 = 1000 + 4 \cdot 100 + 2 \cdot 10 + 5$ ist. Nach diesen Erklärungen sind die Darstellungen aller nichnegativen ganzen Zahlen festgelegt.

Wenn Sie die Ziffern des Dezimalsystems nicht kennen würden, könnte man Sie genauso in das Binärsystem einführen, indem man Ihnen erklären würde, dass $10 := 1 + 1$ ist, dass $100 = 10 \cdot 10$, $1000 = 10 \cdot 10 \cdot 10$ ist usw. und dass z.B. $101011 = 100000 + 1000 + 10 + 1$ ist.

3.5 Bsp. Schriftliche Addition, Subtraktion, Multiplikation und Division zu einer beliebigen Basis b geht analog zur gewohnten Basis $b = 10$. Etwa

Addition zur Basis 2:

$$\begin{array}{rcccc} & 1 & 0 & 1 \\ + & & 1 & 1 \\ \hline 1 & 0 & 0 & 0 \end{array}$$

3.6. Die Konvertierung einer Darstellung zu einer Basis b in die Darstellung zur Basis 10 geht direkt mit der Verwendung von (III.1), aber eine effizientere Konvertierung erhält man, wenn man das so genannte **Horner-Schema** benutzt. Für $k = 2$, benutzt erhält man durch das Ausklammern den Ausdruck

$$z_2b^2 + z_1b + z_0 = b(bz_2 + z_1) + z_0,$$

mit nur zwei Multiplikationen auf der rechten Seite; für $k = 3$, den Ausdruck

$$z_3b^3 + z_2b^2 + z_1b + z_0 = b(b(bz_3 + z_2) + z_1) + z_0$$

mit drei Multiplikationen und so fort für $k \geq 4$. Das Auswerten erfolgt dann von innen nach

außen durch das Auswerten der Werte für die Klammern, etwa für $k = 3$:

$$\underbrace{\underbrace{b\,(b\,\underbrace{(bz_3 + z_2)}_{\text{1. Runde}} + z_1)}_{\text{2. Runde}} + z_0}_{\text{3. Runde}},$$

3.7 Bsp. Die Konvertierung von der Basis 10 zu einer anderen Basis erfolgt durch iterative Division mit Rest mit der Verwendung der konstruktiven Idee des Beweises von Theorem 3.1. Konvertieren wir zum Beispiel die Zahl 46 in das System zur Basis 3. Es gilt

$$\begin{aligned} 46 &= 15 \cdot 3 + 1 = (5 \cdot 3 + 0) \cdot 3 + 1 = 5 \cdot 3^2 + 0 \cdot 3 + 1 \\ &= (1 \cdot 3 + 2) \cdot 3^2 + 0 \cdot 3 + 1 \\ &= 1 \cdot 3^3 + 2 \cdot 3^2 + 0 \cdot 3^1 + 1 \cdot 3^0. \end{aligned}$$

Das heißt:

$$46_{\,10} = 1201_{\,3}.$$

Das Horner-Schema zeigt sich hier in umgekehrter Form!

3.8 Bsp. Je geringer die Basis b des Stellenwertsystems ist, desto mehr Stellen braucht man um eine Zahl $z \in \mathbb{N}$ im Stellenwertsystem zur Basis b zu beschreiben. In digitalen Geräten benutzt man aber die kleinstmögliche Basis $b = 2$. Daher nutzt man im Zusammenhang mit dem binären System auch das Hexadezimalsystem, dessen Basis 16 eine Potenz von 2 ist. Aufgrund dieser Tatsache gibt es einen direkten einfachen Zusammenhang zwischen den Darstellungen zur Basis 2 und zur Basis 16.

Diesen Zusammenhang illustrieren wir an einem Beispiel. Zunächst sei bemerkt dass die 16 Ziffern des Hexadezimalsystems als $0, \ldots, 9, A, B, C, D, E, F$ bezeichnet werden. Das bedeutet es gibt den folgenden Zusammenhang zwischen den Ziffern des Hexadezimalsystem und deren Darstellungen als Zahlen im Binär- und Dezimalsystem:

Hexadezimal	Dezimal	Binär
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Hier ist beispielsweise A zur Basis 16 gleich der 10 zu unserer Standardbasis 10, und F zur Basis 16 ist gleich der 15 zur Basis 10. Die Zahl BEE im Hexadezimal-System kann ins Binärsystem

konvertiert werden indem man jede Ziffer durch ihre Binärdarstellung ersetzt.

$$\mathrm{BEE}_{16} = 1011\ 1110\ 1110_2.$$

Um sich zu vergewissern, dass das tatsächlich stimmt, kann man sich überlegen, was diese Gleichheit im Dezimalsystem bedeutet:

$$\begin{aligned}
&\underbrace{(1\cdot 2^3+0\cdot 2^2+1\cdot 2^1+1\cdot 2^0)}_{\mathrm{B}}16^2+\underbrace{(1\cdot 2^3+1\cdot 2^2+1\cdot 2^1+0\cdot 2^0)}_{\mathrm{E}}16^1+\underbrace{(1\cdot 2^3+1\cdot 2^2+1\cdot 2^1+0\cdot 2^0)}_{\mathrm{E}}16^0\\
=&\underbrace{1\cdot 2^{11}+0\cdot 2^{10}+1\cdot 2^9+1\cdot 2^8}+\underbrace{1\cdot 2^7+1\cdot 2^6+1\cdot 2^5+0\cdot 2^4}+\underbrace{1\cdot 2^3+1\cdot 2^2+1\cdot 2^1+0\cdot 2^0}.
\end{aligned}$$

3.9. Um zu sehen, wie man im Computer Daten mit den Symbolen 0 und 1 darstellt, kann unter Linux oder Mac der hexdump-Befehl benutzt werden. Zum Befehl

```
echo "abcdefghijklmnopqrstuvwxyz" | hexdump -C
```

erhält man die Ausgabe

```
00000000  61 62 63 64 65 66 67 68  69 6a 6b 6c 6d 6e 6f 70  |abcdefghijklmnop|
00000010  71 72 73 74 75 76 77 78  79 7a 0a                 |qrstuvwxyz.|
0000001b
```

Hier ist 61 die hexadezimale Kodierung des Buchstaben a, 62 die hexadezimale Unicode-Kodierung des Buchstaben b und $0a$ die hexadezimale Unicode-Kodierung von ‘neue Zeile’.

3.10 Aufg. Bei rationalen Zahlen erfolgt die Berechnung der Darstellung als Nachkommazahl zur Basis b genauso wie im Dezimalsystem. Berechnen Sie die Darstellung von $1/2$, $1/3$, $1/4$, $1/5$ und $1/6$ als Nachkommazahl im Binärsystem.

3.11 (Restklassenringe)**.** Sei $m \in \mathbb{N}$. Zahlen $x, y \in \mathbb{Z}$ nennt man kongruent modulo m, wenn $x - y$ durch m teilbar sind (das bedeutet, die Zahlen ergeben den selben Rest $r \in \{0, \ldots, m-1\}$ bei der Division durch m). Die Kongruenz Modulo m ist eine Äquivalenzrelation und wir bezeichnen als $[x]$ die Äquivalenzklasse von x bzgl. dieser Relation, das ist also die Menge aller ganzen Zahlen, die den selben Rest als x bei der Division durch m ergeben. Es gibt genau m solche Äquivalenzklassen, das sind die Klassen $[0], \ldots, [m-1]$. Die Menge $\{[0], \ldots, [m-1]\}$ aller Restklassen wird als $\mathbb{Z}/m\mathbb{Z}$ bezeichnet. Es stellt sich heraus, dass durch man in $\mathbb{Z}/m$ die Verknüpfungen $+$ und $\cdot$ durch

$$[x] + [y] := [x + y],$$
$$[x] \cdot [y] := [x \cdot y]$$

einführen kann. Mit diesen Verknüpfungen ist $\mathbb{Z}/m$ ein kommutativen Ring mit der Null $[0]$ und der Eins $[1]$.

