Kapitel 1

Wie aus Raum Zeit wird und umgekehrt

Lernziele dieses Kapitels

① Sie wissen, was Alphabete, Wörter und Sprachen sind.

② Sie verstehen das Konzept von Turingmaschinen.

③ Sie können zeigen, dass ein gegebenes Problem von einer Turingmaschine entschieden wird.

④ Sie können mit der O-Notation umgehen.

⑤ Sie verstehen den Unterschied zwischen deterministischen und nichtdeterministischen Maschinen.

⑥ Sie wissen, was Zeit- und Raum-Bedarf (Speicher-Bedarf) bedeutet.

⑦ Sie kennen die Komplexitätsklassen TIME und SPACE.

⑧ Sie kennen die wichtigsten Beziehungen zwischen den Komplexitätsklassen.

⑨ Sie können weitere Beziehungen ableiten und beweisen.

Ziel dieses Kapitels ist es, einen Begriff für die Komplexität eines Computerprogramms zu finden. Betrachten wir das folgende Beispiel, welches diese Sichtweise etwas mehr illustriert. Anton gefällt die Programmiersprache Scheme sehr gut. In seinen Programmen

Was fallen Ihnen noch für Maße ein?

achtet er immer darauf, möglichst wenige Klammern zu verwenden – und das ist bei Scheme wahrlich keine einfache Aufgabe. Sein Komplexitätsmaß ist nun die Anzahl der (öffnenden) Klammern. Dieses Maß drückt jedoch nicht wirklich aus, wie schwierig oder wie *gut* das Programm ist. Es ist lediglich ein schwacher Fingerzeig für die Länge eines Programms. Eine andere Möglichkeit wäre, die Anzahl von sogenannten λ-Ausdrücken zu betrachten. Dies ist ein spezielles Konzept, welches mit der Verwendung von Unterfunktionen zusammenhängt. Aus deren Anzahl kann man ableiten, wie stark verschachtelt das Programm ist. Die beiden erwähnten Maße messen die Schwierigkeit des Computerprogramms, haben jedoch nichts mit der Effizienz deren Ausführung zu tun. Dafür sind eher Maße wie Zeit- und Speicherbedarf sinnvoll.

Betrachten wir nun zuerst den Speicherbedarf eines Programms, also die Anzahl der benutzten Variablen und die Größe der gespeicherten Werte. Abhängig vom System ist die Größendarstellung innerhalb des Systems gleich. Die kleinste Speicherzelle hat konstante Größe und ist daher auf dem gleichen System auch immer gleich groß. Dies ist ein Wert, der jedoch von System zu System variieren kann und von der Repräsentation dieser Werte im Rechner abhängt. Ähnliche Überlegungen kann man auch für den Zeitbedarf anstellen. Wenn das Programm auf Antons Rechner der Firma I schnell läuft, kann es auf Bertas Rechner der Firma A viel langsamer oder auch schneller sein.

Deshalb möchten wir, wie schon im Vorwort erwähnt, eine sinnvolle Definition von Zeit- und Speicher-Bedarf eines Computerprogramms finden. Es soll ein Begriff sein, welcher *maschinenunabhängig* ist. Dies ist ein wichtiger Aspekt, da wir nicht schon bald wieder einen neuen Begriff definieren wollen, wenn es ein Computer-Update oder einen Betriebssystemwechsel gibt. Wichtig ist auch, dass Programme auf unterschiedlichen Rechnern vergleichbar sein sollen.

Hierzu sind also einige Begriffe zu präzisieren:

- Was ist ein algorithmisches Problem?
- Was ist ein Algorithmus oder Computer?
- Was bedeuten Zeit- und Speicherbedarf genau?

1.1. Welche Probleme wollen wir lösen?

Ein *algorithmisches Problem* P wird dadurch charakterisiert, dass man überprüfen möchte, ob Objekte gewisse Eigenschaften besitzen. Diese Eigenschaften können sehr vielfältig sein. Im Folgenden gehen wir davon aus, dass Sie wissen, was Graphen und aussagenlogische Formeln sind. Sollte dies nicht der Fall sein, so werfen Sie einfach einen Blick in den Anhang dieses Buches. Dort werden die Grundlagen zu beiden Themen ausführlich erläutert. Wenden wir uns nun einem ersten solchen Problem P zu. Uns liegt ein ungerichteter Graph $G = (V, E)$ vor und wir fragen uns nun, ob es in diesem Graphen einen vollständigen Teilgraphen (jeder Knoten ist mit jedem anderen hier verbunden) der Größe k gibt. Formal können wir das Problem so beschreiben:

Problemname: CLIQUE
Eingabe: Ungerichteter Graph $G = (V, E)$ und natürliche Zahl k
Frage: Gibt es einen vollständigen Teilgraphen mit k Knoten?

Das Problem heißt CLIQUE, da man üblicherweise vollständige Teilgraphen Cliquen nennt. Eine Spur mathematischer ist die folgende Mengenschreibweise und wird von uns hier im Buch primär verwendet:

$$\text{CLIQUE} = \left\{ \langle G, k \rangle \;\middle|\; \begin{array}{l} G \text{ ist ein ungerichteter Graph, der als Teil-} \\ \text{graph den vollständigen Graphen mit } k \\ \text{Knoten enthält} \end{array} \right\}$$

Hier treten außerdem komische, spitze Klammern $\langle \cdot \rangle$ auf. Momentan können Sie diese erstmal ignorieren. Auf Seite 59 werden wir uns diesen Klammern wieder zuwenden und genauer ihren Sinn erläutern. Wie Sie momentan sehen, ist die Menge CLIQUE nun die Menge jener Paare von ungerichteten Graphen und natürlichen Zahlen k, die einen vollständigen Graphen mit k Knoten enthalten. Das heißt, das Vorhandensein von Cliquen der Größe k ist eben jene oben angesprochene Eigenschaft, welche unser Objekt (der ungerichtete Graph) besitzen soll. Diese Art von algorithmischen Problemen werden auch *Entscheidungsprobleme* genannt, da die zugehörige Frage nur mit *ja* oder *nein* beantwortet wird.

