ETH Zürich

Schweizer Tag für den Informatikunterricht 2013

Freitag, 11. Januar 2013, Kantonsschule Limmattal(Urdorf)

Workshop:
„Berechenbarkeit, Entscheidbarkeit, Komplexität“

Referent: Armin P. Barth

Stichworte zum Inhalt:
Turing-Maschine, rekursive Funktionen, These von Church, Entscheidbarkeit, busy beaver, Halteproblem und die Methode der Reduktion, NP-Vollständigkeit, Komplexität mathematischer Theorien

Armin P. Barth unterrichtet Mathematik an der Kantonsschule Baden und arbeitet am MINT-Lernzentrum der ETH Zürich (Institut für Lehr- und Lernforschung). Daneben schreibt er seit 2008 die Kolumne „Café Mathe“ in den AZ Medien und seit 2011 die Kolumne „Schwamm drüber“ in der Zeitschrift „Bildung Schweiz“. Herr Barth hat diverse Artikel und Bücher publiziert zu den Themen Mathematik und Didaktik. 2003 erschien beim Vieweg-Verlag das Buch „Algorithmik für Einsteiger“, aktuell arbeitet er an einem neuen Buch über Algorithmik im Auftrag des Springer-Verlages.
Siehe auch: www.armin-p-barth.ch

Januar 2013

Einleitung

(…)

Llull, Leibniz, Hilbert und algorithmische Träume

Ramon Llull, ein mallorquinischer Philosoph, hatte sich eine Aufgabe gestellt, die selbst Helden überfordert. Als er 1232 zur Welt kam, standen grosse Teile Spaniens seit 500 Jahren unter muslimischer Herrschaft. Die Rückeroberung durch die Christen war zwar in vollem Gange, aber noch immer lebten Anhänger aller drei monotheistischen Religionen im gleichen Land und begegneten einander oft verständnislos. Llull fasste den Plan, Antworten zu finden auf die Frage, wie die Verständigung zwischen Vertretern verschiedener Religionen verbessert werden konnte, ein Anliegen, das heute nicht weniger aktuell ist. Dazu erfand er eigens die „Ars magna" (grosse Kunst, vgl. Abb.) Es handelte sich dabei um eine formale Sprache, in der die Grundbegriffe und –werte, die alle drei Religionen teilten, formalisiert werden und in der durch mechanisches Manipulieren dieser Terme, also durch einen Algorithmus, neue, gemeinsam anerkannte Erkenntnisse und Antworten auf Streitfragen gewissermassen „berechnet" werden sollten.

Die Idee war bestechend: Wenn man Wahrheit oder Falschheit von Aussagen rechnerisch entscheiden kann, dann kann es keinen Streit mehr geben. Im 17. Jahrhundert sollte Gottfried Wilhelm Leibniz ein noch gewagteres Unternehmen starten: Er wollte eine formale Sprache (characteristica universalis) schaffen, in der sich gedankliche Prozesse einer metaphysischen Disziplin ebenso ausdrücken lassen wie diejenigen der Logik und Mathematik, so dass dann Rechenschritte, oder besser: formale Manipulationen von Zeichen und Zeichenketten, ein für allemal über Wahrheit oder Falschheit von Aussagen entscheiden konnten. Es ist nicht verwunderlich, dass ein so kühnes Unterfangen ebenso wenig erfolgreich sein konnte wie der Plan von Ramon Llull.

