16 TURINGMASCHINEN

Turingmaschinen sind eine und waren im wesentlichen die erste mathematische Präzisierung des Begriffes des *Algorithmus* so wie er klassisch verstanden wird: Zu jeder endlichen Eingabe wird in endlich vielen Schritten eine endliche Ausgabe berechnet.

Algorithmus

Einen technischen Hinweis wollen wir an dieser Stelle auch noch geben: In dieser Einheit werden an verschiedenen Stellen partielle Funktionen vorkommen. Das sind rechtseindeutige Relationen, die nicht notwendig linkstotal sind (siehe Abschnitt 3.2). Um deutlich zu machen, dass eine partielle Funktion vorliegt, schreiben wir im folgenden $f : M \dashrightarrow M'$. Das bedeutet dann also, dass für ein $x \in M$ entweder eindeutig ein Funktionswert $f(x) \in M'$ definiert ist, oder dass *kein* Funktionswert $f(x)$ definiert ist. Man sagt auch, f sei an der Stelle x undefiniert.

16.1 ALAN MATHISON TURING

Alan Mathison Turing wurde am 23.6.1912 geboren.

Mitte der Dreißiger Jahre beschäftigte sich Turing mit Gödels Unvollständigkeitssätzen und Hilberts Frage, ob man für jede mathematische Aussage algorithmisch entscheiden könne, ob sie wahr sei oder nicht. Das führte zu der bahnbrechenden Arbeit *"On computable numbers, with an application to the Entscheidungsproblem"* von 1936.

Von 1939 bis 1942 arbeitete Turing in Bletchly Park an der Dechiffrierung der verschlüsselten Texte der Deutschen. Für den Rest des zweiten Weltkriegs beschäftigte er sich in den USA mit Ver- und Entschlüsselungsfragen. Mehr zu diesem Thema (und verwandten) können Sie in Vorlesungen zum Thema *Kryptographie* erfahren.

Nach dem Krieg widmete sich Turing unter anderem dem Problem der *Morphogenese* in der Biologie. Ein kleines bisschen dazu findet sich in der Vorlesung „Algorithmen in Zellularautomaten".

Alan Turing starb am 7.6.1954 an einer Zyankalivergiftung.

Eine Art „Homepage" von Alan Turing findet sich unter `http://www.turing.org.uk/turing/index.html`.

16.2 TURINGMASCHINEN

Als *Turingmaschine* bezeichnet man heute etwas, was Turing (1936) in seiner Arbeit eingeführt hat. Die Bezeichnung selbst geht wohl auf eine Besprechung von Turings Arbeit durch Alonzo Church zurück (laut einer WWW-Seite von J. Miller; `http://jeff560.tripod.com/t.html`, 14.1.09).

Eine Turingmaschine kann man als eine Verallgemeinerung endlicher Automaten auffassen, bei der die Maschine nicht mehr darauf beschränkt ist, nur feste konstante Zahl von Bits zu speichern. Im Laufe der Jahrzehnte wurden viele Varianten definiert

und untersucht. Wir führen im folgenden nur die einfachste Variante ein.

Zur Veranschaulichung betrachte man Abbildung 16.1. Im oberen Teil sieht man die *Steuereinheit*, die im wesentlichen ein endlicher *Mealy-Automat* ist. Zusätzlich gibt es ein *Speicherband*, das in einzelne *Felder* aufgeteilt ist, die mit jeweils einem Symbol beschriftet sind. Die Steuereinheit besitzt einen *Schreib-Lese-Kopf*, mit dem sie zu jedem Zeitpunkt von einem Feld ein Symbol als Eingabe lesen kann. Als Ausgabe produziert sie die Turingmaschine ein Symbol, das auf das gerade besuchte Feld geschrieben wird und sie kann den Kopf um ein Feld auf dem Band nach links oder rechts bewegen. Ausgabesymbol und Kopfbewegung ergeben sich ebenso eindeutig aus aktuellem Zustand der Steuereinheit und gelesenem Symbol wie der neue Zustand der Steuereinheit.

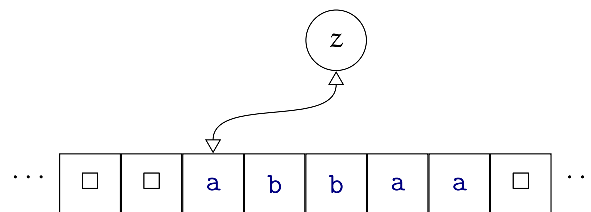


Abbildung 16.1: schematische Darstellung einer (einfachen) Turingmaschine

Turingmaschine

Formal kann man sich also eine *Turingmaschine* $T = (Z, z_0, X, f, g, m)$ festgelegt vorstellen durch

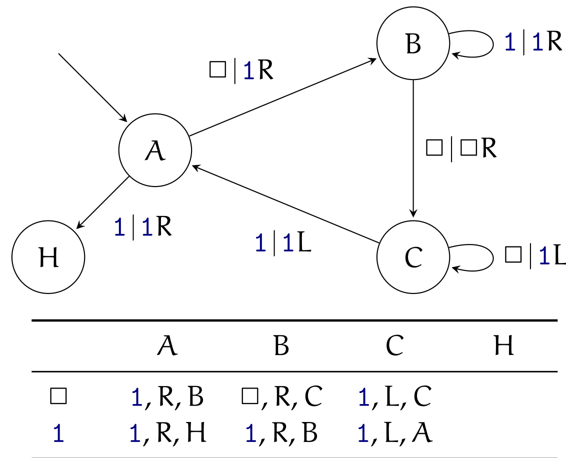
- eine Zustandsmenge Z
- einen Anfangszustand $z_0 \in Z$
- ein Bandalphabet X
- eine partielle Zustandsüberführungsfunktion $f : Z \times X \dashrightarrow Z$
- eine partielle Ausgabefunktion $g : Z \times X \dashrightarrow X$ und
- eine partielle Bewegungsfunktion $m : Z \times X \dashrightarrow \{-1, 0, 1\}$

Wir verlangen, dass die drei Funktionen f, g und m für die gleichen Paare $(z, x) \in Z \times X$ definiert bzw. nicht definiert sind. Warum wir im Gegensatz zu z. B. endlichen Akzeptoren erlauben, dass die Abbildungen nur partiell sind, werden wir später noch erläutern.

Es gibt verschiedene Möglichkeiten, die Festlegungen für eine konkrete Turingmaschine darzustellen. Manchmal schreibt man die drei Abbildungen f, g und m in Tabellenform auf, manchmal macht man es graphisch, ähnlich wie bei Mealy-Automaten. In Abbildung 16.2 ist die gleiche Turingmaschine auf beide Arten definiert. Die Bewegungsrichtung notiert man oft auch mit L (für links) statt -1 und mit R (für rechts) statt 1.

