Skript zur Vorlesung

Algorithmen und Berechnungskomplexität II

Prof. Dr. Heiko Röglin
Institut für Informatik

Sommersemester 2014

8. April 2014

Inhaltsverzeichnis

Bitte senden Sie Hinweise auf Fehler im Skript und Verbesserungsvorschläge an die E-Mail-Adresse `roeglin@cs.uni-bonn.de`.

Kapitel 1

Einleitung

Wir haben im vergangenen Semester effiziente Algorithmen zur Lösung einer Vielzahl von Problemen entworfen. In dieser Vorlesung werden wir die Grenzen der Algorithmik kennenlernen und uns damit beschäftigen, für welche Probleme es keine effizienten Algorithmen gibt und welche überhaupt nicht algorithmisch gelöst werden können.

Wir starten mit einer Einführung in die *Berechenbarkeitstheorie*. Dieses Teilgebiet der theoretischen Informatik ist der Frage gewidmet, welche Probleme nicht durch Algorithmen in endlicher Zeit gelöst werden können und zwar unabhängig davon, wie leistungsfähig unsere Rechner in der Zukunft auch sein mögen. Die Existenz solcher *unberechenbaren* Probleme ist uns bereits aus der Vorlesung „Logik und diskrete Strukturen“ bekannt. Wir haben dort jedoch kein konkretes nicht berechenbares Problem angegeben, sondern nur ausgenutzt, dass die Menge der Funktionen überabzählbar ist, während es nur abzählbar unendlich viele Programme gibt. Dieses Ergebnis lässt die Möglichkeit offen, dass alle in der Praxis relevanten Probleme algorithmisch gelöst werden können und nicht berechenbare Probleme im Informatik-Alltag keine Rolle spielen.

Wir werden jedoch am Beispiel des *Halteproblems* demonstrieren, dass es auch ausgesprochen wichtige Probleme gibt, die nicht berechenbar sind. Bei diesem Problem soll für ein gegebenes Programm entschieden werden, ob es nach endlich vielen Schritten terminiert. Zur Verifikation der Korrektheit von Programmen wäre es hilfreich, einen Compiler zu haben, der bei nicht terminierenden Programmen eine Warnung ausgibt. Wir werden beweisen, dass es einen solchen Compiler nicht geben kann, da das Halteproblem nicht berechenbar ist. Anschließend werden wir Techniken kennenlernen, mit deren Hilfe man auch für viele weitere Probleme nachweisen kann, dass sie nicht berechenbar sind.

Im zweiten Teil der Vorlesung werden wir uns mit *Komplexitätstheorie* beschäftigen. In diesem Teilgebiet der theoretischen Informatik geht es um die Frage, welche Ressourcen notwendig sind, um bestimmte Probleme zu lösen. Wir konzentrieren uns insbesondere auf die Ressource Rechenzeit, denn für praktische Anwendungen ist die Laufzeit eines Algorithmus oft von entscheidender Bedeutung. Wer möchte schon einen Tag oder gar mehrere Jahre auf die Ausgabe seines Navigationsgerätes warten? Leider gibt es eine

ganze Reihe von Problemen, die zwar berechenbar sind, für die es aber vermutlich keine effizienten Algorithmen gibt. Ein solches Problem, das in vielen logistischen Anwendungen eine Rolle spielt, ist das *Problem des Handlungsreisenden*, bei dem eine Landkarte mit mehreren Städten gegeben ist und die kürzeste Rundreise durch diese Städte gesucht wird.

Das Problem des Handlungsreisenden gehört wie viele andere natürliche Probleme zu der Klasse der *NP-schweren* Probleme. Man vermutet, dass es für diese Probleme keine effizienten Algorithmen gibt. Diese sogenannte $P \neq NP$*-Vermutung* ist bis heute unbewiesen und eines der größten ungelösten Probleme der Mathematik und der theoretischen Informatik. Wir werden besprechen, was diese Vermutung genau besagt, und zahlreiche NP-schwere Probleme kennenlernen.

Zwar werden in dieser Vorlesung hauptsächlich negative Ergebnisse gezeigt, diese haben aber wichtige Auswirkungen auf die Praxis. Hat man beispielsweise bewiesen, dass ein Problem nicht effizient gelöst werden kann, so ist klar, dass die Suche nach einem effizienten Algorithmus eingestellt werden kann und dass man stattdessen über Alternativen (Abwandlung des Problems etc.) nachdenken sollte. Außerdem beruhen Kryptosysteme auf der Annahme, dass gewisse Probleme nicht effizient gelöst werden können. Wäre es beispielsweise möglich, große Zahlen effizient zu faktorisieren, so wäre RSA kein sicheres Kryptosystem. In diesem Sinne können also auch negative Ergebnisse gute Nachrichten sein.

Zum Abschluss der Vorlesung werden wir uns mit *Approximationsalgorithmen* beschäftigen. Trifft die P $\neq$ NP-Vermutung zu, so können NP-schwere Probleme nicht effizient gelöst werden. Für solche Probleme ist es deshalb oft sinnvoll, die Anforderungen zu reduzieren und statt nach einer optimalen nur noch nach einer möglichst guten Lösung zu suchen. Ein r*-Approximationsalgorithmus* ist ein effizienter Algorithmus, der für jede Eingabe eine Lösung berechnet, die höchstens um den Faktor r schlechter ist als die optimale Lösung. Ein 2-Approximationsalgorithmus für das Problem des Handlungsreisenden ist also beispielsweise ein Algorithmus, der stets eine Rundreise berechnet, die höchstens doppelt so lang ist wie die kürzeste Rundreise. Wir werden für einige NP-schwere Probleme Approximationsalgorithmen entwerfen und die Grenzen dieses Ansatzes diskutieren.

Es gibt zahlreiche Bücher und Skripte, in denen die Inhalte dieser Vorlesung nachgelesen werden können (z. B. [1, 2, 3]). Diese Vorlesung orientiert sich insbesondere an dem Buch von Ingo Wegener [6] und dem Skript von Berthold Vöcking [5].

1.1 Probleme und Funktionen

Wir haben bereits im letzten Semester diskutiert, dass ein *Algorithmus* eine Handlungsvorschrift ist, mit der Eingaben in Ausgaben transformiert werden können. Diese Handlungsvorschrift muss so präzise formuliert sein, dass sie von einem Computer ausgeführt werden kann. Unter einem *Problem* verstehen wir informell den gewünschten Zusammenhang zwischen der Eingabe und der Ausgabe. Beim Sortierproblem besteht

die Eingabe beispielsweise aus einer Menge von Zahlen und die gewünschte Ausgabe ist eine aufsteigend sortierte Permutation dieser Zahlen.

Um den Begriff „Problem“ zu formalisieren, gehen wir davon aus, dass (wie bei der Betrachtung von formalen Sprachen) eine beliebige endliche Menge Σ gegeben ist, die wir *Alphabet* nennen. Die Eingaben und Ausgaben sind als Zeichenketten über diesem Alphabet codiert. Denkt man an reale Rechner, so ergibt die Wahl $\Sigma = \{0, 1\}$ natürlich Sinn. Für theoretische Betrachtungen ist es aber manchmal hilfreich (wenngleich nicht notwendig) Alphabete mit mehr als zwei Zeichen zu betrachten. Wenn nicht explizit anders erwähnt, so gehen wir in dieser Vorlesung stets davon aus, dass $\{0, 1\} \subseteq \Sigma$ gilt. Wie üblich bezeichnen wir mit Σ^* die Menge aller Zeichenketten endlicher Länge, die sich über dem Alphabet Σ bilden lassen. Dazu gehört insbesondere das leere Wort ε der Länge 0. Für ein Wort $w \in \Sigma^*$ bezeichnen wir mit $|w|$ seine Länge. Außerdem bezeichnet w^R das gespiegelte Wort, das heißt für $w = w_1 \dots w_n$ ist $w^R = w_n \dots w_1$.