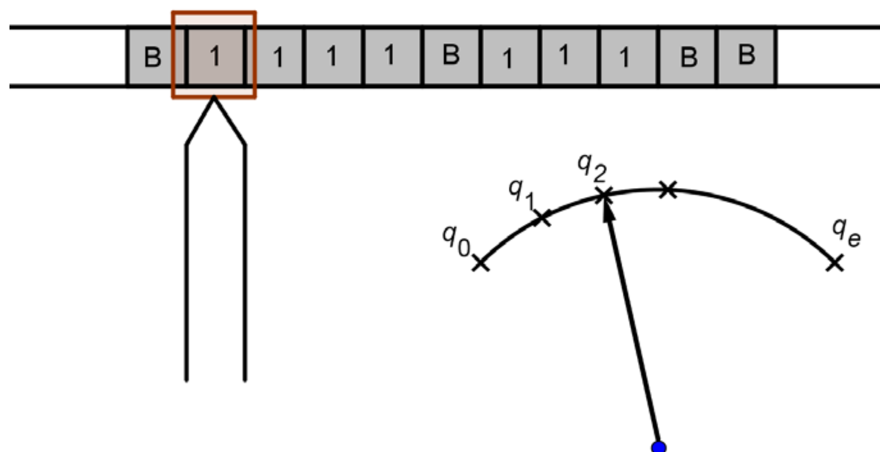
Zu Beginn der 20. Jahrhunderts wurde die Frage nach der formalen Berechenbarkeit von Wahrheiten eingeschränkt auf die Mathematik. David Hilbert hatte es 1900 angeregt, und Heinrich Behmann bezeichnete es als „Hauptproblem der modernen Logik": Kann man allein durch formales Manipulieren mit Zeichen nach gewissen Regeln, also mit einem Algorithmus, all das herleiten, was gilt? Sind Berechenbarkeit und Wahrheit dasselbe? Mathematikern wie Kurt Gödel und Alan Turing (der im Jahr 2012 seinen 100. Geburtstag hätte feiern können), ist es zu verdanken, dass man dazu bahnbrechende Antworten fand: Formale Kalküle und Algorithmen haben immer einen blinden Fleck; sie können niemals all das erzeugen, was gilt.

Um aber nachweisen und verstehen zu können, dass gewisse Aufgaben nicht algorithmisch lösbar sind, muss präzise definiert werden, was ein Algorithmus ist. Denn nur dann kann man auch schlüssig beweisen, dass kein solcher existieren kann.

Turing-Maschinen (TM)

Diese Maschinen dienen dazu, den bloss vagen Begriff „berechenbar" durch den mathematisch präzisen Begriff „TM-berechenbar" zu ersetzen. Natürlich müssen wir dazu einsehen, dass „TM-Berechenbarkeit" mit unserer intuitiven Vorstellung von „Berechenbarkeit" übereinstimmt.

Eine TM besteht aus
- einem Lese- und Schreibband aus theoretisch unendlich vielen Feldern, auf dem mit Hilfe eines Alphabetes eine Zeichenfolge schreibbar und lesbar ist,
- einem Lese- und Schreibkopf, der immer genau ein Feld aufs Mal lesen und/oder beschreiben kann,
- einer endlichen Menge von Zuständen,
- und einem Programm, das zu jedem Zeitpunkt genau festlegt, welcher Schritt als nächstes ausgeführt wird.



Definition:
Eine **Turing-Maschine** (TM) ist ein 3-Tupel (A, Q, δ), wobei
- A ein endliches **Alphabet** von Eingabe- und Ausgabezeichen ist, das das ausgezeichnete Element B (Blank) enthält. Meist ist $A = \{\text{B}, 1\}$ oder $A = \{\text{B}, 0, 1\}$.
- Q eine endliche Menge von **Zuständen** ist mit zwei ausgezeichneten Elementen q_0 (Startzustand) und q_e (Endzustand).
- das **Programm** $\delta : Q \times A \to Q \times A \times \{\text{L, R, S}\}$ eine partielle Funktion ist. Ist die Maschine in einem bestimmten Zustand und liest sie ein bestimmtes Zeichen, so gibt dieses Programm an, in welchen neuen Zustand sie wechseln soll, welches Zeichen sie drucken soll und welche Bewegung (L = Links, R = Rechts, S = Stopp) sie ausführen soll.