Ein anderes Objekt im obigen Sinne könnte eine aussagenlogische Formel $\varphi = (x \vee y) \wedge \neg x$ sein. Hier fragen wir uns natürlich, ob diese

Formel erfüllbar ist. Das zugehörige Problem lautet (ein letztes Mal in der vorherigen Schreibweise):

Problemname: SAT
Eingabe: Aussagenlogische Formel φ
Frage: Ist φ erfüllbar?

Die Mengenschreibweise lautet hierzu:

$$\text{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ ist aussagenlogische, erfüllbare Formel}\}\,.$$

Hier sind wiederum nur jene Formeln ein Teil der Menge SAT, welche erfüllbar sind. Also wird wieder eine Ja/Nein-Frage in Bezug auf eine gegebene Formel beantwortet.

Nun möchten wir solche algorithmischen Probleme durch Computerprogramme oder Algorithmen lösen. Dazu müssen wir die *Eingaben* dieser Probleme in den Computer beziehungsweise den Algorithmus „eingeben“. In den obigen Beispielen müssen also die Graphen und Formeln so beschrieben oder kodiert werden, dass wir diese über eine Tastatur eingeben können. Unser Programm soll nach der Eingabe entscheiden, ob unser Graph oder unsere Formel den geforderten Eigenschaften des algorithmischen Problems genügt. Dazu wenden wir uns jetzt den dazu notwendigen Beschreibungen oder Kodierungen formal zu. Anfänglich werden wir uns mit *Sprachen* beschäftigen. Informell gesprochen umfasst dieser Term lediglich eine Menge von Elementen mit einer Gemeinsamkeit, wie zum Beispiel die *gerade Wortlänge*. Prinzipiell ist es sogar möglich, jede natürliche Sprache – wie Deutsch oder Englisch – über diesen Terminus zu formalisieren.

Ein *Alphabet* ist eine endliche, nichtleere Menge. Die Elemente eines Alphabets heißen auch *Zeichen* oder *Symbole*.

Beispiel 1. $\{0,1\}, \{\mathrm{a},\mathrm{b},\mathrm{c},\ldots,\mathrm{z},\mathrm{A},\ldots,\mathrm{Z}\}$ ◀

Ist M eine Menge, so bezeichnet $|M|$ die Anzahl der Elemente von M. Sei Σ ein Alphabet. Ein *Wort über* Σ ist eine Folge von Symbolen aus Σ. Ein Wort entsteht also durch *Hintereinanderschreiben* (*Konkatenation*) von Symbolen aus Σ.

Beispiel 2. $w = \mathrm{abbab}$ ist ein Wort über dem Alphabet $\{\mathrm{a},\mathrm{b}\}$. ◀

Ein Spezialfall ist das leere Wort. Wir bezeichnen es mit ε. Die Menge aller Wörter über dem Alphabet Σ bezeichnen wir mit Σ^*.

Beispiel 3. Für $\Sigma = \{a, b\}$ ist $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots\}$. ◀

Eine *Sprache über* Σ ist eine Menge von Wörtern über Σ, also eine Teilmenge von Σ^*.

Beispiel 4. Die Menge der deutschen Wörter ist eine Sprache über

$$\{a, b, c, \ldots, z, A, \ldots, Z, ä, ö, ü, \ldots, ß\}.$$

Die Menge aller C-Programme ist eine Sprache über

$$\{a, b, c, \ldots, z, A, \ldots, Z, \{, \}, (,), [,], +, *, -, /, =, ., \_, \sqcup\}.$$

Beachten Sie, dass hierbei noch keine Aussage über die genaue Syntax getroffen wurde. ◀

Eine wichtige Operation auf Wörtern ist die *Konkatenation* bzw. das Hintereinanderschreiben. Die zugehörige Schreibweise hierfür ist $u \circ v$ oder kurz uv für Konkatenation die der Wörter u und v.

Beispiel 5. Ist $u = ab$ und $v = bab$, so ist $u \circ v = abbab$. ◀

Die *Länge* eines Wortes w ist die Anzahl der Symbole in w. *Schreibweise*: $|w|$

Beispiel 6. $|aba| = 3$, $|\varepsilon| = 0$, $|u \circ v| = |u| + |v|$ ◀

Für ein Wort w und $n \in \mathbb{N}$ ist w^n die Konkatenation

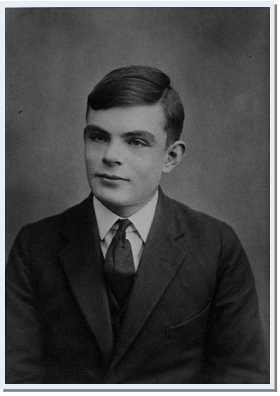
$$w^n = \underbrace{w \circ w \circ \cdots \circ w}_{n\text{mal}}.$$

Wir definieren: $w^0 = \varepsilon$. Es ist $|w^n| = n \cdot |w|$. *Schreibweise:* $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

1.2. Eine Universalmaschine

Nun wissen wir, wie man ein algorithmisches Problem mathematisch darstellt. Im folgenden Schritt möchten wir uns Gedanken machen,

wie wir einen Algorithmus oder Computer modellieren. Stellen wir uns zunächst die Frage, wie ein Mensch ein algorithmisches Problem mit Papier und Stift lösen würde.