Unter einem *Problem* verstehen wir eine Relation $R \subseteq \Sigma^* \times \Sigma^*$ mit der Eigenschaft, dass es für jede Eingabe $x \in \Sigma^*$ mindestens eine Ausgabe $y \in \Sigma^*$ mit $(x, y) \in R$ gibt. Gibt es zu jeder Eingabe eine eindeutige Ausgabe, so können wir das Problem auch als Funktion $f\colon \Sigma^* \to \Sigma^*$ beschreiben, die jeder Eingabe $x \in \Sigma^*$ ihre Ausgabe $f(x) \in \Sigma^*$ zuweist. Ein Algorithmus *löst* ein Problem, das durch eine Relation R beschrieben wird, wenn er zu jeder Eingabe $x \in \Sigma^*$ eine Ausgabe $y \in \Sigma^*$ mit $(x, y) \in R$ produziert. Ein Algorithmus löst ein Problem, das durch eine Funktion f beschrieben wird, wenn er zu jeder Eingabe $x \in \Sigma^*$ die Ausgabe $f(x)$ produziert. Wir sagen dann auch, dass der Algorithmus die Funktion f *berechnet*.

Wir legen ein besonderes Augenmerk auf Funktionen der Form $f\colon \Sigma^* \to \{0, 1\}$. Eine solche Funktion beschreibt ein sogenanntes *Entscheidungsproblem*, da für eine Eingabe $x \in \Sigma^*$ nur entschieden werden muss, ob $f(x) = 0$ oder $f(x) = 1$ gilt. Eine *Sprache* über dem Alphabet Σ ist eine Teilmenge von Σ^*. Zwischen Entscheidungsproblemen und *Sprachen* existiert eine eineindeutige Beziehung. Jedes Entscheidungsproblem $f\colon \Sigma^* \to \{0, 1\}$ kann als Sprache $L_f = \{x \in \Sigma^* \mid f(x) = 1\}$ aufgefasst werden. Ebenso kann jede Sprache $L \subseteq \Sigma^*$ als Funktion $f_L\colon \Sigma^* \to \{0, 1\}$ mit

$$f_L(x) = \begin{cases} 1 & \text{falls } x \in L \\ 0 & \text{falls } x \notin L \end{cases}$$

aufgefasst werden. Die Funktion f_L heißt die *charakteristische Funktion* der Sprache L.

Wir betrachten nun einige Beispiele für Probleme. In diesen Beispielen und im weiteren Verlauf der Vorlesung bezeichnen wir die Binärdarstellung einer Zahl $n \in \mathbb{N}_0$ ohne führende Nullen mit $\text{bin}(n)$ und wir bezeichnen mit $\text{val}(x) \in \mathbb{N}_0$ die Zahl, die durch $x \in \{0, 1\}^*$ binär codiert wird. Erlauben wir in der Binärdarstellung führende Nullen, so ist $\text{val}(x)$ für jede Zeichenkette $x \in \{0, 1\}^*$ eindeutig bestimmt.

- Das Problem, eine natürliche Zahl in Binärdarstellung zu quadrieren, lässt sich sowohl durch die Funktion $f\colon \{0, 1\}^* \to \{0, 1\}^*$ mit $f(x) = \text{bin}(\text{val}(x)^2)$ als auch durch die Relation

$$R = \{(x, y) \mid \text{val}(y) = \text{val}(x)^2\} \subseteq \Sigma^* \times \Sigma^*$$

beschreiben. Obwohl beides sinnvolle Modellierungen desselben Problems sind, sind sie formal nicht ganz identisch. Um dies einzusehen, betrachten wir einen Algorithmus, der das Quadrat der Eingabe korrekt berechnet, binär codiert ausgibt und an die Ausgabe zusätzlich noch eine führende Null anfügt. Dieser Algorithmus löst das Problem, das durch die Relation R beschrieben wird, er berechnet aber nicht die Funktion f.

- Das Problem, einen Primfaktor einer natürlichen Zahl zu bestimmen, lässt sich nicht als Funktion beschreiben, da es zu manchen Eingaben mehrere mögliche Ausgaben gibt. Stattdessen kann es aber durch die Relation

$$R = \{(x, y) \mid \text{val}(y) \text{ ist eine Primzahl, die val}(x) \text{ teilt}\} \subseteq \Sigma^* \times \Sigma^*$$

beschrieben werden.

- Wir betrachten als nächstes das Problem, für einen ungerichteten Graphen $G = (V, E)$ zu entscheiden, ob er zusammenhängend ist oder nicht. Um dieses Problem formal beschreiben zu können, müssen wir zunächst festlegen, wie der Graph G codiert wird. Dazu gibt es zahlreiche Möglichkeiten. Wir entscheiden uns dafür, ihn als Adjazenzmatrix darzustellen, wobei wir die Zeilen dieser Matrix nacheinander schreiben. Für einen Graphen mit n Knoten besitzt diese Darstellung eine Länge von n^2. Somit codieren genau solche Eingaben $x \in \{0,1\}^*$ einen Graphen, deren Länge $|x|$ eine Quadratzahl ist. Ist $|x|$ eine Quadratzahl, so bezeichnen wir mit $G(x)$ den durch x codierten Graphen. Wir können das Zusammenhangsproblem als Funktion $f\colon \{0,1\}^* \to \{0,1\}$ mit

$$f(x) = \begin{cases} 1 & \text{falls } |x| = n^2 \text{ für ein } n \in \mathbb{N} \text{ und } G(x) \text{ ist zusammenhängend} \\ 0 & \text{sonst} \end{cases}$$

modellieren. Diese Funktion gibt den Wert 1 aus, wenn die Eingabe einen zusammenhängenden Graphen codiert, und den Wert 0, wenn die Eingabe entweder syntaktisch nicht korrekt ist oder einen nicht zusammenhängenden Graphen codiert. Dies können wir auch als Sprache

$$L = \{x \in \{0,1\}^* \mid |x| = n^2 \text{ für ein } n \in \mathbb{N} \text{ und } G(x) \text{ ist zusammenhängend}\}$$

ausdrücken.

Der Leser möge sich als Übung für einige weitere Probleme aus dem letzten Semester überlegen, wie sie formal als Funktion oder Relation beschrieben werden können. Außerdem sollte er sich überlegen, warum es keine echte Einschränkung ist, nur das Alphabet $\Sigma = \{0,1\}$ zu betrachten.

1.2 Rechnermodelle

Bereits im vergangenen Semester haben wir gesehen, dass zum Entwurf und zur Analyse von Algorithmen ein *Rechnermodell* benötigt wird. Dabei handelt es sich um ein formales Modell, das festlegt, welche Operationen ein Algorithmus ausführen darf und welche Ressourcen (insbesondere welche Rechenzeit) er für diese Operationen benötigt. Wir haben das Modell der *Registermaschine* kennengelernt, das in etwa dieselben Operationen bietet wie eine rudimentäre Assemblersprache. Wir haben uns dann davon überzeugt, dass Registermaschinen das Verhalten realer Rechner gut abbilden, und deshalb für die allermeisten Anwendungen ein vernünftiges Rechnermodell darstellen.

In dieser Vorlesung werden wir jedoch stattdessen das Modell der *Turingmaschine* zugrunde legen. Dieses hat auf den ersten Blick deutlich weniger mit realen Rechnern gemein als Registermaschinen, es ist dafür aber strukturell einfacher und erleichtert deshalb die theoretischen Betrachtungen. Wir werden sehen, dass Turingmaschinen trotz ihrer Einfachheit genauso mächtig sind wie Registermaschinen. Damit sind auch Turingmaschinen ein gutes Modell für reale Rechner, auch wenn man das anhand ihrer Definition nicht direkt erkennen kann.