3.12 (Ganzzahlige Arithmetik mit Zahlen fester Bit-Größe)**.** Bei der Umsetzung der ganzzahligen Arithmetik für Register und Speicherzellen fester Bit-Größe k ist die Register-Arithmetik eigentlich die Arithmetik des Rings $\mathbb{Z}/m\mathbb{Z}$ mit $m = 2^k$. Werden durch k Bits nicht-negative Zahlen im Bereich $\{0, \ldots, 2^k - 1\}$ dargestellt, so ergibt die Addition der Eins zu $2^k - 1$ den Wert 0. Das Entspricht der Gleichung $[m-1] + [1] = [0]$ im Ring $\mathbb{Z}/m\mathbb{Z}$. Mit anderen Worten sind die Zahlen $0, \ldots, 2^k - 1$ Vertreter der jeweiligen Restklassen modulo m.

Ähnlich ist die Situation mit der Darstellung von ganzen Zahlen mit dem Vorzeichen. In diesem Fall werden ganze Zahlen im Bereich $\{-2^{k-1}, \ldots, 2^{k-1} - 1\}$ durch k Bits kodiert. Die Addition der Eins zu $2^{k-1} - 1$ ergibt in dieser Darstellung -2^{k-1}. Das entspricht der Gleichung $[m/2 - 1] + [1] = [-m/2]$ im Ring $\mathbb{Z}/m\mathbb{Z}$. Es sei bemerkt, dass $m/2 = 2^{k-1}$ ist.

Ganze Zahlen mit fester Bit-Größe sind in den Prozessoren sowie in vielen Programmiersprachen direkt als Standarddatentypen vorhanden. In der Programmiersprache Go hat man etwa die Datentypen int8, int16, int32, int64 sowie uint8, uint16, unit32, uint64.

3.13 Bsp.

- Die Bit-Darstellung von 255 in uint8 ist 11111111. Die Addition von 255 mit 1 in uint8 ergibt 0 - die Zahl, deren Bit-Darstellung in diesem Typ 00000000 ist.

- Die Bit-Darstellung von 127 in int8 ist 0111111. Die Addition von 127 und 1 in int8 ergibt -128 - die Zahl, deren Bit-Darstellung in diesem Typ 11111111 ist.

4 Rechenprobleme

4.1 Def. Wir fixieren die **Standard-Kodierungen** durch Bit-Strings für Mengen $\mathbb{Z}, \mathbb{Q}$ sowie $\mathbb{Z}^n$ und $\mathbb{Q}^n$ mit $n \in \mathbb{N}$ wie folgt:

- Für $z \in \mathbb{Z}$ ist die Standard-Kodierung die Darstellung von z im Binärsystem. Diese Kodierung hat Größe $\Theta(\log_2(|z|+1)$.
- Jedes $q \in \mathbb{Q}$ kann eindeutig als $q = a/b$ mit $a \in \mathbb{Z}$ und $b \in \mathbb{N}$ fixiert werden. Danach wird das Paar $(a, b) \in \mathbb{Z} \times \mathbb{N}$ mit der Verwendung der Standardkodierungen von a und b kodiert (vgl. dazu den Beweis der Proposition (2.10) über das Kodieren von Paaren).
- $\mathbb{Z}^n$ und $\mathbb{Q}^n$ wird in Anlehnung an die oben festgelegten Stadardkodierungen von $\mathbb{Z}$ und $\mathbb{Q}$ kodiert.

Für x aus einer dieser Mengen ($\mathbb{Z}, \mathbb{Q}, \mathbb{Z}^n$ oder $\mathbb{Q}^n$) bezeichnen wir als $\llcorner x \lrcorner \in \{0,1\}^*$ die

Standardkodierung von x und als $\langle x \rangle$ die Länge dieser Kodierung.

4.2 Def. Ein Rechenproblem ist eine binäre Relation auf der Menge der Bit-Strings $\{0,1\}^*$. Mit anderen Worten ist ein Rechenproblem eine Menge

$$\Pi \subseteq \{0,1\}^* \times \{0,1\}^*.$$

Die Bedeutung von $(x,y) \in \Pi$ ist dabei: y ist eine mögliche Rückgabe für die Eingabe x. Somit ist ein Rechenproblem eine Eingabe-Rückgabe-Relation. Ein Rechenproblem Π algorithmisch zu lösen heißt es, einen Algorithmus zu finden, der für jede Eingabe x entscheidet, ob ein y mit $(x,y) \in \Pi$ existiert und ggf. ein solches y berechnet.

4.3 Def. Die Berechnung einer Funktion $f : \{0,1\}^* \to \{0,1\}^*$ ist ein Spezialfall eines Rechenproblems, bei dem man zu jeder Eingabe x eine Eindeutige Rückgabe $f(x)$ hat.

4.4 Def. Eine Teilmenge $L \subseteq \{0,1\}^*$ nennt man Sprache. Eine Sprache algorithmisch zu entscheiden, heißt es einen Algorithmus zu finden, der für jedes $x \in \{0,1\}^*$ entscheidet, ob x zu L gehört oder nicht. Die Entscheidung der Sprache L kann als die Berechnung der Funktion f mit

$$f(x) := \begin{cases} 1 & \text{für } x \in L, \\ 0 & \text{für } x \notin L. \end{cases}$$

formuliert werden. Die Rechenprobleme, die durch eine Sprache definiert werden, nennt man Entscheidungsprobleme. Es handelt sich dabei, um Probleme bei denen man nur zwei mögliche Rückgabewerte hat (ja/nein bzw. wahr/falsch bzw. 1/0).

4.5. Die vorigen drei Definitionen werden wir auch mit anderen Mengen an der Stelle von $\{0,1\}^*$ benutzen, unter der Voraussetzung, dass für diese Mengen eine Kodierung mit Bit-Strings festgelegt ist:

- Ein Rechenproblem ist eine Eingabe-Rückgabe-Relation $\Pi \subseteq X \times Y$ für die Menge X aller möglichen Eingaben (mit einer festgelegten Kodierung der Eingabe) und die Menge Y aller möglichen Rückgaben (mit einer festgelegten Kodierung der Rückgabe).

- Analog kann das Problem der algorithmischen Berechnung einer Abbildung $f : X \to Y$ beschrieben werden, sobald Kodierungen für X und Y festgelegt sind.

- Entscheidungsprobleme entsprechen der algorithmischen Berechnung der Prädikate $P : X \to \{\textsc{Falsch}, \textsc{Wahr}\}$.

4.6 Bsp.

- Das Problem

$$\{(a, p) \in \mathbb{N} \times \mathbb{Z}_{\geq 2} \,:\, p \text{ ist Primfaktor von } a\}$$

  kann mit Worten folgendermaßen formuliert werden: entscheide, für ein gegebenes $a \in \mathbb{N}$, ob a einen Primfaktor $p \in \mathbb{Z}_{\geq 2}$ besitzt und berechne ggf. einen solchen Faktor.

- Die Berechnung von $f : \mathbb{Z} \times \mathbb{N}_0 \to \mathbb{N}$ mit $f(a, k) := a^k$ kann mit Worten so formulieren, für gegebene $a \in \mathbb{Z}, k \in \mathbb{N}$ berechne die k-te Potenz von a.