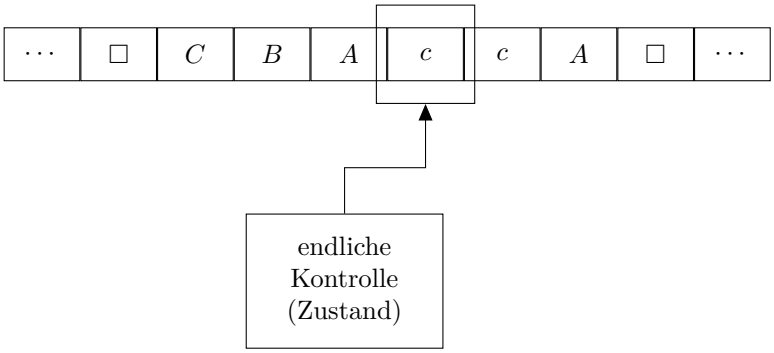
Alan Turing

Der britische Mathematiker und Logiker Alan M. Turing hat dazu im Jahre 1936 folgende Überlegungen angestellt. Er verglich die Vorgehensweise eines Menschen mit der einer Maschine, die in jedem Zeitpunkt ihrer Arbeit endlich viele Situationen unterscheiden oder sich merken kann, anders formuliert: sich in einem von endlich vielen *Zuständen* befindet. Der Mensch/die Maschine hat vor sich ein Blatt Papier liegen, auf dem er/sie arbeitet. Das Papier ist in einzelne Bereiche unterteilt, in die etwas geschrieben werden kann. Wir wollen die Bereiche *Zellen* nennen. In eine Zelle „passt“ also sozusagen genau ein Symbol aus einem festgelegten Zeichenvorrat. Der Einfachheit halber stellen wir uns die Zellen aneinandergereiht vor, wie an einem Band aufgefädelt. Wir sprechen daher nicht von einem Papier, sondern von dem *Band* als Medium zum Zwischenspeichern von Rechenergebnissen. (Man darf bei dem Begriff „Band“, engl.: *tape*, sicher auch an Magnet-Bänder denken, die in früheren Jahren den üblichen Massenspeicher von EDV-Geräten bildeten.) Die Maschine sieht („scannt“) zu einem Zeitpunkt genau eine Bandzelle und kann das dort gespeicherte Symbol lesen. Abhängig von der momentanen Situation/dem momentanen Zustand und dem gelesenen Symbol führt die Maschine nun einen Befehl aus, der daraus besteht, dass sich der Zustand ändern darf, dass das gelesene Symbol durch ein anderes überschrieben oder gelöscht werden darf, und dass die Maschine ihre Aufmerksamkeit nun auf eine benachbarte Zelle richtet, also eine Position nach links oder eine Position nach rechts. Wir setzen voraus, dass der Maschine soviele Bandzellen als Speicher zur Verfügung stehen, wie sie benötigt. Sind alle Zellen beschrieben und werden weitere benötigt, werden sie zur Verfügung gestellt. Man sagt auch: Die Bandgröße ist nicht von vornherein beschränkt, sondern „potentiell unendlich“ – nicht tatsächlich unendlich, da die Maschine zu jedem Zeitpunkt nur eine endliche Anzahl Zellen verwendet.

Eine Turingmaschine (TM) ist ein zunächst sehr abstrakt wirkendes Konzept. Man kann jedoch zeigen, dass dieser Typ von Maschine von seiner Berechungskraft äquivalent ist zu heutigen real existierenden Computern. (Wir werden dies in Kapitel 2 genauer tun.) Der große Vorteil von Turingmaschinen ist, dass sie ein erstaunlich *einfaches* Konzept bilden, das für mathematische Untersuchungen der

Lösbarkeit algorithmischer Probleme und der Effizienz möglicher Lösungsalgorithmen sehr geeignet ist.

Man kann sich eine Turingmaschine also wie folgt vorstellen:



Der obere Streifen steht für das beidseitig (potentiell) unendliche Band, das in Zellen unterteilt ist, in denen Symbole (C, B, A, c im Beispiel) stehen können. Ein besonderes Symbol ist das *Blank* □, welches in leeren Bandzellen (z. B. an den Rändern) steht. Auf den Zellen bewegt sich ein Schreib-Lese-Kopf in drei verschiedenen Modi: links, rechts, keine Bewegung (oder neutral). Das kleine Kästchen unter dem Band (in gewissem Sinne die eigentliche Maschine) enthält das Programm und einen Speicher für den aktuellen Zustand. Abhängig vom aktuellen Zustand und dem gelesenen Zeichen wird dann ein Zeichen geschrieben und der Kopf abhängig von dem gewählten Modus bewegt. Formal gibt man für eine jede solche Maschine M

- die Menge der Zustände Z, die die Maschine hat,
- die Menge der Zeichen Γ, die auf dem Band stehen dürfen,
- die Übergangsfunktion $\delta\colon Z \times \Gamma \to Z \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{N}\}$,
- den Startzustand,
- die Menge der akzeptierenden Zustände A an sowie
- die Menge der verwerfenden Zustände V an.

Konstruieren Sie eine Maschine, die nur akzeptiert, wenn die Eingabelänge gerade ist.

Eine Turingmaschine definieren wir daher als ein Tupel $(Z, \Gamma, \delta, z_0, A, V)$[1].

Man kann dieses Konzept auf eine Maschine mit, beliebig aber endlich vielen, $k \in \mathbb{N}$ Bändern erweitern. Es bleibt jedoch bei einem Zustand und man kann in jedem Schritt k Kopfbewegungen abhängig vom aktuellen Zustand tätigen.

Im Folgenden verwenden wir Ein- und Mehrband-Turingmaschinen und unterscheiden jeweils, ob es ein separates Eingabeband gibt. Auf diesem Eingabeband darf nichts geschrieben werden. Es ist *read only*.