Nachdem wir das Modell der Turingmaschine eingeführt haben, werden wir die *Church-Turing-These* diskutieren. Diese These besagt, dass eine Turingmaschine alle Funktionen berechnen kann, die „intuitiv berechenbar“ sind. Diese These kann prinzipiell nicht bewiesen werden, da der Begriff „intuitiv berechenbar“ nicht formal definiert ist. Eine formalere Variante dieser These, die zwar prinzipiell beweisbar, aber noch unbewiesen ist, lautet wie folgt: Die Gesetze der Physik erlauben es nicht, eine Maschine zu konstruieren, die eine Funktion berechnet, die nicht auch von einer Turingmaschine berechnet werden kann.

1.2.1 Turingmaschinen

Informell kann man eine Turingmaschine als einen endlichen Automaten beschreiben, der mit einem Band mit unendlich vielen Speicherzellen ausgestattet ist. In jeder Zelle des Bandes steht eins von endlich vielen Zeichen. Dabei bezeichne Γ die endliche Menge von möglichen Zeichen und $\square \in \Gamma$ sei das Leerzeichen. Wie ein endlicher Automat befindet sich auch eine Turingmaschine zu jedem Zeitpunkt in einem von endlich vielen Zuständen, wobei wir die endliche Zustandsmenge wieder mit Q bezeichnen.

Darüber hinaus besitzt eine Turingmaschine einen Lese-/Schreibkopf, der zu jedem Zeitpunkt auf einer Zelle des Bandes steht und das dort gespeicherte Zeichen liest. Basierend auf dem gelesenen Zeichen und ihrem aktuellen Zustand kann die Turingmaschine in einem Rechenschritt das Zeichen an der aktuellen Kopfposition ändern, in einen anderen Zustand wechseln und den Kopf um eine Position nach links oder rechts verschieben. Zu Beginn befindet sich die Eingabe auf dem Band, der Kopf steht auf dem ersten Zeichen der Eingabe (oder auf einem Leerzeichen, wenn die Eingabe das leere Wort ε ist), alle Zellen links und rechts von der Eingabe enthalten das Leerzeichen $\square$ und die Turingmaschine befindet sich im Startzustand $q_0 \in Q$. Die Eingabe

ist dabei keine Zeichenkette aus Γ^*, sondern aus Σ^* für ein Eingabealphabet $\Sigma \subseteq \Gamma$ mit $\square \notin \Sigma$.

Die Turingmaschine führt solange Rechenschritte der oben beschriebenen Form durch, bis sie einen bestimmten Endzustand $\bar{q} \in Q$ erreicht hat. Bezeichne $w_i \in \Gamma$ beim Erreichen des Endzustandes $\bar{q}$ den Inhalt von Zelle i und sei $j \in \mathbb{Z}$ die Kopfposition zu diesem Zeitpunkt. Dann ist die Zeichenkette $w_j \dots w_{k-1} \in \Sigma^*$ die Ausgabe, wobei $k \geq j$ den eindeutigen Index bezeichne, für den $w_j, \dots, w_{k-1} \in \Sigma$ und $w_k \in \Gamma \setminus \Sigma$ gilt. Die Ausgabe ist also die Zeichenkette aus Σ^*, die an der aktuellen Kopfposition beginnt und nach rechts durch das erste Zeichen aus $\Gamma \setminus \Sigma$ beschränkt ist.

Genauso wie bei endlichen Automaten sind die Rechenschritte einer Turingmaschine durch eine Zustandsüberführungsfunktion δ bestimmt. Diese erhält als Eingaben den aktuellen Zustand und das Zeichen an der aktuellen Kopfposition und liefert als Ausgaben den neuen Zustand, das Zeichen, durch das das Zeichen an der Kopfposition ersetzt werden soll, und ein Element aus der Menge $\{L, N, R\}$, das angibt, ob der Kopf an der aktuellen Position stehen bleiben soll (N), eine Position nach links (L) oder eine Position nach rechts (R) verschoben werden soll. Formal handelt es sich dabei um eine Funktion $\delta\colon (Q \setminus \{\bar{q}\}) \times \Gamma \to Q \times \Gamma \times \{L, N, R\}$.

Definition 1.1. *Eine* Turingmaschine (TM) M *ist ein 7-Tupel* $(Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$, *das aus den folgenden Komponenten besteht.*

- Q, *die* Zustandsmenge, *ist eine endliche Menge von* Zuständen.
- $\Sigma \supseteq \{0, 1\}$, *das* Eingabealphabet, *ist eine endliche Menge von Zeichen.*
- $\Gamma \supseteq \Sigma$, *das* Bandalphabet, *ist eine endliche Menge von Zeichen.*
- $\square \in \Gamma \setminus \Sigma$ *ist das* Leerzeichen.
- $q_0 \in Q$ *ist der* Startzustand.
- $\bar{q}$ *ist der* Endzustand.
- $\delta\colon (Q \setminus \{\bar{q}\}) \times \Gamma \to Q \times \Gamma \times \{L, N, R\}$ *ist die* Zustandsüberführungsfunktion.

Abbildung 1.1 zeigt exemplarisch die Rechenschritte einer Turingmaschine. Die Eingabe ist das Wort $1001 \in \Sigma^*$, welches links und rechts von Leerzeichen umschlossen ist. Der Kopf steht zu Beginn auf dem ersten Zeichen der Eingabe und die Turingmaschine befindet sich im Startzustand q_0. Die Zustandsüberführungsfunktion δ gibt vor, dass im Zustand q_0 beim Lesen des Zeichens 1 in den Zustand q gewechselt wird, das Zeichen 1 geschrieben wird und der Kopf um eine Position nach rechts bewegt wird. Im zweiten Schritt wird in Zustand q das Zeichen 0 gelesen, was gemäß δ dazu führt, dass in den Zustand q' gewechselt, das Zeichen 1 geschrieben und der Kopf um eine Position nach rechts bewegt wird. Im dritten und letzten Schritt wird das Zeichen 0 im Zustand q' gelesen, was gemäß δ dazu führt, dass in den Endzustand $\bar{q}$ gewechselt, das Zeichen 0 geschrieben und der Kopf nicht bewegt wird. Die Ausgabe der Turingmaschine ist in diesem Beispiel das Wort $01 \in \Sigma^*$.

Mit jeder Turingmaschine M kann man eine Funktion $f_M \colon \Sigma^* \to \Sigma^* \cup \{\perp\}$ assoziieren, die für jede Eingabe $w \in \Sigma^*$ angibt, welche Ausgabe $f_M(w)$ die Turingmaschine bei dieser Eingabe produziert. In dem gerade besprochenen Beispiel gilt folglich $f_M(1001) = 01$. Erreicht die Turingmaschine M bei einer Eingabe w den Endzustand $\bar{q}$ nicht nach endlich vielen Schritten, so sagen wir, dass sie bei Eingabe w nicht hält (oder nicht terminiert), und wir definieren $f_M(w) = \perp$. Wir sagen, dass *die Turingmaschine M die Funktion f_M berechnet.*

Definition 1.2. *Eine Funktion $f \colon \Sigma^* \to \Sigma^*$ heißt* berechenbar *(oder* rekursiv[1]*), wenn es eine Turingmaschine M mit $f_M = f$ gibt. Eine solche Turingmaschine terminiert insbesondere auf jeder Eingabe.*

Wenn wir das Verhalten einer Turingmaschine beschreiben oder analysieren, so sprechen wir im Folgenden oft von der *Konfiguration* einer Turingmaschine. Darunter verstehen wir zu einem Zeitpunkt die aktuelle Kombination aus Bandinhalt, Kopfposition und Zustand.