Konfiguration

Eine Turingmaschine befindet sich zu jedem Zeitpunkt in einem „Gesamtzustand", den wir eine *Konfiguration* nennen wollen. Sie ist vollständig beschrieben durch



	A	B	C	H
□	1, R, B	□, R, C	1, L, C	
1	1, R, H	1, R, B	1, L, A	

Abbildung 16.2: Zwei Spezifikationsmöglichkeiten der gleichen Turingmaschine; sie heißt BB3.

- den aktuellen Zustand $z \in Z$ der Steuereinheit,
- die aktuelle Beschriftung des gesamten Bandes, die man als Abbildung $b : \mathbb{Z} \to X$ formalisieren kann, und
- die aktuelle Position $p \in \mathbb{Z}$ des Kopfes.

Eine Bandbeschriftung ist also ein potenziell unendliches „Gebilde“. Wie aber schon in Abschnitt 5.2 erwähnt und zu Beginn dieser Einheit noch ein betont, interessieren in weiten Teilen der Informatik endliche Berechnungen, die aus endlichen Eingaben endliche Ausgaben berechnen. Um das adäquat zu formalisieren, ist es üblich, davon auszugehen, dass das Bandalphabet ein sogenanntes *Blanksymbol* enthält, für das wir $\square \in X$ schreiben. Bandfelder, die „mit $\square$ beschriftet“ sind, wollen wir als „leer“ ansehen; und so stellen wir sie dann gelegentlich auch dar, oder lassen sie ganz weg. Jedenfalls in dieser Vorlesung (und in vielen anderen auch) sind alle tatsächlich vorkommenden Bandbeschriftungen von der Art, dass nur endlich viele Felder nicht mit $\square$ beschriftet sind.

Blanksymbol

16.2.1 *Berechnungen*

Wenn $c = (z, b, p)$ die aktuelle Konfiguration einer Turingmaschine T ist, dann kann es sein, dass sie einen *Schritt* durchführen kann. Das geht genau dann, wenn für das Paar $(z, b(p))$ aus aktuellem Zustand und aktuell gelesenem Bandsymbol die Funktionen f, g und m definiert sind. Gegebenenfalls führt das dann dazu, dass T in die Konfiguration $c' = (z', b', p')$ übergeht, die wie folgt definiert ist:

Schritt einer Turingmaschine

- $z' = f(z, b(p))$
- $\forall i \in \mathbb{Z} : b'(i) = \begin{cases} b(i) & \text{falls } i \neq p \\ g(z, b(p)) & \text{falls } i = p \end{cases}$
- $p' = p + m(z, b(p))$

Wir schreiben $c' = \Delta_1(c)$. Bezeichnet $\mathcal{C}_T$ die Menge aller Konfigurationen einer Turingmaschine T, dann ist das also die partielle Abbildung $\Delta_1 : \mathcal{C}_T \dashrightarrow \mathcal{C}_T$, die als Funktionswert $\Delta_1(c)$

gegebenenfalls die ausgehend von c nach einem Schritt erreichte Konfiguration bezeichnet.

Falls für eine Konfiguration c die Nachfolgekonfiguration $\Delta_1(c)$ nicht definiert ist, heißt c auch eine *Endkonfiguration* und man sagt, die Turingmaschine habe *gehalten*.

Endkonfiguration

Halten

Die Turingmaschine aus Abbildung 16.2 wollen wir BB3 nennen. Wenn BB3 im Anfangszustand A auf einem vollständig leeren Band gestartet wird, dann macht sie wegen

- $f(A, \square) = B$,
- $g(A, \square) = 1$ und
- $m(A, \square) = R$

folgenden Schritt:

			A				
□	□	□	□	□	□	□	□
				B			
□	□	□	1	□	□	□	□

Dabei haben wir den Zustand der Turingmaschine jeweils über dem gerade besuchten Bandfeld notiert. In der entstandenen Konfiguration kann BB3 einen weiteren Schritt machen, und noch einen und noch einen .... Es ergibt sich folgender Ablauf.

			A				
□	□	□	□	□	□	□	□
				B			
□	□	□	1	□	□	□	□
					C		
□	□	□	1	□	□	□	□
				C			
□	□	□	1	□	1	□	□
			C				
□	□	□	1	1	1	□	□
		A					
□	□	□	1	1	1	□	□
			B				
□	□	1	1	1	1	□	□
				B			
□	□	1	1	1	1	□	□
					B		
□	□	1	1	1	1	□	□
						B	
□	□	1	1	1	1	□	□
							C
□	□	1	1	1	1	□	□

						C	
□	□	1	1	1	1	□	1
					C		
□	□	1	1	1	1	1	1
				A			
□	□	1	1	1	1	1	1
					H		
□	□	1	1	1	1	1	1

In Zustand H ist kein Schritt mehr möglich; es ist eine Endkonfiguration erreicht und BB3 hält.

Eine *endliche Berechnung* ist eine endliche Folge von Konfigurationen $(c_0, c_1, c_2, \dots, c_t)$ mit der Eigenschaft, dass für alle $0 < i \leqslant t$ gilt: $c_i = \Delta_1(c_{i-1})$. Eine Berechnung ist *haltend*, wenn es eine endliche Berechung ist und ihre letzte Konfiguration eine Endkonfiguration ist.

endliche Berechnung

haltende Berechnung

Eine *unendliche Berechnung* ist eine unendliche Folge von Konfigurationen $(c_0, c_1, c_2, \dots)$ mit der Eigenschaft, dass für alle $0 < i$ gilt: $c_i = \Delta_1(c_{i-1})$. Eine unendliche Berechnung heißt auch *nicht haltend*.

unendliche Berechnung

nicht haltende Berechung

Eine nicht haltende Berechungen würden wir zum Beispiel bekommen, wenn wir BB3 dahingehend abändern, dass $f(A, \square) = A$ und $g(A, \square) = \square$ ist. Wenn man dann BB3 auf dem vollständig leeren Band startet, dann bewegt sie ihren Kopf immer weiter nach rechts, lässt das Band leer und bleibt immer im Zustand A.