Konfiguration

Definition 1. Sei $M = (Z, \Gamma, \delta, z_0, A, V)$ eine TM, Σ ein Alphabet, $u, v \in \Sigma^*$ Wörter und $z \in Z$ ein Zustand. Wir nennen uzv dann eine *Konfiguration* von M auf uv. Dies bedeutet, dass M sich in Zustand z befindet und seinen Kopf auf dem ersten Zeichen von v stehen hat. Der Bandinhalt links vom Kopf ist u. ◀

Wie lautet die Konfiguration der dargestellten Maschine auf Seite 13 unter der Annahme, der angenommene Zustand ist z?

Im nächsten Schritt formalisieren wir, wie eine Maschine funktioniert, d. h. wie sie von einer Konfiguration in eine andere Konfiguration übergehen kann. Hierbei müssen wir auf die verschiedenen Möglichkeiten eingehen, also ob die Kopfbewegung nach links, rechts oder nicht geschieht.

Konfigurationsübergang

Definition 2. Seien $uzv, u'z'v'$ zwei Konfigurationen der TM $M = (Z, \Gamma, \delta, z_0, A, V)$. Wir sagen uzv geht in einem Schritt in $u'z'v'$ über, in Zeichen $uzv \vdash_M u'z'v'$, falls

$$
\begin{aligned}
&(z', a', R) \in \delta(z, a) && \text{für } v = aw, u' = w'a', w, w' \in \Sigma^*, a, a' \in \Sigma,\\
&(z', a', L) \in \delta(z, a) && \text{für } v = aw, u = u'b, v' = ba'w', w, w' \in \Sigma^*,\\
& && a, b, a' \in \Sigma,\\
&(z', a', N) \in \delta(z, a) && \text{für } v = aw, v' = a'w', w, w' \in \Sigma^*, a, a' \in \Sigma.
\end{aligned}
$$

Ein wenig visueller bedeutet dies:

$$
\begin{aligned}
uzaw &\vdash_M w'a'zv', && \text{wenn } (z', a', R) \in \delta(z, a),\\
u'bzaw &\vdash_M u'zba'w, && \text{wenn } (z', a', L) \in \delta(z, a),\\
uzaw &\vdash_M uz'a'w, && \text{wenn } (z', a', N) \in \delta(z, a).
\end{aligned}
$$

[1] Häufig werden TMen auch als Tupel über $(Z, \Sigma, \Gamma, \delta, \square, E)$ definiert und damit zwischen Eingabe- und Ausgabealphabet unterschieden sowie das leere Bandsymbol $\square$ explizit angegeben. E ist dann hier einfach die Menge der Endzustände.

Häufig lässt man einfach das tiefgestellte M bei $\vdash_M$ weg und schreibt einfach $\vdash$, wenn es klar ist, um welche Maschine es gerade geht. Gibt es eine Folge von Konfigurationen $k_1, \ldots, k_n$ sodass $k_i \vdash k_{i+1}$ für alle $1 \leq i < n$ gilt, dann schreibt man auch $k_1 \vdash^{n-1} k_n$. Allgemeiner tritt öfters auch die Schreibweise $k \vdash^* k'$ auf, wenn gemeint ist, dass eine beliebige Folge von Konfigurationsübergängen möglich ist, um von k zu k' zu kommen. ◀

Die *Startkonfiguration* einer TM $M = (Z, \Gamma, \delta, z_0, A, V)$ auf Eingabe w ist dann $z_0 w$. Nun können wir auch die Sprache einer Turingmaschine definieren:

$$L(M) = \{w \in \Sigma^* \mid z_0 w \vdash_M^* uz'v \text{ mit } z' \in A\}.$$

Nachdem nun die nötigen Formalia definiert sind, wenden wir uns dem ersten Beispiel zu.

Beispiel 7. Fangen wir mit einem einfachen Beispiel an, um Turingmaschinen besser verstehen zu können. Die Sprache, die wir betrachten wollen, ist

$$L = \{a^n b^n c^n \mid n \geq 1\}.$$

Wie wir auf Seite 11 definiert haben, ist a^n das Wort $\overbrace{a \circ \cdots \circ a}^{n \text{ mal}}$. Das heißt, die hier definierte Sprache L besteht aus Wörtern mit Blöcken aus gleicher Länge von a's, b's und c's.

Zum Beispiel sind also Wörter wie $abc, a^3b^3c^3, a^{42}b^{42}c^{42}$ in L enthalten und Wörter wie $aabc, cab, acabac, ccaabb$ nicht in L enthalten, da entweder die Reihenfolge oder die Anzahlgleichheit verletzt werden.

Nun möchten wir für L eine Turing Maschine M angeben, sodass $L(M) = L$ gilt. Definieren wir hierzu die Turing Maschine

$$M = (\{z_0, \ldots, z_4, z_a, z_b, z_c, z_L\}, \{a, b, c, \#, \square\}, \delta, z_0, \{z_a\}, \{z_v\}),$$

wobei die Überführungsfunktion δ wie folgt gegeben ist.

$z_0 a$	$\to$	$z_a a R$	Test, ob Eingabe die Form $a^+b^+c^+$ hat.
$z_a a$	$\to$	$z_a a R$	
$z_a b$	$\to$	$z_b b R$	
$z_b b$	$\to$	$z_b b R$	
$z_b c$	$\to$	$z_c c R$	
$z_c c$	$\to$	$z_c c R$	
$z_c \square$	$\to$	$z_L \square L$	
$z_L c$	$\to$	$z_L c L$	Durchlauf nach links.
$z_L b$	$\to$	$z_L b L$	
$z_L a$	$\to$	$z_L a L$	
$z_L \#$	$\to$	$z_L \# L$	
$z_L \square$	$\to$	$z_1 \square R$	
$z_1 \#$	$\to$	$z_1 \# R$	Suchen und Überschreiben des ersten as.
$z_1 a$	$\to$	$z_2 \# R$	
$z_1 \square$	$\to$	$z_a \square L$	
$z_2 \#$	$\to$	$z_2 \# R$	Suchen und Überschreiben des ersten bs.
$z_2 b$	$\to$	$z_3 \# R$	
$z_2 a$	$\to$	$z_2 a R$	
$z_3 \#$	$\to$	$z_3 \# R$	Suchen und Überschreiben des ersten cs.
$z_3 c$	$\to$	$z_L \# L$	
$z_3 b$	$\to$	$z_2 b R$	