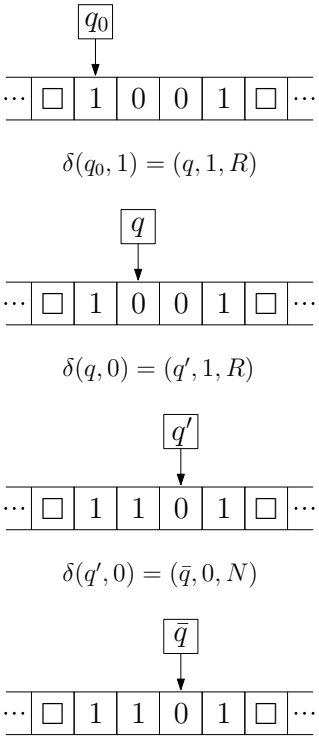

Abbildung 1.1: Beispiel für eine Turingmaschine

Wir betrachten ein Beispiel für eine Turingmaschine $M = (Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$. Es sei $Q = \{q_0, q_1, \ldots, q_5, \bar{q}\}$, $\Sigma = \{0, 1\}$ und $\Gamma = \{0, 1, \#, \square\}$. Die Zustandsüberführungsfunktion δ ist in der folgenden Tabelle dargestellt. Diese enthält an zwei Stellen das Symbol —, da die entsprechenden Kombinationen von Zustand und Zeichen nicht auftreten können (wie wir weiter unten argumentieren werden). Formal könnte man dort ein beliebiges Tripel aus $Q \times \Gamma \times \{L, N, R\}$ einsetzen, um die Tabelle zu vervollständigen.

	q_0	q_1	q_2	q_3	q_4	q_5
0	$(q_0, 0, R)$	$(q_2, \#, N)$	$(q_2, 0, R)$	$(q_3, 0, R)$	$(q_4, 0, L)$	$(\bar{q}, 0, N)$
1	$(q_0, 1, R)$	$(q_3, \#, N)$	$(q_2, 1, R)$	$(q_3, 1, R)$	$(q_4, 1, L)$	$(\bar{q}, 1, N)$
#	—	$(q_1, \#, L)$	$(q_2, \#, R)$	$(q_3, \#, R)$	$(q_1, \#, N)$	$(q_5, \square, R)$
□	$(q_1, \#, N)$	$(q_5, \square, R)$	$(q_4, 0, N)$	$(q_4, 1, N)$	—	$(\bar{q}, \square, N)$

Um zu verstehen, welche Funktion diese Turingmaschine berechnet, kann man sich zunächst ihr Verhalten auf einigen Beispielen anschauen. Diese kann man entweder von Hand durchgehen oder man schreibt ein Programm, mit dem man Turingmaschinen simulieren kann. Insbesondere Letzteres sei dem Leser als Übung empfohlen. Wir können aber auch versuchen, direkt zu verstehen, was die Turingmaschine

[1] Auf den ersten Blick hat das Wort „rekursiv“ in diesem Kontext nichts mit dem normalen Gebrauch dieses Wortes in der Informatik zu tun. Die Bezeichnung stammt von einer alternativen Charakterisierung berechenbarer Funktionen, die wir in dieser Vorlesung nicht besprechen werden.

in den einzelnen Zuständen macht. Die folgenden Eigenschaften können wir direkt aus der Zustandsüberführungsfunktion ablesen.

q_0: M schiebt den Kopf rechts bis zum Ende der Eingabe (ohne den Bandinhalt dabei zu verändern) und schreibt anschließend rechts neben die Eingabe das Zeichen #. Danach wird in den Zustand q_1 gewechselt und Zustand q_0 wird nie wieder erreicht.

q_1: M schiebt den Kopf nach links, solange an der aktuellen Position das Zeichen # steht. Das weitere Vorgehen ist abhängig vom ersten Zeichen ungleich #, das erreicht wird. Ist es 0 oder 1, so wird es durch # ersetzt und in den Zustand q_2 bzw. q_3 gewechselt. Ist es ein Leerzeichen, so wird in den Zustand q_5 gewechselt.

q_2: M schiebt den Kopf nach rechts bis zum ersten Leerzeichen, welches sie durch 0 ersetzt. Anschließend wechselt sie in den Zustand q_4.

q_3: M schiebt den Kopf nach rechts bis zum ersten Leerzeichen, welches sie durch 1 ersetzt. Anschließend wechselt sie in den Zustand q_4.

q_4: M schiebt den Kopf nach links bis sie das erste Mal das Zeichen # erreicht. Dann wechselt sie in den Zustand q_1.

q_5: M schiebt den Kopf nach rechts, bis sie das erste Zeichen ungleich # erreicht. Alle Rauten, die sie dabei sieht, werden durch Leerzeichen ersetzt. Danach terminiert M.

Nun können wir einen Schritt weitergehen und uns überlegen, was aus diesen Eigenschaften folgt. Dazu betrachten wir das Verhalten der Turingmaschine M, wenn ihr Bandinhalt von der Form $xa\#\ldots\#y$ für $x, y \in \{0,1\}^*$ und $a \in \{0,1\}$ ist, der Kopf auf der Raute am weitesten rechts steht und die TM sich im Zustand q_1 befindet. Diese Konfiguration erreicht M wenn sie von q_0 in den Zustand q_1 wechselt (zu diesem Zeitpunkt entspricht xa noch der Eingabe, y ist das leere Wort und genau eine Raute steht auf dem Band). In den folgenden Abbildungen ist das Verhalten der Turingmaschine M in dieser Konfiguration dargestellt (links für $a = 0$ und rechts für $a = 1$). Dabei ist nicht jeder einzelne Schritt eingezeichnet, sondern nur diejenigen, in denen Zustandswechsel erfolgen.

$$
\begin{array}{ll}
\overset{q_1}{x_1 \cdots x_n\ 0\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ \square} & \overset{q_1}{x_1 \cdots x_n\ 1\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ \square} \\
\overset{q_1}{x_1 \cdots x_n\ 0\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ \square} & \overset{q_1}{x_1 \cdots x_n\ 1\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ \square} \\
\overset{q_2}{x_1 \cdots x_n\ \#\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ \square} & \overset{q_3}{x_1 \cdots x_n\ \#\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ \square} \\
\overset{q_2}{x_1 \cdots x_n\ \#\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ \square} & \overset{q_3}{x_1 \cdots x_n\ \#\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ \square} \\
\overset{q_4}{x_1 \cdots x_n\ \#\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ 0} & \overset{q_4}{x_1 \cdots x_n\ \#\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ 1} \\
\overset{q_4}{x_1 \cdots x_n\ \#\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ 0} & \overset{q_4}{x_1 \cdots x_n\ \#\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ 1} \\
\overset{q_1}{x_1 \cdots x_n\ \#\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ 0} & \overset{q_1}{x_1 \cdots x_n\ \#\ \#\ \cdots\ \#\ \#\ y_1 \cdots y_m\ 1}
\end{array}
$$

Nach den dargestellten Schritten befindet sich die Turingmaschine wieder in fast derselben Konfiguration wie zu Beginn. Der einzige Unterschied ist, dass der Bandinhalt nun von der Form $x\#\ldots\#ya$ ist. Das heißt, der letzte Buchstabe auf der linken Seite wurde an die rechte Seite angehängt. Dies wird solange wiederholt, bis die linke Seite leer ist. Anschließend werden die Rauten im Zustand q_5 gelöscht. Danach steht der Kopf auf dem ersten Zeichen der rechten Seite und die Turingmaschine terminiert. Aus diesen Überlegungen folgt, dass M die Funktion $f(w) = w^R$ berechnet. Sie dreht also die Eingabe um.