TM können sog. **zahlentheoretische Funktionen** berechnen, das heisst, Inputs aus $\mathbb{N}^n$, die auf dem Band notiert sind, zu Outputs aus $\mathbb{N}$ verarbeiten. Sie können aber auch sog. **Entscheidungsprobleme** lösen. Bei einem letzteren wird auf dem Band eine Zahl i präsentiert, und die Maschine geht in den speziellen Zustand q_a (akzeptierender Zustand) über, falls diese Zahl eine bestimmte Eigenschaft erfüllt, und sonst in den speziellen Zustand q_n (verneinender Zustand).

Beispiel: Der TM wird eine natürliche Zahl n präsentiert, und die Maschine entscheidet, ob diese Zahl prim ist oder nicht.

Beispiele für Turing-Maschinen:

- Addition
- Erhöhung des Inputs um 1
- Folge von Einsen in Dezimalzahl umwandeln
- Multiplikation
- Euklidischer Algorithmus
- Usw.

Für alles, was im intuitiven Sinne des Wortes berechenbar ist, konnte und kann eine entsprechende TM hergestellt werden. Das nährt natürlich die Vermutung, dass Turing-Maschinen **genau** das abdecken, was man unter berechenbaren Prozessen versteht. Zudem haben sich auch andere Definitionen von Algorithmus („λ-Definability“ von Church, „rekursive Funktion“ von Gödel und Kleene, usw.) als äquivalent zur TM herausgestellt.

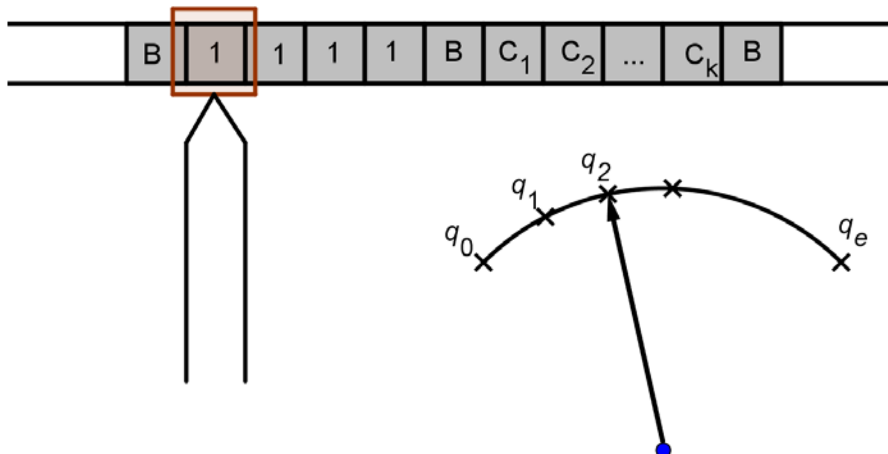
These von Alonzo Church (1936):

Alles, was im intuitiven Sinne berechenbar ist, kann von einer TM berechnet werden.

Codierung und Universalmaschine

Bisher kann eine TM immer nur eine Aufgabe lösen. Gibt es auch eine TM, die „multifunktionell“ ist? Ja, in der Tat! Dazu stellen wir uns einfach vor, dass auf dem Band einer UM (Universalmaschine) genannten Maschine nebst dem zu verarbeitenden Input auch noch der Code eines TM-Programms einer anderen TM steht.

Die UM decodiert dann zuerst das Programm der durch den Code $C_1C_2...C_k$ dargestellten TM und wendet dann das „Fremdprogramm“ auf den Input an. Auf diese Weise kann die UM jede andere TM simulieren. Entscheidend dabei ist, dass man versteht, dass kein Unterschied besteht zwischen einem Programm und einem Datum! Programme und Daten werden gleich behandelt, und diese Idee war eine der zentralen Ideen, die die Erfindung des modernen Computers befeuert haben.