Nun geben wir noch die Fälle an, in denen die Maschine verwerfen soll:

$z_0 x \to z_v x N$	$x \in \{b, c, \square\}$	(kein a vorne)
$z_a x \to z_v x N$	$x \in \{c, \square\}$	(c vor b)
$z_b x \to z_v x N$	$x \in \{a, \square\}$	(a nach b)
$z_1 x \to z_v x N$	$x \in \{b, c\}$	(zu wenig a)
$z_2 x \to z_v x N$	$x \in \{c, \square\}$	(zu wenig b)
$z_3 x \to z_v x N$	$x \in \{a, \square\}$	(zu wenig c)

Um die Notation von vorher aufzugreifen, ist für M also

$$\begin{aligned} Z &= \{z_0, z_1, z_2, z_3, z_4, z_a, z_b, z_c, z_L\}, \\ \Gamma &= \{a, b, c, \#, \square\}, \\ A &= \{z_a\} \text{ und} \\ V &= \{z_v\}. \end{aligned}$$

Begründung: Sei w die Eingabe. Zuerst durchläuft M das Wort w von links nach rechts und testet, ob w die Form $a^+b^+c^+$ hat. Ist dies nicht der Fall, so bleibt M stecken und akzeptiert nicht. Natürlich gilt dann auch $w \notin L$.

Gilt aber $w = a^k b^\ell c^m$ mit $k, \ell, m \geq 1$ so läuft M wieder auf die linke Begrenzung. Ab jetzt ist die Arbeit von M in Durchgänge eingeteilt: In jedem Durchgang läuft M nach rechts und ersetzt jeweils ein a, b und c durch $\#$. Am Ende des Durchgangs läuft M dann wieder auf die linke Begrenzung.

Gilt nicht $k = \ell = m$, d.h. $w \notin L$, so bleibt M in Durchgang $1 + \min\{k, \ell, m\}$ stecken, weil er eines der Zeichen a, b oder c nicht mehr findet.

Gilt aber $k = \ell = m$, d.h. $w \in L$, so steht nach Durchgang k das Wort $\#^k\#^k\#^k$ zwischen den Begrenzungen. In Durchgang $k+1$ läuft dann M im Zustand z_1 komplett durch das Wort nach rechts bis zum rechten Bandende „$\square$“. Dann geht M in den akzeptierenden Zustand z_4 über. ◀

Bemerkung. Häufig ist es so, dass bei der Definition einer Maschine die verwerfenden Fälle (so wie wir sie hier angegeben haben) nicht erwähnt werden. Für alle dann “undefinierten” Fälle nimmt man an, dass die Maschine hierbei in einen verwerfenden Zustand wechselt. Daher geschieht es auch häufig, dass es nur akzeptierende Zustände gibt und die Menge V der verwerfenden Zustände unerwähnt bleibt.

Manchmal tritt dieses Problem auch ein, wenn man diese Nicht-Akzeptanz (bzw. dieses Verwerfen) durch eine Anweisung der Form $(z, a) \in \delta(z, a, N)$ realisiert. Dies ist also eine Endlosschleife der Maschine. Natürlich kann sie dann nie akzeptieren, aber wir werden später auf Seite 18 fordern, dass unsere Maschinen von da an immer halten. Dann müssen natürlich (rein formal) solche Fälle behandelt werden. Die Argumentation darüber, dass undefinierte Fälle zum Verwerfen führen, ist dann nicht mehr so sauber. Hier bietet sich

dann ein zusätzlicher Zustand an, welcher die Eingabe einfach zu Ende liest und am Ende verwirft. ◀

Übungsaufgabe 1. Die Funktion Ω über den Alphabet Σ ist für alle Eingaben $w \in \Sigma^*$ undefiniert. In unserem Kontext möchten wir annehmen, dass die Turingmaschine, die diese Funktion berechnet, dann keine Eingabe akzeptiert. Damit ist die Sprache dieser Maschine äquivalent zur leeren Sprache $\emptyset$. Geben Sie nun diese Turingmaschine vollständig an. Gehen Sie dabei davon aus, dass $\Sigma = \{0, 1\}$ gilt. ◀

1.3. Viele Bänder bringen nicht viel mehr als zwei

Nachdem wir in Abschnitt 1.2 Turingmaschinen auf einer sehr formalen Ebene betrachtet haben, wollen wir von nun an davon Abstand nehmen die genaue Übergangsfunktion im kleinsten Detail angeben. Stattdessen greifen wir auf Algorithmen in *Pseudocode* zu, da diese äquivalent zu Turingmaschinen sind. Betrachten wir die genaue Laufzeit eines solchen Pseudocode Algorithmus, dann überlegen wir uns jeweils, wie eine Turingmaschine die jeweiligen Anweisungen umsetzen kann. Damit Komplexitätsbetrachtung Sinn macht, wollen wir die Vereinbarung treffen, dass alle unsere Maschinen auf jeder Eingabe irgendwann einen (akzeptierenden oder verwerfenden) Endzustand erreichen und daher *halten*.

Laufzeit u. Speicherbedarf

Definition 3.

- Die *Laufzeit* einer Turingmaschine M bei Eingabe eines Wortes w ist die Anzahl der Rechenschritte, bevor M stoppt.
- Der *Speicherbedarf* einer Turingmaschine M bei Eingabe eines Wortes w ist die Anzahl der Bandzellen auf den Arbeitsbändern (d. h., nicht auf dem Eingabeband, falls vorhanden), die M während der Rechnung besucht. ◀

Rechenschritte meint hier Transitionsübergänge der δ-Funktion.