Der Leser sollte als Übung Turingmaschinen für andere einfache Probleme entwerfen. Natürlich ist es nicht Sinn und Zweck dieser Vorlesung, zu lernen, wie man Turingmaschinen konstruiert. Es wird vermutlich nur sehr wenigen Spaß machen, eine Turingmaschine für das Zusammenhangsproblem oder andere nicht triviale Probleme zu konstruieren (auch wenn dies möglich wäre). Das Spielen mit dem Modell der Turingmaschine dient an dieser Stelle dazu, ein Gefühl dafür zu entwickeln, wie mächtig Turingmaschinen sind und wie man prinzipiell damit Probleme lösen und Funktionen berechnen kann.

Wir passen Definition 1.2 nun noch an Entscheidungsprobleme und Sprachen an. Bei einem solchen Problem besteht die Ausgabe immer aus genau einem Zeichen (0 oder 1), weshalb uns nur das Zeichen interessiert, das am Ende der Berechnung an der Kopfposition steht. Dies ist in folgender Definition formalisiert.

Definition 1.3. *Eine Turingmaschine M* akzeptiert *(oder* verwirft*) eine Eingabe $w \in \Sigma^*$, wenn sie bei Eingabe w terminiert und ein Wort ausgibt, das mit* 1 *(bzw.* 0*) beginnt.*

Eine Turingmaschine M entscheidet *eine Sprache $L \subseteq \Sigma^*$, wenn sie jedes Wort $w \in L$ akzeptiert und jedes Wort $w \in \Sigma^* \setminus L$ verwirft.*

Eine Sprache $L \subseteq \{0,1\}^*$ heißt entscheidbar*, wenn es eine Turingmaschine M gibt, die L entscheidet. Wir sagen dann, dass M* eine Turingmaschine für die Sprache L ist*. Eine solche Turingmaschine terminiert insbesondere auf jeder Eingabe.*

Wir betrachten als Beispiel eine Turingmaschine $M = (Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$ mit $Q = \{q_0, q_1, q_2, \bar{q}\}$, $\Sigma = \{0, 1\}$ und $\Gamma = \{0, 1, \square\}$. Die Zustandsüberführungsfunktion δ ist in der folgenden Tabelle dargestellt.

	q_0	q_1	q_2
0	$(q_1, 0, R)$	$(q_1, 0, R)$	$(q_1, 0, R)$
1	$(q_0, 1, R)$	$(q_2, 1, R)$	$(q_0, 1, R)$
$\square$	$(\bar{q}, 0, N)$	$(\bar{q}, 0, N)$	$(\bar{q}, 1, N)$

Man beobachtet zunächst leicht, dass die Turingmaschine in jedem Schritt, in dem sie nicht in den Endzustand wechselt, den Kopf nach rechts bewegt und den Bandinhalt nicht ändert. Die Turingmaschine terminiert, sobald sie das erste Leerzeichen (also das Ende der Eingabe) erreicht hat. Die Turingmaschine verhält sich also wie ein endlicher Automat, der die Eingabe Zeichen für Zeichen von links nach rechts liest und dabei Zustandsübergänge durchführt. Sie akzeptiert die Eingabe genau dann, wenn sie das erste Leerzeichen im Zustand q_2 erreicht. Ansonsten verwirft sie die Eingabe. Betrachtet man die Zustandsübergänge, so sieht man, dass Zustand q_2 genau dann erreicht wird, wenn der bisher gelesene Teil der Eingabe mit 01 endet. Daraus folgt insgesamt, dass die Turingmaschine M genau die Wörter akzeptiert, die mit 01 enden. Alle anderen Wörter verwirft sie.

Das Modell der Turingmaschine wurde bereits 1937 von dem britischen Mathematiker Alan Turing eingeführt [4]. Turing geht in dieser wegweisenden Arbeit der Frage nach, welche Funktionen $f\colon \mathbb{N} \to \{0, 1\}$ von Computern berechnet werden können. Allerdings hatte der Begriff „Computer“ damals eine andere Bedeutung als heute. Man verstand darunter einen Menschen, der mathematische Berechnungen gemäß fester Regeln durchführt, also gewissermaßen einen Menschen, der einen Algorithmus von Hand durchführt. Solche menschlichen Computer wurden einige Jahrhunderte lang für aufwendige Berechnungen in Wissenschaften wie beispielsweise der Astronomie eingesetzt. Letztlich spielt es bei der Frage, welche Funktionen durch Algorithmen berechnet werden können, aber keine Rolle, ob die Algorithmen von einem Menschen oder einem elektronischen Computer ausgeführt werden.

Techniken zum Entwurf von Turingmaschinen

Um ein besseres Gefühl dafür zu bekommen, wie man prinzipiell Turingmaschinen für nicht triviale Probleme entwerfen kann, ist es hilfreich, sich zu überlegen, wie man grundlegende Programmiertechniken auf Turingmaschinen anwenden kann.

1. Eine Variable, die Werte aus einer endlichen Menge annehmen kann, kann in der Zustandsmenge einer Turingmaschine gespeichert werden. Möchte man beispielsweise eine Variable realisieren, die für ein festes $k \in \mathbb{N}$ Werte aus der Menge $\{0, \ldots, k\}$ annehmen kann, so kann man die Zustandsmenge Q zu der Menge $Q' = Q \times \{0, \ldots, k\}$ erweitern. Ein Zustand aus Q' ist dann ein Paar (q, a),

wobei $q \in Q$ den eigentlichen Zustand der Turingmaschine beschreibt und $a \in \{0, \ldots, k\}$ den Wert der Variablen. Die Zustandsüberführungsfunktion kann diesen Wert berücksichtigten und ihn gegebenenfalls ändern. Es ist allerdings wichtig, dass k konstant ist, da die erweiterte Zustandsmenge Q' endlich sein muss. Das bedeutet insbesondere, dass wir die Variable nicht dazu einsetzen können, um zum Beispiel die Länge der Eingabe zu speichern. Analog können auch endlich viele Variablen mit endlichen Wertebereichen realisiert werden.

2. Oft ist es hilfreich, wenn das Band der Turingmaschine aus verschiedenen Spuren besteht. Das bedeutet, in jeder Zelle stehen k Zeichen aus Γ, die alle gleichzeitig in einem Schritt gelesen und geschrieben werden. Ist $k \in \mathbb{N}$ eine Konstante, so kann dies dadurch realisiert werden, dass das Bandalphabet Γ zu $\Gamma' = \Sigma \cup \Gamma^k$ erweitert wird. Dies ist für endliches Γ und endliches k eine endliche Menge. Die Zustandsüberführungsfunktion kann dann entsprechend angepasst werden und in jedem Schritt die k Zeichen verarbeiten, die sich an der aktuellen Kopfposition befinden. Wir werden im Folgenden ein Zeichen $a \in \Sigma$ mit dem Element $(a, \square, \square, \ldots, \square) \in \Gamma^k$ identifizieren.