Analog zu Δ_1 liefere generell für $t \in \mathbb{N}_0$ die Abbildung Δ_t als Funktionswert $\Delta_t(c)$ gegebenenfalls die ausgehend von c nach t Schritten erreichte Konfiguration. Also

$$\begin{aligned} \Delta_0 &= \mathrm{Id} \\ \Delta_{t+1} &= \Delta_1 \circ \Delta_t \end{aligned}$$

Zu jeder Konfiguration c gibt es genau eine Berechnung, die mit c startet, und wenn diese Berechnung hält, dann ist der Zeitpunkt zu dem das geschieht natürlich auch eindeutig. Wir schreiben Δ_* für die partielle Abbildung $\mathcal{C}_T \dashrightarrow \mathcal{C}_T$ mit

$$\Delta_*(c) = \begin{cases} \Delta_t(c) & \text{falls } \Delta_t(c) \text{ definiert und} \\ & \text{Endkonfiguration ist} \\ \text{undefiniert} & \text{falls } \Delta_t(c) \text{ für alle } t \in \mathbb{N}_0 \text{ definiert ist} \end{cases}$$

16.2.2 *Eingaben für Turingmaschinen*

Informell (und etwas ungenau) gesprochen werden Turingmaschinen für „zwei Arten von Aufgaben“ eingesetzt: Zum einen wie endliche Akzeptoren zur Entscheidung der Frage, ob ein Eingabewort zu einer bestimmten formalen Sprache gehört. Man spricht in diesem Zusammenhang auch von *Entscheidungsproblemen*. Zum anderen betrachtet man allgemeiner den Fall der

Entscheidungsproblem

„Berechung von Funktionen", bei denen der Funktionswert aus einem größeren Bereich als nur $\{0,1\}$ kommt.

In beiden Fällen muss aber jedenfalls der Turingmaschine die Eingabe zur Verfügung gestellt werden. Dazu fordern wir, dass stets ein *Eingabealphabet* $A \subset X \setminus \{\square\}$ spezifiziert ist. (Das Blanksymbol gehört also nie zum Eingabealphabet.) Und die Eingabe eines Wortes $w \in A^*$ wird bewerkstelligt, indem die Turingmaschine im Anfangszustand z_0 mit dem Kopf auf Feld 0 gestartet wird mit der Bandbeschriftung

Eingabealphabet

$$b_w : \mathbb{Z} \to X$$
$$b_w(i) = \begin{cases} \square & \text{falls } i < 0 \vee i \geqslant |w| \\ w(i) & \text{falls } 0 \leqslant i \wedge i < |w| \end{cases}$$

Für die so definierte *zur Eingabe w gehörende Anfangskonfiguration* schreiben wir auch $c_0(w)$.

zu w gehörende Anfangskonfiguration

Interessiert man sich z. B. für die Berechnung von Funktionen der Form $f : \mathbb{N}_0 \to \mathbb{N}_0$, dann wählt man üblicherweise die naheliegende Binärdarstellung des Argumentes x für f als Eingabewort für die Turingmaschine und vereinbart z. B., dass in der Endkonfiguration das Band wieder vollständig leer ist bis auf die Binärdarstellung von $f(x)$. Die ganzen technischen Details hierzu wollen wir uns sparen. Sie mögen aber bitte glauben, dass man das alles ordentlich definieren kann.

16.2.3 *Ergebnisse von Turingmaschinen*

Man wählt verschiedene Arten, wie eine Turingmaschine ein Ergebnis „mitteilt", wenn sie hält. Im Falle von Entscheidungsproblemen wollen wir wie bei endlichen Akzeptoren davon ausgehen, dass eine Teilmenge $F \subset Z$ von *akzpetierenden Zuständen* definiert ist. Ein Wort w gilt als *akzeptiert*, wenn die Turingmaschine für Eingabe w hält und der Zustand der Endkonfiguration $\Delta_*(c_0(w))$ ein akzepierender ist. Die Menge aller von einer Turingmaschine T akzeptierten Wörter heißt wieder *akzeptierte formale Sprache* $L(T)$. Wir sprechen in diesem Zusammenhang gelegentlich auch von *Turingmaschinenakzeptoren*.

akzeptierender Zustand

akzeptiertes Wort

akzeptierte formale Sprache

Turingmaschinenakzeptor

Wenn ein Wort w von einer Turingmaschine *nicht* akzeptiert wird, dann gibt es dafür zwei mögliche Ursachen:

- Die Turingmaschine hält für Eingabe w in einem nicht akzeptierenden Zustand.
- Die Turingmaschine hält für Eingabe w nicht.

Im ersten Fall bekommt man sozusagen die Mitteilung „Ich bin fertig und lehne die Eingabe ab." Im zweiten Fall weiß man nach jedem Schritt nur, dass die Turingmaschine noch arbeitet. Ob sie irgendwann anhält, und ob sie die Eingabe dann akzeptiert oder ablehnt, ist im allgemeinen unbekannt. Eine formale Sprache, die von einer Turingmaschine akzeptiert werden kann, heißt auf *aufzählbare Sprache*.

aufzählbare Sprache

Wenn es eine Turingmaschine T gibt, die L akzeptiert und für jede Eingabe hält, dann sagt man auch, dass T die Sprache L

entscheide und dass L *entscheidbar* ist. Dass das eine echt stärkere Eigenschaft ist als Aufzählbarkeit werden wir in Abschnitt 16.5 ansprechen.

entscheiden
entscheidbare Sprache

Als Beispiel betrachten wir die Aufgabe, für jedes Eingabewort $w \in L = \{a, b\}^*$ festzustellen, ob es ein Palindrom ist oder nicht. Um das zu belegen, muss man einen Turingmaschinenakzeptor finden, der genau L entscheidet. In Abbildung 16.3 ist eine solche Turingmaschine angegeben. Ihr Anfangszustand ist l und einziger akzeptierender Zustand ist f_+. Der Algorithmus beruht auf der Idee, dass ein Wort genau dann Palindrom ist, wenn erstes und letztes Symbol übereinstimmen und das Teilwort dazwischen auch ein Palindrom ist.