- Das algorithmische Problem zur Sprache der (Bit-Kodierungen von) Primzahlen ist zu entscheiden, ob die Eingabe (die Bit-Kodierung einer) Primzahl ist. Mit Worten: entscheide, ob eine gegebene Zahl $z \in \mathbb{N}$ eine Primzahl ist.

5 Variablen, Zuweisungen und Kontrollstrukturen

5.1. Zur Beschreibung von Algorithmen unabhängig von einer konkreten Programmiersprache gibt es unter anderem folgende Möglichkeiten:

1. Beschreibung in einer natürlichen Sprache, evtl. mit der Verwendung mathematischer Formeln und Bezeichnungen.

2. Beschreibung in einem sogenannten Pseudocode.

3. Umsetzung in einer Programmiersprache.

Im Gegensatz zu einem richtigen Code, der sehr oft programmiertechnische Details einer konkreten Programmiersprache beinhaltet, kann man im Pseudocode solche Details vermeiden. Beim Pseudocode gibt es meistens nur Richtlinien und keine festen Vorgaben darüber, welche syntaktische Konstrukte eingesetzt werden. In seiner klassischen Form ahmt der Pseudocode die Programmiersprache ALGOL 60 nach, deren Syntax die allermeisten der heutzutage verbreiteten Programmiersprachen beeinflusst hat. Die bekannten Kontrollstrukturen wie if, for

usw. stammen z.B. aus ALGOL. Die Beschreibung von Algorithmen in einem ALGOL-ähnlichen Pseudocode ist in wissenschaftlichen Publikationen sehr verbreitet.

Ein Nachteil des Pseudocodes besteht darin, dass man ihn nicht auf einem Computer testen bzw. ausführen kann. Des Weiteren sind programmiertechnische Details aus der praktischen Sicht wichtig: sie können die praktische Effizienz der Algorithmen (tatsächliche Laufzeit usw.) erheblich beeinflussen.

5.2 Def. Eine **Algorithmus-** bzw. **Programm-Variable** ist ein Behälter für Werte bzw. andere Daten, die man mit Hilfe von Bit-Strings darstellen kann.

Da man auch mit Variablen arbeiten kann, deren Wert keine Zahl ist, nennt man Variablen auch allgemeiner **Objekte** und deren Wert nennt man auch gerne den **Zustand** eines Objektes.

Man beachte, dass man den Begriff Objekt nicht nur in der objektorientierten Programmierung sondern auch allgemein in der Programmierung in dem oben beschriebenen allgemein Sinn benutzt. Der Zustand eines Objekts kann während der Ausführung mit Hilfe von verschiedenen Befehlen und Operationen geändert werden.

Der Zustand eines Objekts kann durch eine Zuweisung geändert werden. Diese hat das

Format: Die Zuweisung hat das Format:

$$\text{OBJEKT} := \text{AUSDRUCK}$$

Hierbei kann die linke Seite der Name des Objekts sein oder auch ein Ausdruck, der ein Objekt festlegt. Die rechte ist ein Ausdruck, der ausgewertet werden kann. Eine weitere Bezeichnung, die man im Pseudocode für die Zuweisung benutzt ist $\leftarrow$.

5.3 Bsp. Wir führen den folgenden sehr kurzen Pseudocode manuell aus:

1: $x := 5$

2: $x := 2 \cdot x + 4$

Hier wird in der ersten Zeile einer Variablen x der Wert 5 zugewiesen. In der zweiten Zeile wird der Variablen x ein neuer Wert zugewiesen, wobei man sich bei der Zuweisung in der rechten Seite auf den aktuellen Wert bezieht. Der Verlauf der Ausführung ist somit wie folgt.

Nach der Ausführung der ersten Zeile ist ein Objekt mit dem Namen x entstanden, dessen aktueller Wert gleich 5 ist:

Variable	Wert
x	5

Nach der Auswertung der rechten Seite in der zweiten Zeile ist ein Objekt ohne Namen und mit dem Wert $2 \cdot 5 + 4 = 14$ entstanden.

Variable	Wert
x	5
	14

Nach der Ausführung der Zuweisung in der zweiten Zeile hat man dann

Variable	Wert
x	14

5.4 Def. Programmiersprachen, bei denen die Berechnungen auf den Objekten basieren, deren Zustand sich während der Ausführung ändert nennt man **imperativ**.

5.5. Die interne Sprache der Prozessoren (Assembly) ist imperativ, da sie auf den Befehlen basiert, die den Zustand der Register im Speicher und im Prozessor ändern.

5.6 Def. Die **Kontrollstrukturen** einer imperativen Sprache unterteilen sich in die folgenden Arten:

- *Verzweigungen*:

  `if-then`: Ausführung des Then-Blocks, nur wenn die If-Bedingung erfüllt ist.

  `if-then-else`. Ausführung des Then-Blocks, nur wenn die If-Bedingung erfüllt ist, und Ausführung des Else-Blocks, nur wenn die If-Bedingung nicht erfüllt ist.

- *Schleifen*:

  `for`: Iterieren über aufeinanderfolgende ganzzahlige Werte $a, \ldots, b$. Bei einer

Schleife mit $i := a, \ldots, b$ wird der For-Block (auch Rumpf der For-Schleife genannt) für alle i von a bis b (in dieser Reihenfolge) ausgeführt.

`while`: Solange die While-Bedingung erfüllt ist, den While-Block ausführen.

`repeat-until`: Den Repeat-Block ausführen. Wenn die Until-Bedingung wahr ist, zur nächsten Zeile kommen. Ansonsten wieder den Repeat-Block ausführen usw.

- *Sprungbefehl* `goto`: Übergang zur gegebenen Zeile des Programms/Algorithmus.

5.7. Es stellt sich heraus, dass man `goto` und `if-then` alle anderen Kontrollstrukturen aus Definition 5.6 umsetzen kann. Das Ziel der redundanten Kontrollstrukturen ist, die Lesbarkeit des Codes bzw. der Algorithmen zu erhöhen.

5.8 Bsp. Hier ist ein Pseudocode, der die Werte der Variablen x und y vertauscht, wenn am Anfang $x > y$ gilt:

```
1: if x > y :
2:     t := x
3:     x := y
4:     y := t
5: end
```

Das heißt, die drei Zuweisungen werden genau dann ausgeführt wenn beim Erreichen der Zeile 1 des Codes die Bedingung $x > y$ gilt. Die Variable t ist eine Zusatzvariable, die beim Vertauschen benutzt wird.

If-then-else ist analog aufgebaut. Im else-Teil stehen die Befehle, die ausgeführt werden, wenn die gegebene Bedingung *nicht* erfüllt ist. Die Verzweigungen lassen sich nach Belieben

verschachteln um komplexere Handlungsanweisungen aufzubauen.

5.9 Def. Ein **Array** A der Länge n ist eine Liste aus n Variablen, wobei die Variablen mit aufeinanderfolgenden ganzen Zahlen indiziert sind. In den allermeisten Programmiersprachen werden die Arrays beginnend mit 0 indiziert. Bei Indizierung ab 0 ist ein Array A der Länge n aus den Variablen $A[0], \ldots, A[n-1]$ zusammengesetzt, auf welche man durch die Angabe des Index i zugreifen kann. Die Variable $A[i]$ heißt die *i-te Komponente*, oder das *i-te Element* des Arrays A. Die Anzahl der Komponenten eines Arrays A wird als *Länge* von A bezeichnet und mit $\textsc{len}[A]$ notiert.