   Um den Sinn von mehreren Spuren und das Abstraktionsniveau, auf dem wir zukünftig Turingmaschinen beschreiben wollen, zu verdeutlichen, betrachten wir das Problem, zwei gegebene Zahlen in Binärdarstellung zu addieren. Sei $\Sigma = \{0, 1, \#\}$ das Eingabealphabet und sei $\text{bin}(x)\#\text{bin}(y)$ für $x \in \mathbb{N}$ und $y \in \mathbb{N}$ eine Eingabe. Die Aufgabe ist es, $\text{bin}(x+y)$ zu berechnen. Dazu konstruieren wir eine Turingmaschine mit drei Spuren und dem Bandalphabet

$$\Gamma' = \left\{ 0, 1, \#, \square, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\}.$$

   Die Turingmaschine arbeitet in mehreren Phasen. In der ersten Phase verschiebt sie die Binärdarstellungen $\text{bin}(x)$ und $\text{bin}(y)$ so, dass sie auf den ersten beiden Spuren rechtsbündig untereinander stehen (ggf. mit führenden Nullen, wenn die Darstellungen nicht dieselbe Länge haben). Die Eingabe 101#1100 wird in dieser Phase beispielsweise in den Bandinhalt

$$\ldots \square \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \square \ldots$$

   überführt. In der zweiten Phase geht die Turingmaschine die Bits von rechts nach links durch und führt die Addition bitweise nach der Schulmethode durch. Das Ergebnis wird dabei auf die dritte Spur geschrieben. Tritt in einem Schritt ein Übertrag auf, so merkt die Turingmaschine sich dies im Zustand. In dem obigen Beispiel ergibt sich nach dieser Phase der Bandinhalt

$$\ldots \square \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \square \ldots$$

In der dritten Phase wird das Ergebnis zurück in das (einspurige) Eingabealphabet Σ übersetzt. Dabei werden die Inhalte der ersten beiden Spuren gelöscht und nur der Inhalt der dritten Spur bleibt erhalten.

3. Mithilfe mehrerer Spuren können auch Variablen realisiert werden, die beliebig viele Zustände annehmen können. Diese können nicht mehr in den Zustand codiert werden, sondern sie müssen auf dem Band gespeichert werden. Man könnte zum Beispiel für jede Variable eine Spur benutzen oder mehrere durch Trennzeichen getrennte Variablen auf derselben Spur abspeichern.

4. Mit etwas Aufwand kann man beim Entwurf von Turingmaschinen auch Unterprogramme einsetzen. Dazu kann man beispielsweise für jedes Unterprogramm eine Teilmenge von Zuständen auszeichnen, die ausschließlich in diesem Unterprogramm angenommen werden. Ebenso kann man beim Aufruf eines Unterprogramms einen separaten (beispielsweise weit rechts liegenden) Speicherbereich auf dem Band reservieren, in dem die Berechnungen des Unterprogramms erfolgen. Es können dabei einige technische Probleme auftreten. Genügt der reservierte Speicherbereich beispielsweise nicht, so muss er gegebenenfalls verschoben werden. Ebenso muss ein Aufrufstack verwaltet werden. All dies ist möglich, aber relativ komplex und wenig elegant. Wir wollen die Diskussion hier nicht weiter vertiefen. Für die weiteren Betrachtungen ist lediglich wichtig zu wissen, dass Unterprogramme prinzipiell realisiert werden können.

5. Auch können for- und while-Schleifen in Turingmaschinen realisiert werden. Eine solche Schleife haben wir bereits implizit in der obigen Turingmaschine gesehen, die bei Eingabe w das gespiegelte Wort w^R berechnet. Letztendlich kann man Schleifen aber auch als eine spezielle Art von Unterprogramm auffassen, sodass sie genauso wie andere Unterprogramme realisiert werden können.

Turingmaschinen mit mehreren Bändern

Wir haben oben bereits diskutiert, dass Turingmaschinen mit mehreren Spuren durch eine einfache Anpassung des Bandalphabets simuliert werden können. Diese Maschinen besitzen aber weiterhin nur einen Lese-/Schreibkopf, der die k Zeichen unter der aktuellen Kopfposition gleichzeitig verarbeiten kann. Nun gehen wir einen Schritt weiter und betrachten Turingmaschinen mit einer konstanten Zahl an Bändern. Bei diesen Maschinen besitzt jedes Band seinen eigenen Lese-/Schreibkopf, der unabhängig von den anderen Köpfen bewegt werden kann.

Eine *k-Band-Turingmaschine* M ist ein 7-Tupel $(Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$, bei dem die ersten sechs Komponenten die gleiche Bedeutung haben wie in Definition 1.1. Die Zustandsüberführungsfunktion δ ist von der Form

$$\delta\colon (Q \setminus \{\bar{q}\}) \times \Gamma^k \to Q \times \Gamma^k \times \{L, N, R\}^k.$$

Sie erhält als Eingabe den aktuellen Zustand und die k Zeichen, die sich an den aktuellen Kopfpositionen befinden. Als Ausgabe liefert sie den Nachfolgezustand, die k

Zeichen, durch die die alten ersetzt werden sollen, und die Bewegungen der k Köpfe. Das erste Band fungiert dabei wie bei einer normalen Turingmaschine als Ein-/Ausgabeband. Die anderen Bändern sind initial leer. Eine 1-Band-Turingmaschine ist somit identisch zu einer normalen Turingmaschine gemäß Definition 1.1.

Da sich die Köpfe unabhängig bewegen können, erscheint eine einfache Simulation durch eine 1-Band-Turingmaschine nicht ohne Weiteres möglich zu sein. Wir werden nun aber beweisen, dass k-Band-Turingmaschinen stets durch normale Turingmaschinen simuliert werden können, wobei allerdings ein gewisser Zeitverlust auftritt. Um dies quantifizieren zu können, definieren wir die Begriffe Rechenzeit und Platzbedarf zunächst formal.

Definition 1.4. *Es sei M eine k-Band-Turingmaschine. Die* Rechenzeit $t_M(w)$ von M bei Eingabe w *ist die Anzahl an Zustandsübergängen, die M bei Eingabe w bis zur Terminierung durchführt. Terminiert M nicht auf w, so ist die Rechenzeit unendlich. Der* Platzbedarf $s_M(w)$ von M bei Eingabe w *ist die Anzahl (summiert über alle Bänder) an verschiedenen Zellen, auf denen sich im Laufe der Rechnung mindestens einmal ein Lese-/Schreibkopf befunden hat. Terminiert M nicht, so kann der Platzbedarf unendlich sein.*

Die Rechenzeit $t_M(n)$ von M bei Eingaben der Länge $n \in \mathbb{N}$ *ist die maximale Rechenzeit, die bei Eingaben der Länge n auftritt, also $t_M(n) = \max_{w \in \Sigma^n} t_M(w)$. Analog ist der* Platzbedarf $s_M(n)$ von M bei Eingaben der Länge n *als $s_M(n) = \max_{w \in \Sigma^n} s_M(w)$ definiert.*

Nun werden wir sehen, wie man k-Band-Turingmaschinen durch 1-Band-Turingmaschinen simulieren kann.

Theorem 1.5. *Eine k-Band Turingmaschine M mit Rechenzeit $t(n)$ und Platzbedarf $s(n)$ kann durch eine 1-Band-Turingmaschine M' mit Rechenzeit $O(t(n)^2)$ und Platzbedarf $O(s(n))$ simuliert werden.*

Beweis. Die Turingmaschine M' simuliert die Turingmaschine M Schritt für Schritt und verwendet dafür $2k$ Spuren. Nach der Simulation des t-ten Rechenschrittes von M durch M' mit $t \in \mathbb{N}_0$ sind die folgenden Invarianten erfüllt.

1. Die ungeraden Spuren $1, 3, \ldots, 2k-1$ enthalten den Inhalt der k Bänder von M.
2. Auf den geraden Spuren $2, 4, \ldots, 2k$ sind die Kopfpositionen von M mit dem Zeichen # markiert (Spur $2i$ enthält die Markierung für die Kopfposition auf Band i von M). Alle anderen Zellen auf diesen Bändern enthalten das Leerzeichen.
3. Der Kopf von M' steht an der linkesten Position, die auf einem der geraden Bänder mit # markiert ist.