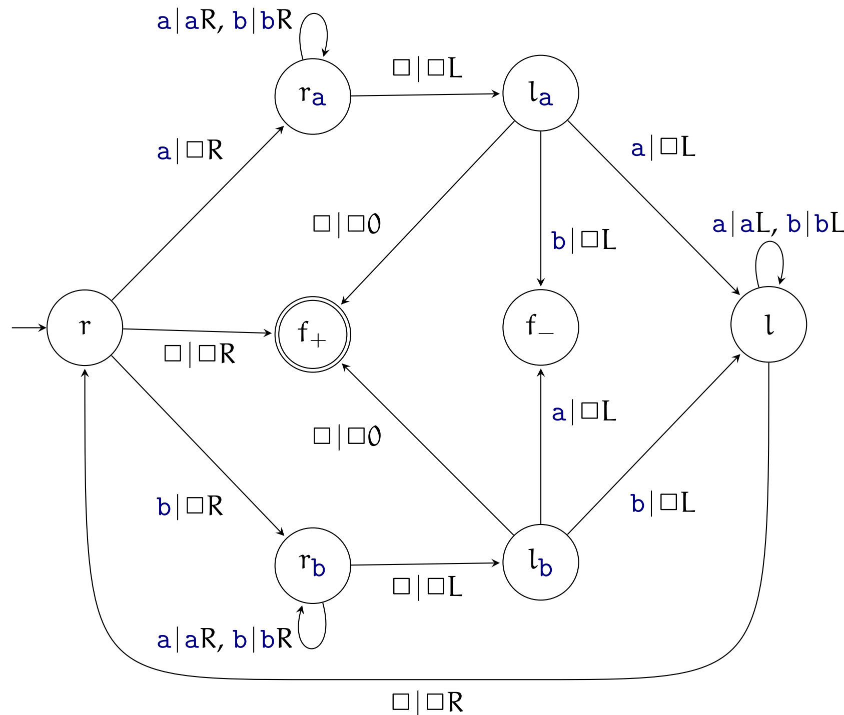


Abbildung 16.3: Eine Turingmaschine zur Palindromerkennung; f_+ sei der einzige akzeptierende Zustand

Zur Erläuterung der Arbeitsweise der Turingmaschine ist in Abbildung 16.4 beispielhaft die Berechnung für Eingabe abba angegeben. Man kann sich klar machen (tun Sie das auch), dass die Turingmaschine alle Palindrome und nur die akzeptiert und für jede Eingabe hält. Sie entscheidet die Sprache der Palindrome also sogar.

16.3 Berechnungskomplexität

Wir beginnen mit einem wichtigen Hinweis: Der Einfachheit halber wollen wir in diesem Abschnitt und im nachfolgenden Abschnitt 16.4 davon ausgehen, dass wir ausschließlich mit Turingmaschinen zu tun haben, die für jede Eingabe halten. Warum das „in Ordnung“ ist, erfahren Sie vielleicht einmal in einer Vorlesung über Komplexitätstheorie.

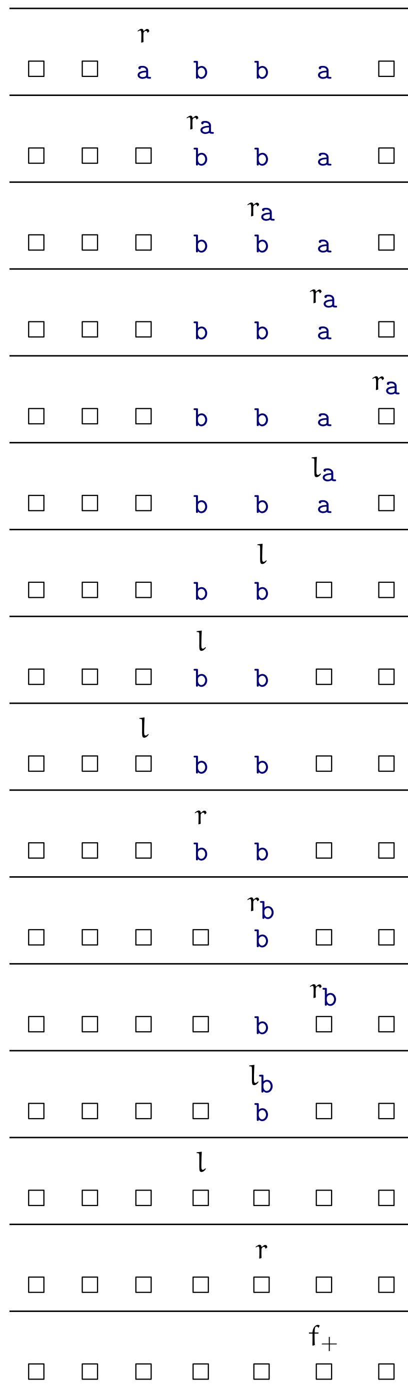


Abbildung 16.4: Akzeptierende Berechung der Turingmaschine aus Abbildung 16.3 für Eingabe abba

In Abschnitt 16.5 werden wir dann aber wieder gerade von dem allgemeinen Fall ausgehen, dass eine Turingmaschine für manche Eingaben *nicht* hält. Warum das wichtig ist, werden Sie hoffentlich schon in Abschnitt 16.5 verstehen. Viel mehr zu diesem Thema werden Sie vielleicht einmal in einer Vorlesung über Berechenbarkeit bzw. Rekursionstheorie hören.

16.3.1 *Komplexitätsmaße*

Bei Turingmaschinen kann man leicht zwei sogenannte *Komplexitätsmaße* definieren, die Rechenzeit und Speicherplatzbedarf charakterisieren.

Komplexitätsmaß

Für die Beurteilung des Zeitbedarfs definiert man zwei Funktionen $\text{time}_T : A^+ \to \mathbb{N}_+$ und $\text{Time}_T : \mathbb{N}_+ \to \mathbb{N}_+$ wie folgt:

$$\begin{aligned} \text{time}_T(w) &= \text{dasjenige } t \text{ mit } \Delta_t(c_0(w)) = \Delta_*(c_0(w)) \\ \text{Time}_T(n) &= \max\{\text{time}(w) \mid w \in A^n\} \end{aligned}$$

Üblicherweise heißt die Abbildung Time_T die *Zeitkomplexität* der Turingmaschine T. Man beschränkt sich also darauf, den Zeitbedarf in Abhängigkeit von der Länge der Eingabe (und nicht für jede Eingabe einzeln) anzugeben (nach oben beschränkt). Man sagt, dass die Zeitkomplexität einer Turingmaschine *polynomiell* ist, wenn ein Polynom $p(n)$ existiert, so dass $\text{Time}_T(n) \in O(p(n))$.

Zeitkomplexität

pollynomielle Zeitkomplexität

Welche Zeitkomplexität hat unsere Turingmaschine zur Palindromerkennung? Für eine Eingabe der Länge $n \geqslant 2$ muss sie schlimmstenfalls

- erstes und letztes Symbol miteinander vergleichen, stellt dabei fest, dass sie übereinstimmen, und muss dann
- für das Teilwort der Länge $n-2$ ohne die Randsymbole wieder einen Palindromtest machen.

Der erste Teil erfordert $2n-1$ Schritte. Für den Zeitbedarf $\text{Time}(n)$ gilt also jedenfalls:

$$\text{Time}(n) \leqslant 2n + 1 + \text{Time}(n-2)$$

Der Zeitaufwand für Wörter der Längen 0 und 1 ist jedenfalls durch eine Konstante c beschränkt. Eine kurze Überlegung zeigt, dass daher die Zeitkomplexität $\text{Time}(n) \in O(n^2)$, also polynomiell ist.