Zeit und Platz

Definition 4. Sei M eine Turingmaschine. Sei $f\colon \mathbb{N} \to \mathbb{N}$.

M arbeitet in *Zeit* f, falls für alle n und für alle Wörter w der Länge n die Laufzeit von M bei Eingabe w durch $f(n)$ beschränkt ist.

M arbeitet in *Platz* f, falls für alle n und für alle Wörter w der Länge n der Speicherbedarf von M bei Eingabe w durch $f(n)$ beschränkt ist. ◀

Beispiel 8. Wir betrachten die Sprache $A = \{0^k 1^k \mid k \geq 0\}$. A kann durch eine Einband-Turingmaschine erkannt werden, die durch den folgenden Pseudocode spezifiziert wird:

```
1: if eine 0 rechts von einer 1 steht then
2:     verwerfe
3: while sowohl Symbol 0 und 1 auf dem Arbeitsband bleiben do
4:     Streiche eine 0 und eine 1.
5: if es gibt noch eine 0 aber keine 1 oder umgekehrt then
6:     verwerfe
7: else
8:     akzeptiere
```

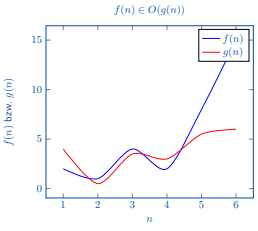
Geben Sie zu Ihrer Paritätsmaschine von S. 13 die Zeitzeit und den Speicherbedarf an.

Donald Knuth

Nun möchten wir uns der Laufzeit dieser Maschine widmen und stoßen auf ein kleines Problem: Wie wir bereits zu Beginn des Kapitels gemerkt haben, ist eine genaue Angabe in hohem Grade vom verwendeten Maschinenmodell abhängig. Um dieses jedoch auszublenden, sind wir nur am *Wachstumsverhalten* der Laufzeit interessiert und treffen daher folgende Vereinbarung, die auf Knuth zurückgeht, der die O-Notation zur Algorithmenanalyse eingeführt hat.

Definition 5. Seien $f, g\colon \mathbb{N} \to \mathbb{N}$ zwei Funktionen. Dann ist:
$f \in O(g)$, falls es $c, n_0 \in \mathbb{N}$ gibt, sodass für alle $n \geq n_0$ gilt:

$$f(n) \leq c \cdot g(n),$$



wir schreiben dann auch: $f(n) \in O(g(n))$. Das heißt, f wächst *höchstens so stark wie* g. Man kann die O-Notation auch über Grenzwerte wie folgt definieren:

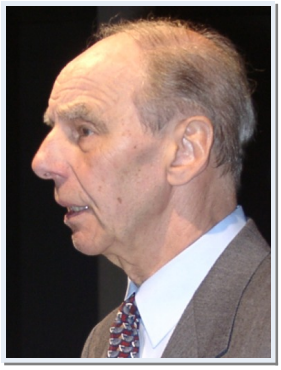
$$f(n) \in O(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty,$$

wenn der Grenzwert existiert, ansonsten als $\limsup_{n\to\infty} \frac{f(n)}{g(n)}$.

Es gilt $f \in o(g)$, falls

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Analog schreiben wir auch: $f(n) \in o(g(n))$. Das heißt, f wächst *echt schwächer als* g. ◀



Juris Hartmanis

Nun wieder zurück zum Beispiel: Bei einer Eingabelänge von n ergibt sich eine Laufzeit dann wie folgt:

Schritt (1):	$O(n)$
Schritt (4):	$O(n)$
Schritt (5):	$O(n)$
Schleife (3)–(4)	wird $O(n)$ mal durchlaufen
Insgesamt:	$O(n^2)$

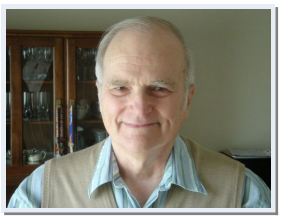
◀

Die folgenden Klassen gehen nun zurück auf Hartmanis und Stearns, die in ihrer Arbeit *On the computational complexity of algorithms. In: Transactions of the American Mathematical Society. Vol. 117, 1965, S. 285–306* die erste Definition hierzu gebracht haben.

Übungsaufgabe 2. Zeigen Sie, dass das vorherige Resultat für die Sprache A aus Beispiel 8 auch für die Sprache L aus Beispiel 7 gilt. ◀

TIME **Definition 6.** Sei $t\colon \mathbb{N} \to \mathbb{N}$. Die Komplexitätsklasse TIME(t) besteht aus allen Sprachen A, für die es eine Mehrband-Turingmaschine gibt, die A entscheidet und in Zeit $O(t)$ arbeitet. ◀

SPACE **Definition 7.** Sei $s\colon \mathbb{N} \to \mathbb{N}$. Die Komplexitätsklasse SPACE(s) besteht aus allen Sprachen A, für die es eine Mehrband-Turingmaschine mit Eingabeband gibt, die A entscheidet und in Platz $O(s)$ arbeitet. ◀



Dick Stearns

Nun haben wir uns auf ein formales Konzept geeinigt und wollen es auf das vorherige Beispiel anwenden. Es gilt also $A \in \text{TIME}(n^2)$. Eine bessere Schranke ergibt sich mit Hilfe von 2-Band-Turingmaschinen:

```
1: if eine 0 rechts von einer 1 gefunden wurde then
2:     verwerfe
3: Kopiere alle Symbole 0 auf Arbeitsband.
4: Für jede 1 auf Eingabeband: Lösche eine 0 vom Arbeitsband.
5: if 1 auf Eingabeband oder 0 auf Arbeitsband then
6:     verwerfe
7: akzeptiere
```

Es folgt: $A \in \mathrm{TIME}(n)$.