5.10 Bsp. Wir illustrieren nun eine andere Kontrollstruktur, die *for*-Schleife, indem wir zeigen, wie man mit ihrer Hilfe die Summe der Elemente eines n-elementigen Arrays bestimmen kann.

1: $S := 0$
2: **for** $i := 0, \ldots, \text{LEN}[A] - 1$ **:**
3: $\quad S := S + A[i]$
4: **end**

5.11 Bsp. Nachfolgend ein Beispiel, das zeigt wie man die Komponenten eines Arrays mit Hilfe einer while-Schleife umkehren kann:

1: $i := 0$
2: $j := \text{LEN}[A] - 1$
3: **while** $i < j$ **:**
4: $A[i]$ und $A[j]$ vertauschen
5: $i := i + 1$ $\triangleright$ *zum nächsten* i
6: $j := j - 1$ $\triangleright$ *zum vorigen* j
7: **end**

5.12. Im Pseudocode nutzen wir hier das Symbol ▷ für Kommentare, die den Zweck haben einzelne Abschnitte des Codes zu erläutern.

6 Prozeduren und Arten der Parameterübergabe

6.1 Def. Eine **Prozedur** bzw. eine **Programm-Funktion** bzw. ein **Unterprogramm** ist ein Code innerhalb eines Programms mit eigener Eingabe und Rückgabe. Prozeduren ohne Rückgabe sind auch möglich.

6.2. Stellen wir uns vor, wir müssen zur Lösung einer Rechenaufgabe immer wieder testen, ob $x \in [p, q]$ für gegebene $x, p, q \in \mathbb{Z}$ gilt. In diesem Fall lohnt es sich, eine sogenannte *Prozedur* anzulegen, welche genau diesen Test durchführt:

$b = \textsc{Ist-zwischen}(x, p, q)$

---

if $p \leq x \leq q$ oder $q \leq x \leq p$ **:**

$b = \textsc{Wahr}$

end

$b = \textsc{Falsch}$

Die Variablen x, p, q heißen **Eingabeparameter** der Prozedur und die Variable b heißt **Rückgabe-Variable**. In vielen modernen Programmiersprachen benutzt man für die Rückgabe keinen Variablennamen sondern den Befehl `return`. Das sieht dann so aus:

IST-ZWISCHEN(x, p, q)

if $p \leq x \leq q$ oder $q \leq x \leq p$ **:**

  return WAHR

end

return FALSCH

Durch den Befehl `return` wird die Prozedur mit dem vorgegebenen Wert an dieser Stelle beendet.

6.3. In manchen Sprachen (wie z.B. in C++) stehen mehrere Arten der Parameterübergabe zur Verfügung, wie z.B. **Übergabe durch Kopie** und die **Übergabe durch Referenz**. Wenn zum Beispiel im vorigen Pseudocode x, p und q durch Kopie übergeben werden, so entstehen bei jedem Aufruf der Prozedur die drei Variablen x, p und q, welche dann entsprechend initialisiert werden. Etwa, bei der Ausführung von $\textsc{Ist-zwischen}(a, b, c)$ mit $x = a, p = b, q = c$.

6.4 Bsp. Bei der Übergabe durch Referenz, ist der Eingabeparameter lediglich ein weiterer Name für eine Variable, die bereits existiert. Wir illustrieren dies am Beispiel vom Vertauschen in konkretem C++-Code:

```
void vertauschen(int& x,int& y) {
    int t=x;
    x=y;
    y=t;
}
int main() {
    int a=2,b=3;
    vertauschen(a,b);
    return 0;
}
```

Damit die Werte a und b in der `main`-Funktion vertauscht werden, müssen die Eingabeparameter x und y Referenzvariablen sein. In diesem Fall sind x und y zweite Namen für a bzw.

b. Die Variable t ist eine *lokale* Variable der Funktion `vertauschen`. Sie entsteht bei jeder Ausführung von `vertauschen` und verschwindet nach der Terminierung dieser Funktion.

6.5. In der Beschreibung von Algorithmen im Pseudocode halten wir uns im Folgenden an die Konvention, bei der Parameterübergabe Arrays durch Referenz und einfache Datentypen durch Kopie zu übergeben.

7 Datentypen und Datenstrukturen

7.1 Def. Ein **Datentyp** bzw. eine **Datenstruktur** T ist durch die folgenden Angaben definiert:

der **Wertebereich** für den Zustand der Objekt des Datentyps T

das **Interface:** Prozeduren (unter anderem die Grundoperationen, -Relationen) für den Datentyp T, durch welche man den Zustand der Objekte vom Typ T ablesen und ändern kann.

7.2. Neben den Angaben des Wertebereiches und des Interface macht man auch oft die Vorgaben zur **Effizienz** (der Speicheraufwand und die Laufzeiten der Prozeduren im Interface). Zum Beispiel wäre ein Stack keine wertvolle Datenstruktur, wenn die Grundoperationen PUSH und POP dafür Ressourcen-aufwändig wären.

7.3. Man unterscheidet zwischen **einfachen** und **zusammengesetzten Datentypen**. Das Objekt eines zusammengesetzten Datentyps ist aus den Objekten einfacherer Datentypen zusammengesetzt.

7.4. Verschiedene Operationen und Konstruktion für Mengen, die wir eingeführt haben entsprechen verschiedenen Datentypen, die man in vielen Programmiersprachen findet:

Mathe-Begriff	Menge	Datenstruktur
Element einer endlichen Menge	$\{x_1, \ldots, x_n\}$	Aufzählung
n-Tupel mit Komponenten in T	T^n	Array der Länge n über T
String über T	T^*	Array über T
n-Tupel	$X_1 \times \cdots \times X_n$	Verbund mit n Komponenten
Abbildung		Wörterbuch mit
		Schlüsseln in K und Werten in V
Menge		Menge
Multimenge		Multimenge

Arrays werden in der Programmierung auch Listen oder Felder genannt. Die Verunde nennt man auch Structures oder Records. Während man die Komponenten eines n-Tupels mit Zahlen $1, \ldots, n$ indexiert, werden für die Komponenten eines Verbunds Namen festgelegt. Wörterbücher

werden auch Maps genannt.

7.5 Aufg. Schlagen Sie nach, wie die folgenden Datenstrukturen definiert sind und umgesetzt werden können:

- Stack, Warteschlange und Deque
- Heap
- Wörterbuch (= Map)

7.6. Bei der konkreten Umsetzung bzw. Implementierung einer Datenstruktur werden oftmals sogenannte *Zeiger* verwendet. Das sind Adress-Variablen, d.h., eine Zeiger-Variable speichert die Adresse eines Ortes (einer anderen Variable) im Speicher des Rechners. Für Zeiger gibt es zwei Grundoperationen:

- ADRESSE von einem Objekt, und
- OBJEKT unter gegebener Adresse.

Manche Programmiersprachen (wie C, C++, Go) stellen direkt Zeiger zur Verfügung. In vielen höheren Programmiersprachen haben die komplexen Objekte das Zeigerverhalten (z.B. Listen in Python), auch wenn die Zeiger in solchen Sprachen nicht direkt vorhanden sind.