Die Spuren können in konstanter Zeit initialisiert werden. Dazu muss lediglich das Zeichen # an der aktuellen Position (die zu Beginn dem ersten Zeichen der Eingabe entspricht) auf alle geraden Spuren geschrieben werden. Um nun einen Rechenschritt

von M zu simulieren, geht M' wie folgt vor. Sie schiebt den Kopf solange nach rechts, bis sie alle k Markierungen # gesehen hat. Dabei merkt sie sich im Zustand, welche k Zeichen sich an den aktuellen Kopfpositionen befinden. Nachdem sie die letzte Markierung erreicht und alle k Zeichen gelesen hat, wertet sie die Zustandsüberführungsfunktion von M aus und merkt sich im Zustand den neuen Zustand von M, durch welche Zeichen die k Zeichen an den Kopfpositionen ersetzt werden sollen und wie die Köpfe sich bewegen sollen. Anschließend schiebt sie den Kopf wieder nach links zurück und ändert dabei den Bandinhalt an den markierten Positionen und verschiebt die Markierungen entsprechend den Bewegungen von M. Sie stoppt, sobald sie alle k Markierungen gesehen und den Rechenschritt von M komplett simuliert hat. Danach ist die Invariante wieder hergestellt. Der Leser mache sich noch einmal klar, dass M' sich bei dieser Konstruktion nur endlich viel im Zustand merken muss.

Die Anzahl Schritte, die M' benötigt, um einen Schritt von M zu simulieren, ist proportional zu dem Abstand, den die beiden Markierungen am weitesten links und am weitesten rechts voneinander haben, da M' diese Entfernung zweimal zurücklegen muss. Da die Laufzeit von M durch $t(n)$ beschränkt ist, können diese beiden Markierungen zu jedem Zeitpunkt um maximal $2t(n)$ viele Positionen auseinander liegen (schlimmstenfalls geht ein Kopf von M in jedem Schritt nach links und ein anderer geht in jedem Schritt nach rechts). Jeder einzelne der $t(n)$ vielen Schritte kann somit in Zeit $O(t(n))$ simuliert werden, woraus direkt die behauptete Rechenzeit von $O(t(n)^2)$ folgt. Dass der Platzbedarf von M' in derselben Größenordnung wie der von M liegt, folgt direkt daraus, dass M' nur dann eine Zelle besucht, wenn M auf einem seiner Bänder ebenfalls diese Zelle besucht. □

1.2.2 Registermaschinen

Wir haben uns bereits im letzten Semester mit dem Modell der *Registermaschine* beschäftigt, das an eine rudimentäre Assemblersprache erinnert, die auf die wesentlichen Befehle reduziert wurde. In diesem Modell steht als Speicher eine unbegrenzte Anzahl an Registern zur Verfügung, die jeweils eine natürliche Zahl enthalten und auf denen grundlegende arithmetische Operationen durchgeführt werden können. Die Inhalte zweier Register können addiert, subtrahiert, multipliziert und dividiert werden. Ebenso können Registerinhalte kopiert werden und Register können mit Konstanten belegt werden. Darüber hinaus unterstützen Registermaschinen unbedingte Sprungoperationen (GOTO) und bedingte Sprungoperationen (IF), wobei als Bedingung nur getestet werden darf, ob ein Register kleiner, gleich oder größer als ein anderes Register ist.

Formal besteht ein Registermaschinenprogramm aus einer endlichen durchnummerierten Folge von Befehlen, deren Syntax in Tabelle 1.1 zusammengefasst ist. Wird ein solches Programm ausgeführt, so wird ein Befehlszähler b verwaltet, der angibt, welcher Befehl als nächstes ausgeführt werden soll. Zu Beginn wird $b = 1$ gesetzt. Jeder Befehl ändert den Befehlszähler (ist es kein Sprungbefehl, so wird b lediglich um eins erhöht) und gegebenenfalls den Inhalt der Register. Wir bezeichnen mit $c(0), c(1), c(2), \ldots$ die Inhalte der Register. Dabei gilt stets $c(i) \in \mathbb{N}_0$. Zu Beginn steht in dem Register $c(0)$ die Eingabe und für alle anderen Register gilt $c(i) = 0$. Die Ausgabe findet sich am

Syntax	Zustandsänderung	Änderung von b
LOAD i	$c(0) := c(i)$	$b := b+1$
CLOAD i	$c(0) := i$	$b := b+1$
INDLOAD i	$c(0) := c(c(i))$	$b := b+1$
STORE i	$c(i) := c(0)$	$b := b+1$
INDSTORE i	$c(c(i)) := c(0)$	$b := b+1$
ADD i	$c(0) := c(0) + c(i)$	$b := b+1$
CADD i	$c(0) := c(0) + i$	$b := b+1$
INDADD i	$c(0) := c(0) + c(c(i))$	$b := b+1$
SUB i	$c(0) := c(0) - c(i)$	$b := b+1$
CSUB i	$c(0) := c(0) - i$	$b := b+1$
INDSUB i	$c(0) := c(0) - c(c(i))$	$b := b+1$
MULT i	$c(0) := c(0) \cdot c(i)$	$b := b+1$
CMULT i	$c(0) := c(0) \cdot i$	$b := b+1$
INDMULT i	$c(0) := c(0) \cdot c(c(i))$	$b := b+1$
DIV i	$c(0) := \lfloor c(0)/c(i) \rfloor$	$b := b+1$
CDIV i	$c(0) := \lfloor c(0)/i \rfloor$	$b := b+1$
INDDIV i	$c(0) := \lfloor c(0)/c(c(i)) \rfloor$	$b := b+1$
GOTO j	-	$b := j$
IF $c(0) = x$ GOTO j	-	$b := \begin{cases} j \text{ falls } c(0) = x \\ b+1 \text{ sonst} \end{cases}$
IF $c(0) < x$ GOTO j	-	$b := \begin{cases} j \text{ falls } c(0) < x \\ b+1 \text{ sonst} \end{cases}$
IF $c(0) \leq x$ GOTO j	-	$b := \begin{cases} j \text{ falls } c(0) \leq x \\ b+1 \text{ sonst} \end{cases}$
END	Ende der Rechnung	-

Tabelle 1.1: Syntax und Semantik der Befehle einer Registermaschine

Ende der Rechnung ebenfalls in Register $c(0)$. In Tabelle 1.1 ist auch die Semantik der einzelnen Befehle aufgeführt.

In einer Registermaschine kann jedes Register eine beliebig große natürliche Zahl enthalten. Im letzten Semester haben wir das *uniforme Kostenmaß* zugrunde gelegt, bei dem die Ausführung jedes Befehls unabhängig von der Größe der Zahlen eine Zeiteinheit benötigt. Werden in den Registern große Zahlen gespeichert, so ist es realistischer das *logarithmische Kostenmaß* einzusetzen. Bei diesem ist die Laufzeit eines Befehls proportional zu der Länge der Zahlen in den angesprochenen Registern in Binärdarstellung. Die Laufzeit eines Befehl ist also proportional zum Logarithmus der beteiligten Zahlen.

Analog zu Definition 1.4 interessiert uns auch bei Registermaschinen die Laufzeit in Abhängigkeit von der Eingabegröße. Diese messen wir im logarithmischen Kostenmaß durch die Länge der Binärdarstellung der Eingabe in Register $c(0)$. Wir nennen eine Registermaschine im logarithmischen Kostenmaß $t(n)$-*zeitbeschränkt*, wenn die Laufzeit im logarithmischen Kostenmaß für jedes $n \in \mathbb{N}$ und jede Eingabe mit n Bits

durch $t(n)$ beschränkt ist.

Um die Mächtigkeit des Modells der Turingmaschine zu verdeutlichen, zeigen wir nun, dass Registermaschinen durch Turingmaschinen simuliert werden können und andersherum. Da wir Registermaschinen bereits im letzten Semester als realistisches Modell realer Rechner akzeptiert haben, bedeutet dies, dass auch Turingmaschinen ein realistisches Modell darstellen.