Für die Beurteilung des Speicherplatzbedarfs definiert man zwei Funktionen $\text{space}_T(w) : A^+ \to \mathbb{N}_+$ $\text{Space}_T(n) : \mathbb{N}_+ \to \mathbb{N}_+$ wie folgt:

$$\begin{aligned} \text{space}_T(w) &= \text{die Anzahl der Felder, die während der} \\ &\qquad \text{Berechnung für Eingabe } w \text{ benötigt werden} \\ \text{Space}_T(n) &= \max\{\text{space}(w) \mid w \in A^n\} \end{aligned}$$

Üblicherweise heißt die Abbildung Space_T die *Raumkomplexität* oder *Platzkomplexität* der Turingmaschine T. Dabei gelte ein Feld

Raumkomplexität
Platzkomplexität

als „benötigt“, wenn es zu Beginn ein Eingabesymbol enthält oder irgendwann vom Kopf der Turingmaschine besucht wird. Man sagt, dass die Raumkomplexität einer Turingmaschine *polynomiell* ist, wenn ein Polynom $p(n)$ existiert, so dass $\text{Space}_T(n) \in O(p(n))$.

polynomielle Raumkomplexität

Unsere Beispielmaschine zur Palindromerkennung hat Platzbedarf $\text{Space}(n) = n + 2 \in \Theta(n)$, weil außer den n Feldern mit den Eingabesymbolen nur noch jeweils ein weiteres Feld links bzw. rechts davon besucht werden.

Welche Zusammenhänge bestehen zwischen der Zeit- und der Raumkomplexität einer Turingmaschine? Wenn eine Turingmaschine für eine Eingabe w genau $\text{time}(w)$ viele Schritte macht, dann kann sie nicht mehr als $1 + \text{time}(w)$ verschiedene Felder besuchen. Folglich ist sicher immer

$$\text{space}(w) \leqslant \max(|w|, 1 + \text{time}(w)) .$$

Also hat eine Turingmaschine mit polynomieller Laufzeit auch nur polynomiellen Platzbedarf.

Umgekehrt kann man aber auf k Feldern des Bandes $|X|^k$ verschieden Inschriften speichern. Daraus folgt, dass es sehr wohl Turingmaschinen gibt, die zwar polynomielle Raumkomplexität, aber exponentielle Zeitkomplexität haben.

16.3.2 *Komplexitätsklassen*

Eine *Komplexitätsklasse* ist eine Menge von Problemen. Wir beschränken uns im folgenden wieder auf Entscheidungsprobleme, also formale Sprachen. Charakterisiert werden Komplexitätsklasse durch Beschränkung der zur Verfügung stehen Ressourcen, also Schranken für Zeitkomplexität oder/und Raumkomplexität (oder andere Maße). Zum Beispiel könnte man die Menge aller Entscheidungsprobleme betrachten, die von Turingmaschinen entschieden werden können, bei denen gleichzeitig die Zeitkomplexität in $O(n^2)$ und die Raumkomplexität in $O(n^{3/2} \log n)$ ist, wobei hier n wieder für die Größe der Probleminstanz, also die Länge des Eingabewortes, steht.

Komplexitätsklasse

Es hat sich herausgestellt, dass unter anderem die beiden folgenden Komplexitätsklassen interessant sind:

- **P** ist die Menge aller Entscheidungsprobleme, die von Turingmaschinen entschieden werden können, deren Zeitkomplexität polynomiell ist.
- **PSPACE** ist die Menge aller Entscheidungsprobleme, die von Turingmaschinen entschieden werden können, deren Raumkomplexität polynomiell ist.

Zu **P** gehört zum Beispiel das Problem der Palindromerkennung. Denn wie wir uns überlegt haben, benötigt die Turingmaschine aus Abbildung 16.3 für Wörter der Länge n stets $O(n^2)$ Schritte, um festzustellen, ob w Palindrom ist.

Ein Beispiel eines Entscheidungsproblemes aus **PSPACE** haben wir schon in Einheit 15 erwähnt, nämlich zu entscheiden, ob

zwei reguläre Ausdrücke die gleiche formale Sprache beschreiben. In diesem Fall besteht also jede Probleminstanz aus zwei regulären Ausdrücken. Als das (jeweils eine) Eingabewort für die Turingmaschine würde man in diesem Fall die Konkatenation der beiden regulären Ausdrücke mit einem dazwischen gesetzten Trennsymbol, das sich von allen anderen Symbolen unterscheidet, wählen.

Welche Zusammenhänge bestehen zwischen **P** und **PSPACE**? Wir haben schon erwähnt, dass eine Turingmaschine mit polynomieller Laufzeit auch nur polynomiell viele Felder besuchen kann. Also ist eine Turingmaschine, die belegt, dass ein Problem in **P** liegt, auch gleich ein Beleg dafür, dass das Problem in **PSPACE** liegt. Folglich ist

$$\mathbf{P} \subseteq \mathbf{PSPACE} \,.$$

Und wie ist es umgekehrt? Wir haben auch erwähnt, dass eine Turingmaschine mit polynomiellem Platzbedarf exponentiell viele Schritte machen kann. Und solche Turingmaschinen gibt es auch. Aber Vorsicht: Das heißt *nicht*, dass **PSPACE** eine echte Obermenge von **P** ist. Bei diesen Mengen geht es um Probleme, nicht um Turingmaschinen. Es könnte ja sein, dass es zu jeder Turingmaschine mit polynomiellem Platzbedarf auch dann, wenn sie exponentielle Laufzeit hat, eine andere Turingmaschine mit nur polynomiellem Zeitbedarf gibt, die genau das gleiche Problem entscheidet. Ob das so ist, weiß man nicht. Es ist eines der großen offenen wissenschaftlichen Probleme, herauszufinden, ob $\mathbf{P} = \mathbf{PSPACE}$ ist oder $\mathbf{P} \neq \mathbf{PSPACE}$.

16.4 schwere probleme

Es seien L_1 und L_2 zwei formale Sprachen über dem gleichen Alphabet A (oder mit anderen Worten: Entscheidungsprobleme). Man sagt, dass L_1 *auf* L_2 *reduzierbar* ist, wenn es eine in Polynomialzeit berechenbare Funktion $f : A^* \to A^*$ gibt mit der Eigenschaft: $w \in L_1 \iff f(w) \in L_2$. (Eine solche Funktion kann auch wieder nur polynomiell viel Platz benötigen.) Wenn man das hat und z. B. $L_2 \in \mathbf{P}$ ist, dann kann man auch L_1 in Polynomialzeit entscheiden: Für jede Eingabe w

L_1 *auf* L_2 *reduzierbar*

- berechnet man zuerst $f(w)$ und
- überprüft dann ob $f(w) \in L_2$ ist.