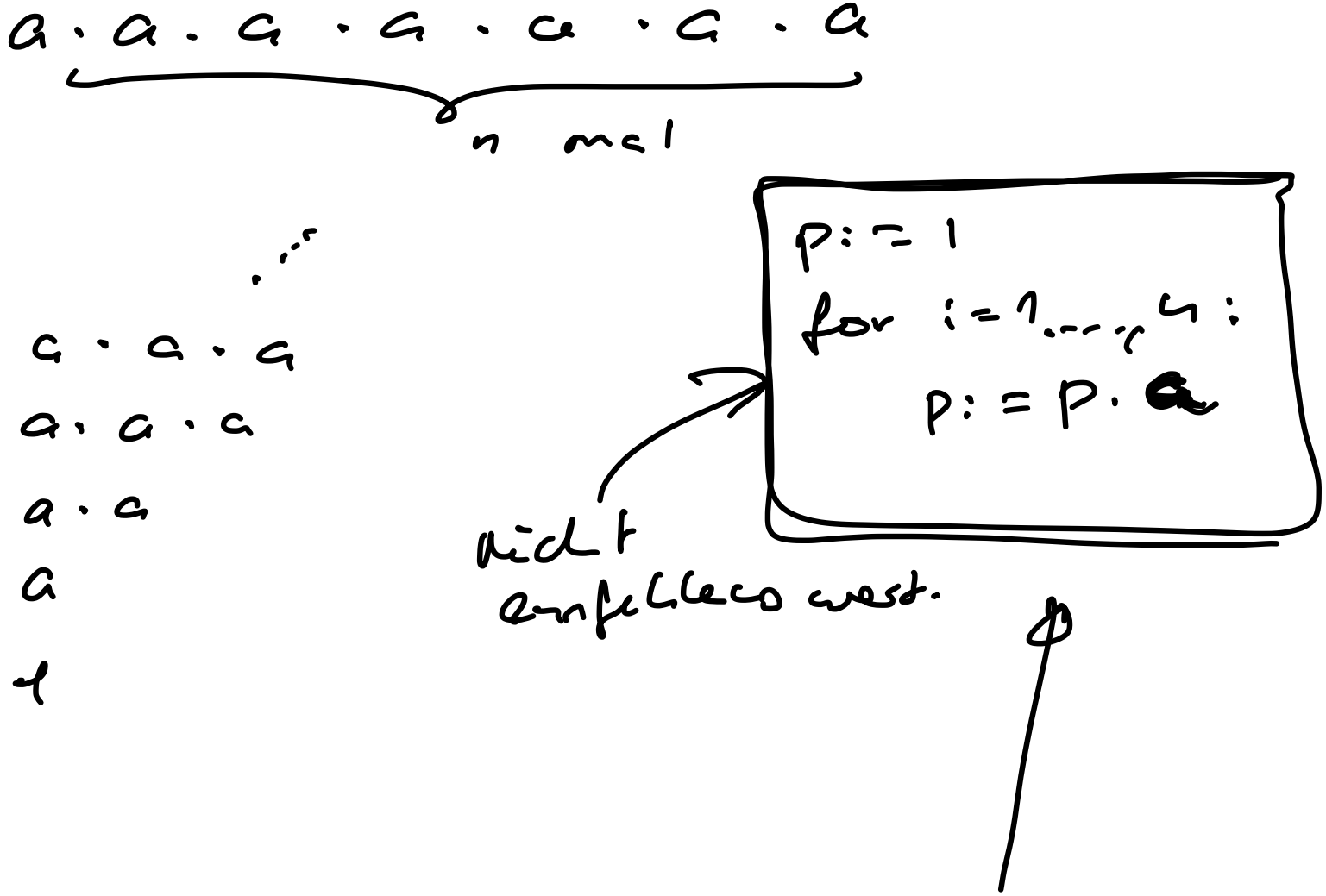
7.7 Aufg. Schlagen Sie nach wie die folgenden Datenstrukturen auf den Zeigern basierend umgesetzt werden:

- Einfach verkette Listen (linear und zyklisch)
- Doppelt verkette Listen (linear und zyklisch)
- Binäre Bäume

8 Rekursion

8.1 Def. Prozeduren, die sich selbst aufrufen, heißen **rekursiv**.

8.2 Bsp. Hier ein Beispiel einer Prozedur, die a^n für $a \in \mathbb{Z}$ und $n \in \mathbb{N}_0$ mittels einer Rekursion mit Hilfe von $\Theta(\log n)$ arithmetischen Operationen berechnet.



q

$\Rightarrow$ Klammerung

$$\Big(\underbrace{(a \cdot a) \cdot \underbrace{(a \cdot a)}_{q}}_{q} \cdot \underbrace{((a \cdot a) \cdot (a \cdot a))}_{q}\Big)$$

$p := \textsc{Potenz}(a, n)$

if $n = 0$ ▷ *Terminierungsbedingung* **:**

$p := 1$

else if $n = 1$ ▷ *Terminierungsbedingung*

$p := a$

else if n gerade **:**

$q := \textsc{Potenz}(a, n/2)$ ▷ q *setzt man gleich* $a^{n/2}$

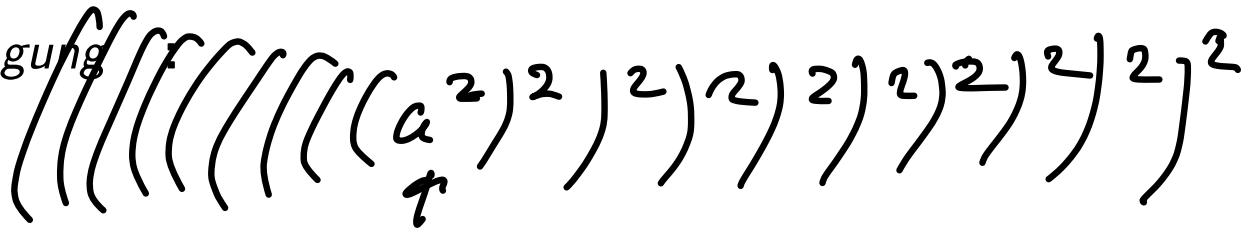
$p := q \cdot q$ ▷ p *setzt man gleich* $q^2 = (a^{n/2})^2 = a^n$

else:

$q := \textsc{Potenz}(a, (n-1)/2)$ ▷ q *setzt man gleich* $a^{(n-1)/2}$

$p := a \cdot q \cdot q$ ▷ p *setzt man gleich* $a \cdot q^2 = a \cdot (a^{(n-1)/2})^2 = a^n$

end



a^{1024} 10 Multipl.

a^{2^k} k Multipl.

$n = 2^k$

$k = \log_2 n$

Diese rekursive Prozedur ist in vielen Situationen besser als die nicht-rekursive iterative

Umsetzung mit Hilfe $\Theta(n)$ arithmetischen Operationen berechnet:

$p := \textsc{Potenz-Langsam}(a, n)$

$p := 1$

for $i := 1, \ldots, n$ **:**

$\triangleright$ *An dieser Stelle gilt* $p = a^{i-1}$ *– die Invariante der Schleife*

$p := p \cdot a$

end

8.3 Prop. Die rekursive Prozedur $\textsc{Potenz}(a, n)$ berechnet a^n mit Hilfe von $\Theta(\log_2 n)$ Multiplikationen.

Beweis. Wir zeigen zuerst durch Induktion über n, dass $\textsc{Potenz}(a, n)$ für jedes $n \in \mathbb{N}_0$ mit der Rückgabe a^n terminiert. Diese Behauptung ist in den Fällen $n = 0$ und $n = 1$ klar. Angenommen, $n > 1$ und es ist bereits verifiziert worden, dass $\textsc{Potenz}(a, k)$ für jedes $k = 0, \ldots, n-1$ mit der Rückgabe a^k terminiert. Ist n gerade, so ist $n/2$ ganze Zahl, die kleiner als n ist, und es gilt $a^n = (a^{n/2})^2$. Nach der Induktionsvoraussetzung terminiert $\textsc{Potenz}(a, n/2)$ mit der Rückgabe $a^{n/2}$, sodass $\textsc{Potenz}(a, n)$ mit der Rückgabe $(a^{n/2})^2 = a^n$ terminiert. Ist n ungerade, so ist $(n-1)/2$ ganze Zahl, die ebenfalls kleiner als n ist, und es gilt $a^n = a \cdot (a^{(n-1)/2})^2$. Nach der Induktionsvoraussetzung terminiert $\textsc{Potenz}(a, (n-1)/2)$ mit der Rückgabe $a^{(n-1)/2}$, sodass $\textsc{Potenz}(a, n)$ mit der Rückgabe $a \cdot (a^{(n-1)/2})^2 = a^n$ terminiert.