Allerdings muss dazu noch geklärt werden, wie man genau ein TM-Programm durch eine einzige Zahl codiert. Dafür gibt es mehrere Möglichkeiten. Zunächst wird das Programm δ einer TM so umgeordnet und ergänzt, dass die Prämissen in standardisierter Reihenfolge vorliegen, nämlich zuerst alle Paare mit erstem Eintrag q_0, dann alle Paare mit erstem Eintrag q_1 und so weiter, und innerhalb jeder Gruppe alphabetisch nach dem zweiten Eintrag (dem Zeichen) sortiert. Da die Prämissen damit festgelegt sind, braucht man nur noch die Konklusionen zu codieren.

Eine typische Konklusion hat die Form (q_i, Z_j, B_k). Wir codieren also einfach so:

Code der Konklusion (q_i, Z_j, B_k) sei: $2^i \cdot 3^j \cdot 5^k$.

Ist jede Konklusion codiert, kann man auch das ganze Programm durch eine einzige Zahl codieren durch:

$$\text{Code}(\delta) := 2^{\text{Code}(C_1)} \cdot 3^{\text{Code}(C_2)} \cdot \ldots \cdot p_n^{\text{Code}(C_n)}$$

Alternativ könnte man auch die folgende rekursiv definierte Codierung benutzen:

$$\sigma_2(a,b) := 2^a \cdot (2b+1) - 1$$
$$\sigma_3(a,b,c) := \sigma_2(\sigma_2(a,b),c)$$
$$usw.$$

Codierungen dieser Art werden oft **Gödelnummerierung** (GN) genannt.

Berechenbare Funktionen

Wie viele Funktionen gibt es eigentlich? Wir beschränken uns hier einmal auf Funktionen der Art $f : \mathbb{N} \to \mathbb{N}$.

Solche Funktionen gibt es bestimmt überabzählbar viele vermöge folgender Injektionen:

$$\begin{aligned} \varphi : [0\,,1[\quad &\to \quad \{f : \mathbb{N} \to \mathbb{N}\} \\ 0.a_1a_2a_3... &\mapsto \quad f : f(i) = a_i \end{aligned}$$

und

$$\begin{aligned} \psi : \{f : \mathbb{N} \to \mathbb{N}\} &\to [0\,,1[\\ f \qquad &\mapsto 0.\underbrace{11...1}_{f(0)+1}0\underbrace{11...1}_{f(1)+1}0\underbrace{11...1}_{f(2)+1}0... \end{aligned}$$

Es gibt aber höchstens abzählbare viele Algorithmen, da es nur abzählbar viele TM-Programme gibt. (Solche lassen sich ja durch natürliche Zahlen codieren!)

Schon aus Mächtigkeitsgründen muss es also nicht-berechenbare Funktionen geben.

Beispiele:

1.) $f : j \mapsto j$-te Ziffer in der Dezimalbruchentwicklung von π
ist natürlich berechenbar, da Pi beliebig genau bestimmt werden kann.

2.) $g : j \mapsto \begin{cases} 1, \text{ falls die Ziffernfolge } j \text{ in } \pi \text{ vorkommt} \\ 0, \text{ sonst} \end{cases}$
Wir wissen nicht, ob g berechenbar ist oder nicht. Taucht eine Ziffernfolge lange nicht auf, wissen wir ja nicht, wie weit wir noch suchen müssen.

3.) $h : j \mapsto \begin{cases} 1, \text{ falls jede gerade Zahl } \geq 4 \text{ Summe zweier Primzahlen ist} \\ 0, \text{ sonst} \end{cases}$
Diese Funktion ist natürlich berechenbar, entweder mit dem Algorithmus „Ordne jedem j die Zahl 1 zu" oder mit dem Algorithmus „Ordne jedem j die Zahl 0 zu". Wir wissen bloss nicht, mit welchem, jedenfalls so lange nicht, bis die Goldbach-Vermutung gelöst ist.

Eine Funktion $\begin{cases} f : \mathbb{N} \to \mathbb{N} \\ \quad x \mapsto f(x) \end{cases}$ heisst *TM-berechenbar*, gewenn es eine TM M_f gibt, die bei Input $x \in \mathbb{N}$ auf dem Band in endlich vielen Schritten $f(x)$ berechnet und dann im Zustand q_e anhält.