Beides geht in Polynomialzeit, also ist dann auch $L_1 \in \mathbf{P}$.

Und wenn $L_2 \in \mathbf{PSPACE}$ ist, dann folgt aus der gleichen Konstruktion, dass auch $L_1 \in \mathbf{PSPACE}$ ist.

Ein Problem C heißt **PSPACE**-*hart* oder **PSPACE**-*schwer* wenn es für jedes Problem $L \in \mathbf{PSPACE}$ eine Polynomialzeitreduktion auf C gibt. Ein Problem heißt **PSPACE**-*vollständig*, falls es **PSPACE**-hart ist und selbst in **PSPACE** liegt. **PSPACE**-vollständige Probleme gehören also sozusagen „zu den schwersten in **PSPACE**“.

PSPACE-*hart*
PSPACE-*schwer*
PSPACE-*vollständig*

Solche Probleme gibt es. Man kennt eine ganze Reihe von ihnen, z. B. die schon erwähnte Überprüfung, ob zwei reguläre Ausdrücke die gleiche formale Sprache beschreiben.

Eine Folge der eben gemachten Überlegungen ist: Wenn man nur für ein einziges **PSPACE**-vollständiges Problem einen Polynomialzeitalgorithmus findet, dann kann man *alle* Probleme aus **PSPACE** in Polynomialzeit entscheiden. Der etwas kuriose Stand der Wissenschaft **PSPACE**-vollständige Probleme betreffend ist: Man kennt für kein einziges einen Polynomialzeit-Algorithmus, man kann aber bislang auch nicht beweisen, dass keine existieren.

Damit Sie aus den vorangegangenen Abschnitten nicht falsche Schlüsse ziehen, wollen wir zum Abschluss noch folgendes mitteilen: Man kann beweisen, dass es noch viel schwierigere Probleme gibt. Z. B. kennt man Probleme, für die jede Turingmaschine, die eines von ihnen löst, Laufzeit $2^{2^{2^n}}$ hat. Und Analoges gilt für höhere „Türme“ von Exponenten.

16.5 Unentscheidbare Probleme

In gewisser Weise noch schlimmer als Probleme, die exorbitanten Ressourcenaufwand zur Lösung erfordern, sind Probleme, die man algorithmisch, also z. B. mit Turingmaschinen oder Javaprogrammen, überhaupt nicht lösen kann.

In diesem Abschnitt wollen wir zumindest andeutungsweise sehen, dass es solche Probleme tatsächlich gibt.

16.5.1 *Codierungen von Turingmaschinen*

Zunächst soll eine Möglichkeit beschrieben werden, wie man jede Turingmaschine durch ein Wort über dem festen Alphabet $A = \{$`[`, `]`, `0`, `1`$\}$ beschreiben kann. Natürlich kann man diese Symbole durch Wörter über $\{$`0`, `1`$\}$ codieren und so die Alphabetgröße auf einfache Weise noch reduzieren.

Eine Turingmaschine $T = (Z, z_0, X, \square, f, g, m)$ kann man zum Beispiel wie folgt codieren.

- Die Zustände von T werden ab 0 durchnummeriert.
- Der Anfangszustand bekommt Nummer 0.
- Alle Zustände werden durch gleich lange Binärdarstellungen ihrer Nummern, umgeben von einfachen eckigen Klammern, repräsentiert.
- Wir schreiben $\text{cod}_Z(z)$ für die Codierung von Zustand z.

- Die Bandsymbole werden ab 0 durchnummeriert.
- Das Blanksymbol bekommt Nummer 0.
- Alle Bandsymbole werden durch gleich lange Binärdarstellungen ihrer Nummern, umgeben von einfachen eckigen Klammern, repräsentiert.
- Wir schreiben $\text{cod}_X(x)$ für die Codierung von Bandsymbol x.

- Die möglichen Bewegungsrichtungen des Kopfes werden durch die Wörter [10], [00] und [01] (für -1, 0 und 1) repräsentiert.
- Wir schreiben $\text{cod}_M(r)$ für die Codierung der Bewegungsrichtung r.

- Die partiellen Funktionen f, g und m werden wie folgt codiert:
  - Wenn sie für ein Argumentpaar (z, x) nicht definiert sind, wird das codiert durch das Wort $\text{cod}_{fgm}(z, x) =$ $[\text{cod}_Z(z)\,\text{cod}_X(x)[][][]]$.
  - Wenn sie für ein Argumentpaar (z, x) definiert sind, wird das codiert durch das Wort $\text{cod}_{fgm}(z, x) =$ $[\text{cod}_Z(z)\,\text{cod}_X(x)\,\text{cod}_Z(f(z, x))\,\text{cod}_X(g(z, x))\,\text{cod}_M(m(z, x))]$.
  - Die Codierung der gesamten Funktionen ist die Konkatenation aller $\text{cod}_{fgm}(z, x)$ für alle $z \in Z$ und alle $x \in X$.

- Die gesamte Turingmaschine wird codiert als Konkatenation der Codierung des Zustands mit der größten Nummer, des Bandsymbols mit der größten Nummer und der Codierung der gesamten Funktionen f, g und m.
- Wir schreiben auch T_w für die Turingmaschine mit Codierung w.

Auch ohne dass wir das hier im Detail ausführen, können Sie hoffentlich zumindest glauben, dass man eine Turingmaschine konstruieren kann, die für jedes beliebiges Wort aus A^* feststellt, ob es die Codierung einer Turingmaschine ist. Mehr wird für das Verständnis des folgenden Abschnittes nicht benötigt.

Tatsächlich kann man sogar noch mehr: Es gibt sogenannte *universelle Turingmaschinen*. Eine universelle Turingmaschine U

universelle Turingmaschine

- erhält als Eingabe zwei Argumente, etwa als Wort $[w_1][w_2]$,
- überprüft, ob w_1 Codierung einer Turingmaschine T ist, und
- falls ja, simuliert Schritt für Schritt die Arbeit, die T für Eingabe w_2 durchführen würde,
- und falls T endet, liefert U am Ende als Ergebnis das, was T geliefert hat.