Es bleibt zu zeigen, dass die Anzahl der Multiplikation, die in $\textsc{Potenz}(a, n)$ benutzt wird

die Ordnung $\Theta(\log_2 n)$ hat. Dafür zeigen wir durch Induktion über n, dass die Anzahl der Multiplikation $M(n)$ bei der Ausführung von $\textsc{Potenz}(a, n)$ für jedes $n \in \mathbb{N}$ mit $n \geq 2$ die Bedingung

$$\frac{1}{2}\log_2 n \leq M(n) \leq 2\log_2 n$$

erfüllt. Für $n = 2$ ist das klar: $\textsc{Potenz}(a, 2)$ multipliziert a mit $q = \textsc{Potenz}(a, 1)$ und $\textsc{Potenz}(a, 1)$ macht keine Multiplikationen. Sei $n \geq 3$ und sei die Ungleichung $\log_2 k \leq M(k) \leq 2\log_2 k$ für alle $k = 1, \ldots, n-1$ verifiziert. Ist n gerade, so haben wir $M(n) = M(n/2) + 1$. Da $n/2 \in \{1, \ldots, n-1\}$ gilt, erhalten wir aus der Induktionsvoraussetzung einerseits

$$M(n) = M(n/2) + 1 \geq 1 + \frac{1}{2}\log_2(n/2) = \frac{1}{2}\log_2 n$$

und andererseits

$$M(n) \leq M(n/2) + 1 \leq 1 + 2\log_2 n/2 + 1 = -1 + 2\log_2 n \leq 2\log_2 n.$$

Ist n ungerade, so haben wir $M(n) = M((n-1)/2) + 2$. Da $(n-1)/2 \in \{1, \ldots, n-1\}$ gilt, erhalten wir aus der Induktionsvoraussetzung einerseits

$$M(n) = M((n-1)/2) + 2 \geq 2 + \frac{1}{2}\log_2((n-1)/2) \geq \frac{1}{2}\log_2 n$$

und andererseits

$$M(n) = M((n-1)/2) + 2 \leq 2 + 2\log_2((n-1)/2)) = 2\log_2(n-1) \leq 2\log_2 n.$$

□

8.4. Es kann überprüft werden, dass die rekursiven Algorithmen auch ohne Rekursion, etwa mit Schleifen und Arrays, umgesetzt werden können. Dies gilt auch für das Potenzieren oben. Die rekursiven Umsetzungen sind aber manchmal leichter zu verstehen oder intuitiver als das nicht-rekursive Analogon dazu.

8.5 Def. Zur Ausführung einer rekursiven Prozedur P auf einer Eingabe x definiert man den **Rekursionsbaum** als das Paar (V, A), in dem V die Menge aller Aufrufe von P während der Ausführung von P auf x ist und A ist die Menge aller paare (u, v) derart, dass der Aufruf v direkt aus u stattgefunden hat. Das größte k mit $(a_0, a_1), \ldots, (a_{k-1}, a_k) \in A$ für gewisse $a_0, \ldots, a_k$ nennt man die **Rekursionstiefe** der Ausführung von P auf x. Den Aufruf von P auf x nennt man die Wurzel des Baums.

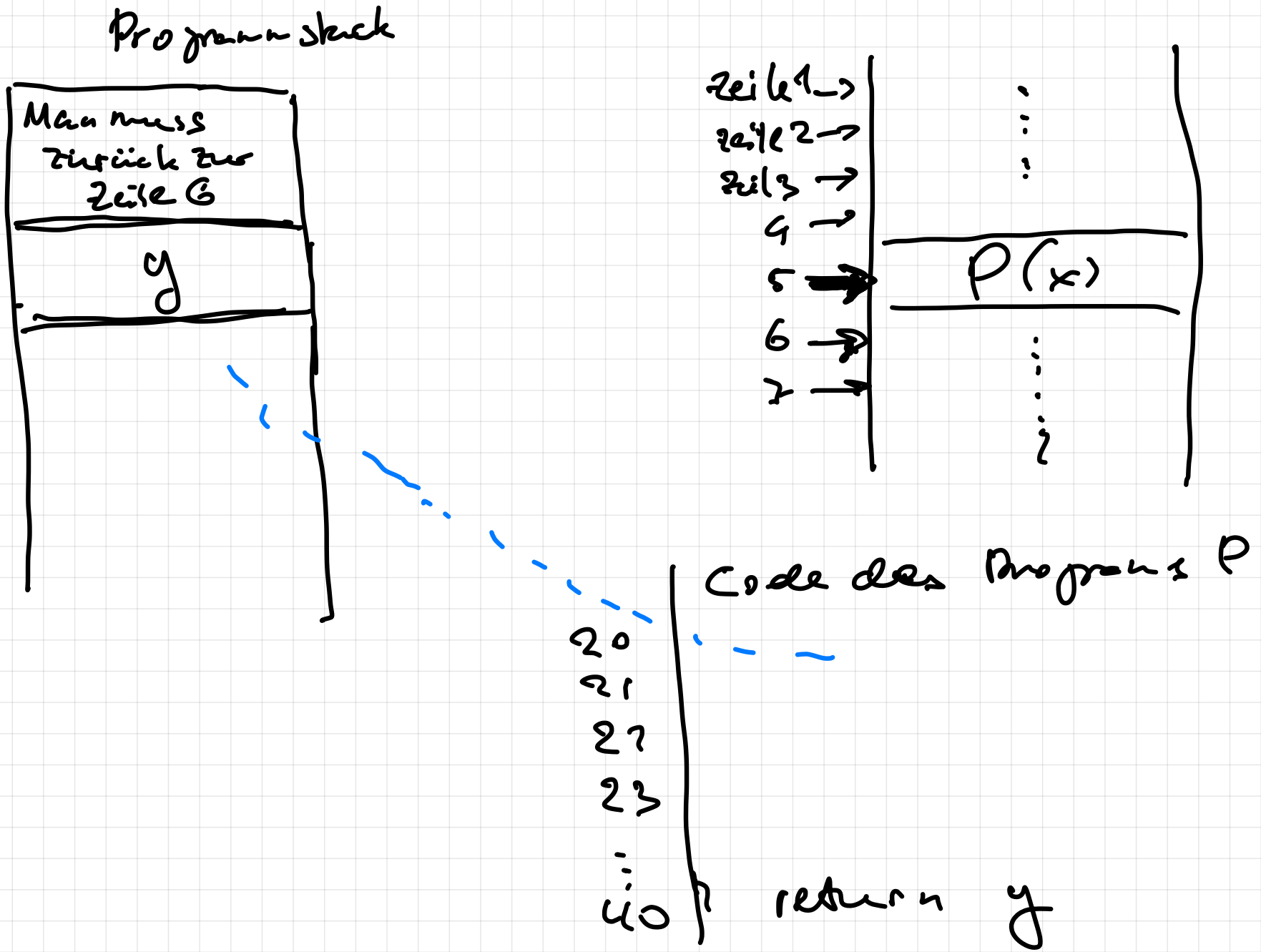
8.6 Bsp. Der Rekursionsbaum von $\textsc{Potenz}(a, n)$ ist ein Baum ohne Verzweigungen.

$$\textsc{Potenz}(a, 10)$$
$$\downarrow$$
$$\textsc{Potenz}(a, 5)$$
$$\downarrow$$
$$\textsc{Potenz}(a, 2)$$
$$\downarrow$$
$$\textsc{Potenz}(a, 1)$$

Rekursionsbäume mit Verzweigungen erhält man, wenn während der Ausführung einer rekursiven Prozedur mehr als ein rekursiver Aufruf stattfindet.