Übungsaufgabe[L] 3. Auf 1-Band-Turingmaschinen kann A in Zeit $O(n \log(n))$ akzeptiert werden. ◀

Bemerkung. Eine Sprache L ist regulär genau dann, wenn L in Zeit $o(n \cdot \log(n))$ auf einer 1-Band-Turingmaschine akzeptiert wird. ◀

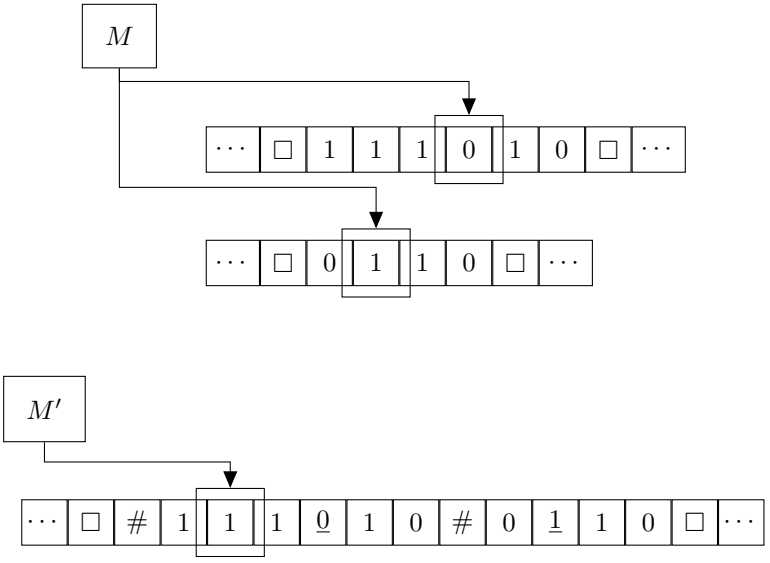
Im Folgenden stellen wir uns die berechtigte Frage, wie mächtig die Benutzung von mehreren Bändern ist. Diese kann man mit dem folgenden Satz beantworten.

Satz 1. Sei $t(n) \geq n$ eine Funktion. Jede Mehrband-Turingmaschine M, die in Zeit t arbeitet, kann von einer 1-Band-Turingmaschine M' simuliert werden, die in Zeit $O(t^2)$ arbeitet. ◀

Das heißt, die Benutzung von beliebig vielen Bändern kann durch *ein einziges* Band mit einem quadratischen Anstieg der Laufzeit simuliert werden.

Beweis (Satz 1) Ausgehend von der Maschine M mit k Bändern, konstruieren wir nun unsere 1-Band-Maschine M'. Die Idee ist, dass M' auf seinem Band die ganzen k Bandinhalte und Kopfpositionen von M speichert. Diese Informationen werden bandweise getrennt durch ein Sonderzeichen (#). Zur Simulation eines Schritts von M muss M' über sein gesamtes Band laufen, um die Inhalte und Kopfpositionen der k Bänder von M zu aktualisieren.

Das folgende Bild zeigt die Konstruktion im Beispiel einer 2-Band-TM:



Die Kopfpositionen werden gesondert markiert (mit einem Unterstrich).

In jedem Simulationsschritt von M benötigt M' eine Laufzeit von $O(t(n))$. Insgesamt hat M' also eine Laufzeit von $O(t^2)$. (Satz 1) ■

Im folgenden Satz sieht man, dass man den quadratischen Anstieg verhindern kann, wenn man mindestens zwei Bänder hat. Wir beweisen den Satz jedoch nicht.

Satz 2. Sei $t(n) \geq n$ eine Funktion. Jede Mehrband-Turingmaschine M, die in Zeit t arbeitet, kann von einer 2-Band-Turingmaschine M' simuliert werden, die in Zeit $O(t \cdot \log(t))$ arbeitet. ◀

Übungsaufgaben.

Übungsaufgabe[L] 4. Beweisen oder widerlegen Sie die folgenden Behauptungen.

a) $2n \in O(n)$ für alle festen $k \in \mathbb{N}\backslash\{0,1\}$

b) $n^2 \in O(n)$

c) $\log_2(n) \in O(\log_k(n))$

d) $n \cdot \log_2(n) \in O(n^2)$

e) $3^n \in 2^{O(n)}$

f) $(2^n)^3 \in 2^{O(n)}$

g) $2^{n^3} \in 2^{O(n)}$

h) $O(2^n) = O(3^n)$

i) $O(n^2) + O(n) = O(n^2)$

j) $O(n) - O(n) = O(0)$ ◀

Übungsaufgabe[L] 5. Beweisen oder widerlegen Sie die folgenden Behauptungen.

a) $n \in o(2n)$

b) $2n \in o(n^2)$

c) $n^k \in o(2^n)$ für alle festen $k \in \mathbb{N}$

d) $1 \in o(n)$

e) $1 + 2 + \cdots + n \in o(n^2)$ ◀

Übungsaufgabe 6. Beweisen oder widerlegen Sie:

a) $n^2 \in O(n \cdot \log(n))$

b) $n! \in O(2^n)$

c) $O(2^{2n}) = O(2^n)$

d) $O(n) \cdot (c^{s(n)})^k \cdot (s(n))^k \subseteq 2^{O(s(n))}$, wobei $s(n) \geq \log n$ eine Funktion ist und k eine Konstante.

e) $2^n \in o(3^n)$

f) $\log_2(n) \in o(n)$

g) $n^2 \in o(\log_2(n))$

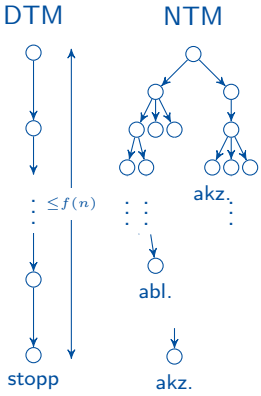
h) $o(g(n)) \subseteq O(g(n))$ für alle $g : \mathbb{N} \to \mathbb{N}$ ◀