16.5.2 *Das Halteproblem*

Der Kern des Nachweises der Unentscheidbarkeit des Halteproblems ist *Diagonalisierung*. Die Idee geht auf Georg Ferdinand Ludwig Philipp Cantor (1845–1918, siehe z. B. http://www-history.mcs.st-and.ac.uk/Biographies/Cantor.html, 25.1.09), der sie benutzte um zu zeigen, dass die Menge der reellen Zahlen nicht abzählbar unendlich ist. Dazu sei eine „zweidimensionale unendliche Tabelle“ gegeben, deren Zeilen mit Funktionen f_i ($i \in \mathbb{N}_0$) indiziert sind, und die Spalten mit Argumenten x_j ($j \in \mathbb{N}_0$). Eintrag in Zeile i und Spalte j der Tabelle sei gerade der Funktionswert $f_i(x_j)$. Die f_i mögen als Funktionswerte zumindest 0 und 1 annehmen können.

Diagonalisierung

	x_0	x_1	x_2	x_3	x_4	$\cdots$
f_0	$f_0(x_0)$	$f_0(x_1)$	$f_0(x_2)$	$f_0(x_3)$	$f_0(x_4)$	$\cdots$
f_1	$f_1(x_0)$	$f_1(x_1)$	$f_1(x_2)$	$f_1(x_3)$	$f_1(x_4)$	$\cdots$
f_2	$f_2(x_0)$	$f_2(x_1)$	$f_2(x_2)$	$f_2(x_3)$	$f_2(x_4)$	$\cdots$
f_3	$f_3(x_0)$	$f_3(x_1)$	$f_3(x_2)$	$f_3(x_3)$	$f_3(x_4)$	$\cdots$
f_4	$f_4(x_0)$	$f_4(x_1)$	$f_4(x_2)$	$f_4(x_3)$	$f_4(x_4)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Cantors Beobachtung war (im Kern) die folgende: Wenn man die Diagonale der Tabelle nimmt, also die Abbildung d mit $d(x_i) = f_i(x_i)$ und dann *alle Einträge ändert*, also

$$\overline{d}(x_i) = \overline{f_i(x_i)} = \begin{cases} 1 & \text{falls } f_i(x_i) = 0 \\ 0 & \text{sonst} \end{cases}$$

dann erhält man eine Abbildung („Zeile") $\overline{d}$,

	x_0	x_1	x_2	x_3	x_4	$\cdots$
d	$f_0(x_0)$	$f_1(x_1)$	$f_2(x_2)$	$f_3(x_3)$	$f_4(x_4)$	$\cdots$
$\overline{d}$	$\overline{f_0(x_0)}$	$\overline{f_1(x_1)}$	$\overline{f_2(x_2)}$	$\overline{f_3(x_3)}$	$\overline{f_4(x_4)}$	$\cdots$

die sich von jeder Abbildung („Zeile") der gegebenen Tabelle unterscheidet, denn für alle i ist

$$\overline{d}(x_i) = \overline{f_i(x_i)} \neq f_i(x_i) \ .$$

Die Ausnutzung dieser Tatsache ist von Anwendung zu Anwendung (und es gibt in der Informatik mehrere, z. B. in der Komplexitätstheorie) verschieden.

Im folgenden wollen wir mit Hilfe dieser Idee nun beweisen, dass das Halteproblem unentscheidbar ist. Es ist keine Beschränkung der Allgemeinheit, wenn wir uns auf ein Alphabet A festlegen, über dem wir bequem Codierungen von Turingmaschinen aufschreiben können. Statt „Turingmaschine T hält für Eingabe w" sagen wir im folgenden kürzer „$T(w)$ hält".

Das *Halteproblem* ist die formale Sprache

Halteproblem

$$H = \{w \in A^* \mid w \text{ ist eine TM-Codierung und } T_w(w) \text{ hält.}\}$$

Wir hatten weiter vorne erwähnt, dass es kein Problem darstellt, für ein Wort festzustellen, ob es z. B. gemäß der dort beschriebenen Codierung eine Turingmaschine beschreibt. Das wesentliche Problem beim Halteproblem ist also tatsächlich das Halten.

16.1 Satz. *Das Halteproblem ist unentscheidbar, d. h. es gibt keine Turingmaschine, die H entscheidet.*

16.2 Beweis. Wir benutzen (eine Variante der) Diagonalisierung. In der obigen großen Tabelle seien nun die x_i alle Codierungen von Turingmaschinen in irgendeiner Reihenfolge. Und f_i sei die Funktion, die von der Turingmaschine, deren Codierung x_i ist, also T_{x_i}, berechnet wird.

Da wir es mit Turingmaschinen zu tun haben, werden manche der $f_i(x_j)$ nicht definiert sein, da T_{x_i} für Eingabe x_j nicht hält. Da

die x_i *alle* Codierungen von Turingmaschinen sein sollen, ist in der Tabelle für *jede* Turingmaschine eine Zeile vorhanden.

Nun nehmen wir an, dass es doch eine Turingmaschine T_h gäbe, die das Halteproblem entscheidet, d. h. *für jede Eingabe* x_i *hält* und als Ergebnis mitteilt, ob T_{x_i} für Eingabe x_i hält.

Wir führen diese Annahme nun zu einem Widerspruch, indem wir zeigen: Es gibt eine Art „verdorbene Diagonale" $\overline{d}$ ähnlich wie oben mit den beiden folgenden sich widersprechenden Eigenschaften.

- Einerseits unterscheidet sich $\overline{d}$ von jeder Zeile der Tabelle, also von jeder von einer Turingmaschine berechneten Funktion.
- Andererseits kann auch $\overline{d}$ von einer Turingmaschine berechnet werden.

Und das ist ganz einfach: Wenn die Turingmaschine T_h existiert, dann kann man auch den folgenden Algorithmus in einer Turingmaschine $T_{\overline{d}}$ realisieren:

- Für eine Eingabe x_i berechnet $T_{\overline{d}}$ zunächst, welches Ergebnis T_h für diese Eingabe liefern würde.
- Dann arbeitet $T_{\overline{d}}$ wie folgt weiter:
  - Wenn T_h mitteilt, dass $T_{x_i}(x_i)$ hält,
    dann geht $T_{\overline{d}}$ in eine Endlosschleife.
  - Wenn T_h mitteilt, dass $T_{x_i}(x_i)$ nicht hält,
    dann hält $T_{\overline{d}}$ (und liefert irgendein Ergebnis, etwa 0).
- Andere Möglichkeiten gibt es nicht, und in beiden Fällen ist das Verhalten von $T_{\overline{d}}$ für Eingabe x_i anders als das von T_{x_i} für die gleiche Eingabe.

Also: Wenn Turingmaschine T_h existiert, dann existiert auch Turingmaschine $T_{\overline{d}}$, aber jede Turingmaschine (T_{x_i}) verhält sich für mindestens eine Eingabe (x_i) anders als $T_{\overline{d}}$.