$a^{10} = (a^5)^2$ $\qquad$ $a^5 = a \cdot (a^2)^2$


a, 10
Zeile ....
a, 5
Zeile ...
a, 2
Zeile ..
a, 1
Zeile ...


Programmstack
Man muss zurück zur Zeile 6
y
Zeile 1
Zeile 2
Zeile 3
4
5
6
7
P(x)
i
Code des Programms P
20
21
27
23
40
return y

8.7. Auf der praktischen Seite erfolgt die Umsetzung der Aufrufe von Prozeduren und, insbesondere rekursiven Prozeduren, in Betriebssystemen und Programmiersprachen mit Hilfe des sogenannten Programmstacks. Bei jedem Aufruf wird (intern,auf der Ebene der Maschinensprache) die Stelle im Programm notiert, zu der man nach der Ausführung der Prozedur zurückkehren soll. Außerdem werden durch einen Aufruf potenziell neue Variablen eingeführt (lokale Variablen der Prozedur, Eingabeparameter). All diese Daten werden auf dem sogenannten **Aufrufstapel** (engl. call stack, procedure stack) aufgehoben. Die Größe des Aufrufstapels ist standardmäßig ziemlich eingeschränkt, sodass es bei der Ausführung von reksuriven Prozeduren mit einer großen Rekursionstiefe zu einem Überlauf des Aufrufstapels kommen kann.

In höheren Programmiersprache, bei denen die Rekursion nicht unbedingt direkt durch den Aufrufstapel umgesetzt wird, wird oft trotzdem das Maximum der Rekursionstiefe festgelegt. Hier ein Beispiel eines Python-Codes, der diese Problematik illustriert:

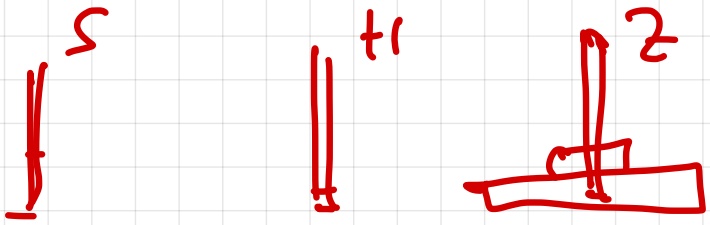
```
def dive(x):
```

```
    print(x)
    dive(x+1)
```

Auf meinem Computer führt der Aufruf von dive auf der Eingabe 0 zum eine Stacküberlauf in der Rekursionstiefe 995. Die maximale Rekursionstiefe kann oft manuell erhöht werden, und es gibt auch Sprachen, die in gewissen Situationen die rekursiven Prozeduren in nicht-rekursiven Maschinencode übersetzen können. Generell gilt aber: wenn man eine rekursive Prozedur mit einer hohen Rekursionstiefe hat, soll man sich Gedanken darüber machen, wie man diese Prozedur in eine nicht-rekursiv umsetzen könnte. Sehr oft kann man dabei seinen eigenen Stapel anlegen.

n=1

S → Z



h=2

S → H

S → Z

H → Z

n=3

1
2
3

1
⋮
n-1
n

S H Z

8.8 Bsp (Türme von Hanoi)**.** Gegeben sind drei Stäbe: Startstab, Hilfsstab und Zielstab. Auf dem Startstab sind n Scheiben der Größen 1 bis n aufgestapelt, so dass die Scheiben von oben nach unten betrachtet nach der Größe steigend sortiert sind. Die Rechenaufgabe besteht darin, eine Folge von Schritten zu bestimmen, mit denen man alle n Scheiben vom Startstab auf den Zielstab derart versetzt, dass in jedem Schritt die Scheiben von jedem der drei Stäbe sortiert sind (von oben nach unten betrachtet aufsteigend nach der Größe).

Eine rekursive Lösung dieser Aufgabe sieht so aus: das Ziel ist n Scheiben vom Start s nach Ziel z mit Hilfe von h als Hilfsstab zu versetzen. Wir versetzen die $n-1$ Scheiben vom Start- auf den Hilfsstab, indem wir das Zielstab als Hilfsstab benutzen. Danach wird die größte Scheibe verfügbar: sie kann auf den Zielstab versetzt werden. Danach ist der Startstab frei und kann bei der Versetzung der $n-1$ Scheiben vom Hilfs- auf den Zielstab als Hilfsstab benutzt werden.

HANOI-SCHRITTE(n, s, h, z)

if $n = 1$ ▷ *Terminierungsbedingung* **:**

  print: von s nach z

else:

  HANOI-SCHRITTE$(n-1, s, z, h)$ ▷ *alle bis auf die größte Scheibe auf* h

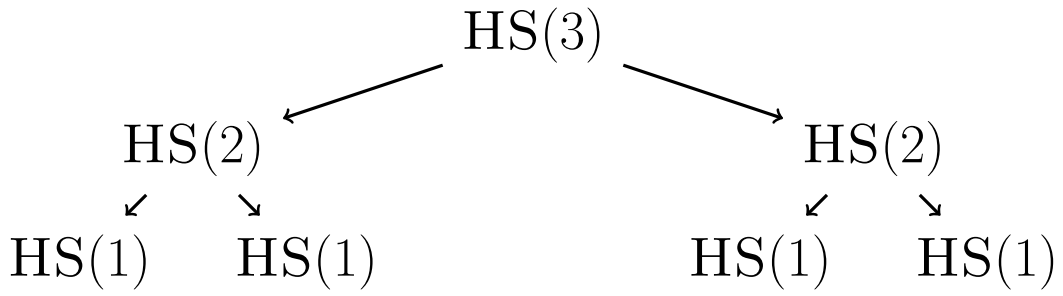
  ▷ *Wir verschieben die oberste Scheibe endgültig:*

  print: von s nach z

  HANOI-SCHRITTE$(n-1, h, s, z)$ ▷ *die restlichen* $n-1$ *Scheiben auf* z

end

Der Rekursionsbaum für $n = 3$ ist wie folgt:



Hierbei steht $\mathrm{HS}(k)$ für $\textsc{Hanoi-Schritte}(k, s, h, z)$ mit irgendeiner Wahl von s, h, z.

Hier Umsetzung in Python mit der Verwendung von Iteratoren:

```
def hanoi_schritte(n,Start,Hilfe,Ziel):
    """
        n Scheiben die auf den drei aufgestappelt sind
        von Start nach Ziel mithilfe von Hilfe zu versetzen
    """
    if n==1:
        yield (Start,Ziel)
    else:
        for schritt in hanoi_schritte(n-1,Start=Start,Hilfe=Ziel,Ziel=Hilfe):
            yield schritt
```

```
        yield (Start,Ziel)
        for schritt in hanoi_schritte(n-1,Start=Hilfe,Hilfe=Start,Ziel=Ziel):
            yield schritt

def hanoi_movie(n):
    T=[list(range(1,n+1)),[],[]]
    yield T
    for a,b in hanoi_schritte(n,0,1,2):
        T[b].append(T[a].pop())
        yield T

def hanoi_movie_run(n):
    for T in hanoi_movie(n):
        print(T[0])
        print(T[1])
        print(T[2])
        print()
        input('press any key to watch further')
```

Die Demo-Datei für 4 Türme: https://de.wikipedia.org/wiki/T%C3%BCrme_von_Hanoi#/media/Datei:Tower_of_Hanoi_4.gif

9 Random Access Machine

9.1. Die **Random Access Machine** (kurz **RAM**), oder auf deutsch, **Maschine mit wahlfreiem Zugriff**, wird unsere Idealisierung bzw. mathematische Abstraktion des realen Rechners sein. Alle Analysen und Entwürfe von Algorithmen in diesem Kurs werden im Rahmen der RAM durchgeführt.