1.4. Nichtdeterminismus

Nichtdeterminismus kann auf verschiedene Weisen dargestellt werden. Für diejenigen Hörer, welche die Veranstaltung *Grundlagen der Theoretischen Informatik* nicht besucht haben, betrachten wir dieses Konzept zunächst genauer. Nehmen wir eine Turingmaschine $M = (Z, \Gamma, \delta, E)$ aus dem vorherigen Abschnitt als gegeben an. Formal gilt $|\delta(z, x)| \leq 1$ für $z \in Z$ und $x \in \Gamma$; das heißt, für jeden Übergang ist *determiniert*, was im Folgeschritt geschieht. Dies ist für nichtdeterministische Maschinen *nicht* der Fall. Diese Maschinen haben in jedem Schritt die Möglichkeit, zwischen verschiedenen auszuführenden Befehlen zu *wählen*.

Diese Wahl kann man sich anschaulich auf verschiedenste Weise vorstellen:

- Die Maschine rät den Folgebefehl.
- Die Maschine kopiert sich $|\delta(z, x)|$ mal und rechnet parallel weiter.



Formal wird diese Eigenschaft der Überführungsfunktion durch eine Potenzmenge definiert:

$$\delta\colon Z \times \Gamma \to \mathcal{P}(Z \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{N}\}).$$

Die Struktur einer Berechnung einer solchen nichtdeterministischen Turingmaschine (NTM) ist nun ein Baum. Wir sagen, dass eine NTM M ihre Eingabe w akzeptiert, falls es eine akzeptierende Rechnung gibt. Das heißt, es muss einen Pfad in dem Berechnungsbaum von M geben, der zu einem akzeptierenden Zustand führt. Die von M *entschiedene* Sprache ist die Menge der von M akzeptierten Wörter.

Die Laufzeit von M ist die Tiefe des Berechnungsbaumes von M, also die maximale Anzahl von Rechenschritten, bevor M stoppt, über alle nichtdeterministischen Wahlmöglichkeiten.

Laufzeit NTM

Definition 8. Seien eine nichtdeterministische Turingmaschine M und die Funktion $f\colon \mathbb{N} \to \mathbb{N}$ gegeben. M arbeitet in Zeit f, falls für alle $n \in \mathbb{N}$ und für alle Wörter w der Länge n die Laufzeit von M bei Eingabe w durch $f(n)$ beschränkt ist. ◀

Die nichtdeterministische Komplexitätsklasse ist wie folgt definiert.

Definition 9. Sei $t\colon \mathbb{N} \to \mathbb{N}$ eine Funktion. Die Komplexitätsklasse $\mathrm{NTIME}(t)$ besteht aus allen Sprachen A, für die es eine Mehrband-NTM gibt, die A entscheidet und in Zeit $O(t)$ arbeitet. ◀

Der Speicherbedarf einer NTM wird nun analog zum Speicherbedarf einer gewöhnlichen (deterministischen) TM über die *maximale Anzahl* der besuchten Bandzellen *über alle Rechenwege* definiert.

Speicherbedarf NTM, NSPACE

Definition 10. Sei $s\colon \mathbb{N} \to \mathbb{N}$ eine Funktion. Eine NTM M arbeitet in Platz s, falls für alle n und für alle Wörter w der Länge n der

Speicherbedarf von M bei Eingabe w (auf jedem Rechenweg) durch $s(n)$ beschränkt ist.

Die Komplexitätsklasse NSPACE(s) besteht aus allen Sprachen A, für die es eine Mehrband-NTM mit Eingabeband gibt, die A entscheidet und in Platz $O(s)$ arbeitet. ◀

Übungsaufgaben.

Übungsaufgabe 7. Es sei $f\colon \mathbb{N} \to \mathbb{N}$.

a) Zeigen Sie: Ist $A \in \mathrm{TIME}(f(n))$, so ist auch $\overline{A} \in \mathrm{TIME}(f(n))$.

b) Zeigen Sie: Ist $A \in \mathrm{SPACE}(f(n))$, so ist auch $\overline{A} \in \mathrm{SPACE}(f(n))$.

a) Sind die Erkenntnisse aus den Aufgaben a) und b) direkt übertragbar auf die Komplexitätsklassen NTIME bzw. NSPACE? Begründen Sie ihre Antwort. ◀

Übungsaufgabe 8. Geben Sie je eine Sprache aus der Klasse TIME(1) und SPACE(1) an. ◀

Übungsaufgabe 9. Im Folgenden betrachten wir die von Seite 19 bekannte Sprache $A := \{0^k 1^k \mid k \geq 0\}$. Zeigen Sie, dass $A \in \mathrm{SPACE}(\log(n))$ gilt. Beschreiben Sie hierzu die Funktionsweise der Turingmaschine vollständig und begründen Sie den Speicherbedarf der Maschine. ◀

Übungsaufgabe 10. Es sei ⋆⋆⋆

$$B := \{\mathrm{bin}(0) \diamond \mathrm{bin}(1) \diamond \cdots \diamond \mathrm{bin}(n) \mid n \in \mathbb{N}\},$$

wobei $\mathrm{bin}(k)$ für $k \in \mathbb{N}$ die Binärdarstellung von k ohne führende Nullen ist. Zeigen Sie, dass $B \in \mathrm{SPACE}(\log(\log(n)))$ gilt. Beschreiben Sie hierzu die Funktionsweise Ihrer Turingmaschine vollständig und zeigen Sie, dass der Speicherbedarf der Maschine $O(\log(\log(n)))$ ist.

Hinweis: Vergleichen Sie benachbarte Binärzahlen miteinander und beachten Sie, dass n hier nicht die Länge der Eingabe kodiert. ◀