Das ist ein Widerspruch. Folglich war die Annahme, dass es die Turingmaschine T_h gibt, die H entscheidet, falsch. ■

16.5.3 *Die Busy-Beaver-Funktion*

In Abschnitt 16.2 hatten wir BB3 als erstes Beispiel einer Turingmaschine gesehen. Diese Turingmaschine hat folgende Eigenschaften:

- Bandalphabet ist $X = \{\square, 1\}$.
- Die Turingmaschine hat $3 + 1$ Zustände, wobei
  - in 3 Zuständen für jedes Bandsymbol der nächste Schritt definiert ist,
  - einer dieser 3 Zustände der Anfangszustand ist und
  - in einem weiteren Zustand für kein Bandsymbol der nächste Schritt definiert ist („Haltezustand").
- Wenn man die Turingmaschine auf dem leeren Band startet, dann hält sie nach endlich vielen Schritten.

Wir interessieren uns nun allgemein für Turingmaschinen mit den Eigenschaften:

- Bandalphabet ist $X = \{\square, 1\}$.
- Die Turingmaschine hat $n + 1$ Zustände, wobei
  - in n Zuständen für jedes Bandsymbol der nächste Schritt definiert ist,
  - einer dieser n Zustände der Anfangszustand ist und
  - in einem weiteren Zustand für kein Bandsymbol der nächste Schritt definiert ist („Haltezustand“).
- Wenn man die Turingmaschine auf dem leeren Band startet, dann hält sie nach endlich vielen Schritten.

Solche Turingmaschinen wollen wir der Einfachheit halber *Bibermaschinen* nennen. Genauer wollen wir von einer n-Bibermaschine reden, wenn sie, einschließlich des Haltezustands $n + 1$ Zustände hat. Wann immer es im folgenden explizit oder implizit um Berechnungen von Bibermaschinen geht, ist immer die gemeint, bei der sie auf dem vollständig leeren Band startet. Bei BB3 haben wir gesehen, dass sie 6 Einsen auf dem Band erzeugt.

Bibermaschine

Abbildung 16.5: Europäischer Biber (castor fiber), Bildquelle: http://upload.wikimedia.org/wikipedia/commons/d/d4/Beaver_2.jpg

Eine Bibermaschine heiße *fleißiger Biber*, wenn sie am Ende die maximale Anzahl Einsen auf dem Band hinterlässt unter allen Bibermaschinen mit gleicher Zustandszahl.

fleißiger Biber

Die *Busy-Beaver-Funktion* ist wie folgt definiert:

Busy-Beaver-Funktion

$$\begin{aligned} bb &: \mathbb{N}_+ \to \mathbb{N}_+ \\ bb(n) &= \text{die Anzahl von Einsen, die ein fleißiger Biber mit} \\ &\qquad n+1 \text{ Zuständen am Ende auf dem Band hinterlässt} \end{aligned}$$

Diese Funktion wird auch *Radó-Funktion* genannt nach dem ungarischen Mathematiker Tibor Radó, der sich als erster mit dieser Funktion beschäftigte. Statt $bb(n)$ wird manchmal auch $\Sigma(n)$ geschrieben.

Radó-Funktion

Da BB3 am Ende 6 Einsen auf dem Band hinterlässt, ist also jedenfalls $bb(3) \geqslant 6$. Radó fand heraus, dass sogar $bb(3) = 6$ ist. Die Turingmaschine BB3 ist also ein fließiger Biber.

Brady hat 1983 gezeigt, dass $bb(4) = 13$ ist. Ein entsprechender fleißiger Biber ist

	A	B	C	D	H
$\square$	1, R, B	1, L, A	1, R, H	1, R, D	
1	1, L, B	$\square$, L, C	1, L, D	$\square$, R, A	

H. Marxen (http://www.drb.insel.de/~heiner/BB/, 24.1.09) und J. Buntrock haben 1990 eine 5-Bibermaschine gefunden, die 4098 Einsen produziert und nach 47 176 870 Schritten hält. Also ist $bb(5) \geqslant 4098$. Man weiß nicht, ob es eine 5-Bibermaschine gibt, die mehr als 4098 Einsen schreibt.

Im Dezember 2007 haben Terry and Shawn Ligocki eine 6-Bibermaschine gefunden, die mehr als $4.6 \cdot 10^{1439}$ Einsen schreibt und nach mehr als $2.5 \cdot 10^{2879}$ Schritten hält. Also ist $bb(6) \geqslant 4.6 \cdot 10^{1439}$. Nein, hier liegt kein Schreibfehler vor!

Man hat den Eindruck, dass die Funktion bb sehr schnell wachsend ist. Das stimmt. Ohne den Beweis hier wiedergeben zu wollen (Interessierte finden ihn z. B. in einem Aufsatz von J. Shallit (http://grail.cba.csuohio.edu/~somos/beaver.ps, 24.1.09) teilen wir noch den folgenden Satz mit. In ihm und dem unmittelbaren Korollar wird von der Berechenbarkeit von Funktionen $\mathbb{N}_+ \to \mathbb{N}_+$ gesprochen. Damit ist gemeint, dass es eine Turingmaschine gibt, die

- als Eingabe das Argument (zum Beispiel) in binärer Darstellung auf einem ansonsten leeren Band erhält, und
- als Ausgabe den Funktionswert (zum Beispiel) in binärer Darstellung auf einem ansonsten leeren Band liefert.

16.3 Satz. *Für jede berechenbare Funktion* $f : \mathbb{N}_+ \to \mathbb{N}_+$ *gibt es ein* n_0, *so dass für alle* $n \geqslant n_0$ *gilt:* $bb(n) > f(n)$.

16.4 Korollar. Die Busy-Beaver-Funktion $bb(n)$ ist nicht berechenbar.

16.6 Ausblick

Sie werden an anderer Stelle lernen, dass es eine weitere wichtige Komplexitätsklasse gibt, die **NP** heißt. Man weiß, dass $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$ gilt, aber für keine der beiden Inklusionen weiß man, ob sie echt ist. So wie für **PSPACE** gibt es auch für **NP** vollständige Probleme. **NP**-vollständige Probleme spielen an noch viel mehr Stellen (auch in der Praxis) eine wichtige Rolle als **PSPACE**-vollständige. Und wieder ist es so, dass alle bekannten Algorithmen exponentielle Laufzeit haben, aber man nicht beweisen kann, dass das so sein muss.

Literatur

Turing, A. M. (1936). "On computable numbers, with an application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society*. 2. Ser. 42, S. 230–265.
Die Pdf-Version einer HTML-Version ist online verfügbar; siehe http://web.comlab.ox.ac.uk/oucl/research/areas/ieg/e-library/sources/tp2-ie.pdf (14.1.09).