Wir nehmen an, dass die Zellen unserer Maschine ganze Zahlen beliebiger Größe speichern können (d.h., die Bit-Größe der Speicherzellen ist unendlich). Die Speichergröße (d.h., die Anzahl der Speicherzellen) ist ebenfalls unbeschränkt (d.h., unendlich). Wir können des Weiteren alle anderen Datentypen auf der Basis der ganzen Zahlen umsetzen.

Die Random Access Machine kann auch rein formal eingeführt werden. Wir betrachten hier (zunächst) allerdings eine etwas informelle Beschreibung, in der wir festlegen welche Datentypen, Operationen und Kontrollstrukturen für uns elementar sind.

Als **Grundoperationen** für unsere RAM erlauben wir:

- Zuweisung (für ganzzahlige Datentypen)

- Addition von ganzzahligen Variablen
- Multiplikation einer ganzzahligen Variablen mit einer Konstanten
- Ganzzahlige Division einer ganzzahligen Variablen durch eine Konstante
- Zugriff zu Speicherzellen über einen Index
- Vergleichsoperationen $<$, $\leq$, $=$, $\geq$, $>$
- Kontrollstrukturen if-then-else, while, for

Eine genauere Beschreibung einer RAM findet man bei [Lov20, Sect. 1.3].

9.2. Wir lassen die Multiplikation von zwei ganzzahligen Variablen in unserem Modell nicht als Grundoperation zu. Denn, wenn das eine Grundoperation wäre, so hätte der folgende Algorithmus die Laufzeit $O(n)$:

$x := 2$
for $i = 1, \ldots, n$ **:**
  $x := x^2$
end

Dieser Algorithmus würde also 2^{2^n} in der Zeit $O(n)$ berechnen. Die Zahl 2^{2^n} hat allerdings 2^n+1 Binärstellen. Wir würden also eine Zahl exponentieller Bit-Größe in linearer Zeit berechnen, was wir als unrealistisch ansehen.

10 Polynomialzeit-Berechenbarkeit

10.1 Def. Sei A Algorithmus im Rahmen eines Rechenmodells, dessen Schritte als Grundoperationen im gegebenen Rechenmodell beschrieben sind.

Die **Laufzeit** $L(A, x)$ von A auf der Eingabeinstanz x ist die Anzahl der Schritte des Algorithmus A auf der Eingabe x bis zur Terminierung.

Die **Laufzeitfunktion** $L_A : \mathbb{N} \to \mathbb{N}$ von A ist definiert als

$$L_A(n) := \max \{L(A, x) \, : \, x \text{ Eingabe für } A \text{ mit } \langle x \rangle \leq n\} \, .$$

Das heißt, $L_A(n)$ ist die maximale Laufzeit auf allen Eingaben, deren Eingabelänge höchstens n ist.

10.2 Def. Eine Funktion $p : \mathbb{R} \to \mathbb{R}$ heißt **Polynomfunktion** vom Grad $d \in \mathbb{N}_0$, falls p die Form

$$p(t) = c_0 + c_1 t + c_2 x t 2 + \ldots + c_d t^d$$

hat, mit $c_0, \ldots, c_d \in \mathbb{R}$ und $c_d \neq 0$.

10.3 Def. Wir nennen einen Algorithmus A **polynomial** oder **Polynomialzeit-Algorithmus** (bezüglich des zugrundeliegenden Rechenmodells und Kodierungsschemas für die Eingabe), falls eine Polynomfunktion p existiert, so dass $L_A(n) \leq p(n)$ für alle $n \in \mathbb{N}_0$ gilt.

10.4. Ein Polynomialzeit-Algorithmus A ist ein Algorithmust mit der Laufzeitfunktion der Ordnung $O(n^d)$ für ein $d \in \mathbb{N}_0$.

10.5. In der Theorie nennt man Polynomialzeit-Algorithmen oft auch effiziente Algorithmen. Welche Algorithmen aus der praktischen Sicht effizient sind, lässt sich schwer formal beschreiben.

10.6 Def. Ein Rechenproblem heißt **Polynomialzeit-berechenbar**, wenn es ein Polynomialzeit-Algorithmus existiert, der das Rechenproblem löst.

Ein Entscheidungsproblem bzw. eine Sprache heißt **Polynomialzeit-entscheidbar**, wenn es ein Polynomialzeit-Algorithmus existiert, der das Problem bzw. die Sprache entscheidet.

10.7 Def. Die Menge aller Polynomialzeit-entscheidbaren Sprachen $L \subseteq \{0,1\}^*$ wird die Komplexitätsklasse P genannt. Als Formel:

$$\mathrm{P} := \{L \subseteq \{0,1\}^* \ : \ L \text{ Polynomialzeit-entscheidbar}\}$$

Literaturverzeichnis

[AZ02] Aigner, Ziegler. Das Buch der Beweise. Springer 2002

[Ber17] Berghammer: Mathematik für Informatiker. Grundlegende Begriffe und Strukturen. Springer Vieweg 2017

[Ber19] Berghammer: Mathematik für Informatiker. Grundlegende Begriffe und Strukturen und ihre Anwendung. Springer Vieweg 2019

[Big05] Biggs. Discrete mathematics. Oxford University Press 2005

[Bri01] Mathematik für Inforamtiker. Einführung an praktischen Beispielen aus der Welt der Computer. München: Hanser 2001

[CLRS17] Cormen, T. H., Leiserson, C. E., Rivest, R., Stein, C. Algorithmen-Eine Einführung. De Gruyter Oldenbourg. 2017.

[GR14] Goebbels, Rethmann. Mathematik für Informatiker: eine aus der Informatik motivierte Einführung mit zahlreichen Anwendungs- und Programmbeispielen. Springer Vieweg 2014

[GS11] Heinz Peter Gumm, Manfred Sommer. Einführung in die Informatik, Oldenbourg Verlag 2011

[KK15] Knauer, Knauer. Diskrete und algebraische Strukturen - kurz gefasst. Springer Spektrum 2015.

[KP09] Kreußler, Pfister. Mathematik für Informatiker: Algebra, Analysis, Diskrete Strukturen. Springer 2009.

[LLM21] Lehman, Leighton, Meyer. Mathematics for Computer Science. Lecture notes at MIT. `https://courses.csail.mit.edu/6.042/spring18/mcs.pdf`

[Lov20] Lovász, László. Complexity of Algorithms. Lecture Notes. 2020. `https://web.cs.elte.hu/~kiraly/complexity.pdf`

[Sch12] Schubert. Mathematik für Informatiker: ausführlich erklärt mit vielen Programmbeispielen und Aufgaben.

[Ste01] Steger. Diskrete Strukturen 1. Kombinatorik, Graphentheorie, Algebra. Springer 2001

[Tit19] Peter Tittmann. Einführung in die Kombinatorik, 3. Auflage, Springer 2019