Theorem 1.6. *Jede im logarithmischen Kostenmaß $t(n)$-zeitbeschränkte Registermaschine kann durch eine Turingmaschine simuliert werden, deren Rechenzeit $O(q(n + t(n)))$ für ein Polynom q beträgt.*

Wir werden das Theorem nicht beweisen. Der Beweis ist zwar nicht schwer, aber wenig erhellend. Man kann dazu eine Turingmaschine mit 2-Bändern konstruieren, die für jede Zeile des Registermaschinenprogramms ein Unterprogramm enthält. Die Unterprogramme werden dabei auf dem ersten Band ausgeführt, während das zweite Band die Inhalte der Register in Binärdarstellung enthält. Die Turingmaschine simuliert dann Schritt für Schritt das Verhalten der Registermaschine. Die genaue Konstruktion dieser Turingmaschine ist allerdings sehr technisch.

Theorem 1.6 sagt nicht nur, dass jede Registermaschine durch eine Turingmaschine simuliert werden kann, sondern auch, dass die Laufzeit der entsprechenden Turingmaschine nur polynomiell größer ist als die der Registermaschine. Insbesondere ist die Laufzeit der Turingmaschine polynomiell, wenn die Laufzeit der Registermaschine polynomiell ist. Um dies einzusehen, nehmen wir an, dass die Laufzeit $t(n)$ der Registermaschine ein Polynom vom Grad $d \geq 1$ ist. Laut dem Theorem existiert ein Polynom q, dessen Grad wir mit $d^\star$ bezeichnen, sodass die Rechenzeit der Turingmaschine durch $O(q(n+t(n))) = O(q(t(n))) = O(q(n^d)) = O(n^{dd^\star})$ beschränkt ist. Damit ist auch die Laufzeit der Turingmaschine durch ein Polynom beschränkt.

Wir werden später in dieser Vorlesung noch ausführlich über die Bedeutung polynomieller Laufzeit sprechen. An dieser Stelle sollte der Leser die Erkenntnis mitnehmen, dass jedes Problem, das von einer Registermaschine im logarithmischen Kostenmaß in polynomieller Zeit gelöst werden kann, auch von einer Turingmaschine in polynomieller Zeit gelöst werden kann.

Auch umgekehrt kann jede Turingmaschine mit polynomiellem Zeitverlust durch eine Registermaschine im logarithmischen Kostenmaß simuliert werden kann.

Theorem 1.7. *Jede Turingmaschine, deren Rechenzeit durch $t(n)$ beschränkt ist, kann durch eine im logarithmischen Kostenmaß $O((t(n) + n)\log(t(n) + n))$-zeitbeschränkte Registermaschine simuliert werden.*

Zusammen implizieren die Theoreme 1.6 und 1.7, dass die Klasse der von Turingmaschinen berechenbaren Funktionen und die Klasse der von Registermaschinen berechenbaren Funktionen überein stimmen. Darüber hinaus sind sogar die Rechenzeiten bis auf polynomielle Faktoren vergleichbar. Etwas präziser formuliert besagen die Theoreme, dass die Klasse der von Turingmaschinen in polynomieller Zeit berechenbaren Funktionen und die Klasse der von Registermaschinen in polynomieller Zeit berechenbaren Funktionen überein stimmen.

1.2.3 Die Church-Turing-These

In Definition 1.2 haben wir festgelegt, dass eine Funktion berechenbar ist, wenn es eine Turingmaschine gibt, die zu jeder Eingabe in endlich vielen Schritten die durch die Funktion beschriebene Ausgabe liefert. Der Begriff der Berechenbarkeit ist demnach zunächst an das Modell der Turingmaschine gekoppelt. Dies ist uns aber nicht allgemein genug, denn haben wir für eine Funktion nachgewiesen, dass sie nicht berechenbar ist, so wäre es zunächst ja durchaus denkbar, dass sie zwar nicht von einer Turingmaschine, aber von einem Java- oder C++-Programm berechnet werden kann. Geht auch dies nicht, so kann man die Funktion vielleicht mithilfe anderer Hardware (Quantencomputer, Analogrechner, etc.) berechnen. Falls dies wirklich der Fall wäre, so wäre ein Begriff der Berechenbarkeit, der sich wie in Definition 1.2 nur auf Turingmaschinen stützt, nicht besonders interessant.

Tatsächlich haben wir aber bereits in Theorem 1.6 gesehen, dass Registermaschinen nicht mächtiger sind als Turingmaschinen. Registermaschinen wiederum sind ein realistisches Modell für heutige reale Rechner (unabhängig von der benutzten Programmiersprache). Ist eine Funktion also auf einem realen Rechner berechenbar, so ist sie auch auf einer Registermaschine und damit auch auf einer Turingmaschine berechenbar. Andersherum formuliert, kann eine Funktion, die nicht von einer Turingmaschine berechnet werden kann, auch von keinem heutigen realen Rechner berechnet werden.

Darüber hinaus gibt es neben Turingmaschinen und Registermaschinen eine ganze Reihe von weiteren theoretischen Modellen, mit denen man versucht hat, den Begriff der Berechenbarkeit zu formalisieren (WHILE-Programme, μ-rekursive Funktionen, etc.). Diese Modelle wurden unabhängig voneinander entworfen, es hat sich aber im Nachhinein herausgestellt, dass sie alle zur selben Klasse von berechenbaren Funktionen führen. Dies sind starke Indizien für die Vermutung, dass alle Funktionen, die man auf irgendeine algorithmische Art berechnen kann, auch von Turingmaschinen berechnet werden können. Dies ist die sogenannte *Church-Turing-These*.

These 1.8 (Church-Turing-These)**.** *Alle „intuitiv berechenbaren“ Funktionen können von Turingmaschinen berechnet werden.*

Diese These ist prinzipiell nicht beweisbar, da der Begriff „intuitiv berechenbar“ nicht formal definiert ist. Eine formalere Variante dieser These, die zwar prinzipiell beweisbar, aber noch unbewiesen ist, lautet wie folgt.

These 1.9 (Physikalische Church-Turing-These)**.** *Die Gesetze der Physik erlauben es nicht, eine Maschine zu konstruieren, die eine Funktion berechnet, die nicht auch von einer Turingmaschine berechnet werden kann.*

Auch für diese These gibt es eine ganze Reihe von Indizien. Beispielsweise weiß man, dass weder mit Quantencomputern noch mit Analogrechnern Funktionen berechnet werden können, die man nicht auch mit einer Turingmaschine berechnen kann. Tatsächlich gibt es mittlerweile unter dem Stichwort „unconventional computing“ ein ganzes Kuriositätenkabinett an physikalischen, biologischen und chemischen Modellen, mit denen Berechnungen durchgeführt werden können. Keines der bekannten Modelle widerspricht aber der oben formulierten physikalischen Church-Turing-These.

Literaturverzeichnis

[1] Norbert Blum: **Einführung in formale Sprachen, Berechenbarkeit, Informations- und Lerntheorie**. Oldenbourg, 2007.

[2] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman: **Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit**. Pearson Studium, 3. Auflage, 2007.

[3] Christos H. Papadimitriou: **Computational Complexity**. Addison Wesley, 1993.

[4] Alan M. Turing: **On Computable Numbers, with an Application to the Entscheidungsproblem**. Proceedings of the London Mathematical Society, 42:230–265, 1937.

[5] Berthold Vöcking: **Berechenbarkeit und Komplexität**, Vorlesungsskript, RWTH Aachen, Wintersemester 2013/14. `http://algo.rwth-aachen.de/Lehre/WS1314/VBuK/BuK.pdf`.

[6] Ingo Wegener: **Theoretische Informatik – eine algorithmenorientierte Einführung**. Teubner, 3. Auflage, 1993.