Wichtige Folgerung:
Es ist nicht dasselbe,

- eine beliebige Zuordnung $\mathbb{N} \to \mathbb{N}$ zu verlangen – oder
- eine Zuordnung $\mathbb{N} \to \mathbb{N}$ zu verlangen, die sich überdies durch eine endlich formulierbare Anweisung (etwa eine Formel) realisieren lässt.

Die 2. Forderung ist viel stärker! Damit eine TM f berechnen kann (und zwar in endlich vielen Schritten), muss eine endlich formulierbare Anweisung existieren, und solche gibt es nur abzählbar viele!

Es bleiben aber unendlich viele Funktionen übrig, die nicht durch eine endliche Anweisung definiert werden können.

Fleissige Biber sind Spielverderber

Nun könnte man denken, dass nicht-berechenbare Funktionen in der Praxis gar nicht vorkommen und auch nicht konkret angegeben werden können. Weit gefehlt. Tibor Rado hat 1962 als erster eine Funktion angegeben, die nicht TM-berechenbar ist:

Definition:
Die Funktion *bb* (busy beaver) ordnet jeder natürlichen Zahl *n* die maximale Anzahl Einsen zu, die eine TM mit genau *n* Zuständen zusammenhängend auf ein leeres Band schreiben kann, bevor sie anhält.

Es ist klar, dass schon für kleine Zahlen *n* die Funktionswerte nicht praktisch bestimmbar sind, denn man müsste für eine irrsinnig grosse Zahl von Maschinen nachweisen, dass sie anhalten oder eben nicht. Was man heute weiss:

$bb(1) = 1$ (Rado, 1962)

$bb(2) = 4$ (Rado, 1962)

$bb(3) = 6$ (Lin, Rado, 1965)

$bb(4) = 13$ (Weimann, Casper, Fenzel, 1972)

$bb(5) \geq 4098$ (Buntrock, Marxen, 1989)

$bb(6) \geq 3.514 \cdot 10^{18267}$ (Kropitz, 2010)

...

$bb(12) \geq 6 \cdot 4096^{\left[4096^{\left[4096^{\cdots\left[4096^{4}\right]}\right]}\right]}$ (mit 166 Zahlen 4096)

Eine einfache Variante von C. Dunham (1965):

Sei $D(n)$ die maximale Anzahl Einsen, die eine TM mit *n* Zuständen schreiben kann, wenn sie auf ein Band mit einem Einsen-Block der Länge *n* angesetzt wird.

Wäre D TM-berechenbar, so gäbe es auch eine TM M, die die Funktion $n \mapsto D(n)+1$ berechnet. Angenommen, M habe *m* Zustände. Dann wäre $D(m)+1 = M(m) \leq D(m)$. Widerspruch!

Gibt es einen sicheren Virus-Test?

Ein Axiom der Aussagenlogik besagt:

$$A \to (\neg A \to \neg B)$$

Dieses Gesetz ist (natürlich) tautologisch:

A	$\neg A$	B	$\neg A \to B$	$A \to (\neg A \to B)$
T	F	T	T	T
T	F	F	T	T
F	T	T	T	T
F	T	F	F	T

Wenn nun A <u>und</u> $\neg A$ beweisbar wären:

$$\begin{array}{l} A \\ A \to (\neg A \to B) \\ \text{-----------}(MP) \\ \quad \neg A \to B \\ \quad \neg A \\ \text{-----------}(MP) \\ \qquad B \end{array}$$

...dann wäre also jede beliebige Aussage B beweisbar!

Daher wäre es wünschenswert, eine Maschine *ITM* (Inkonsistenz-Test-Maschine) zu entwickeln, die Alarm schlägt bei widersprüchlichem Input! In dieser Allgemeinheit ist das ein unrealisierbarer Traum. Nicht einmal viel eingeschränktere Aufgaben lassen sich prinzipiell nicht algorithmisch lösen. Zum Beispiel:

„There are no safe virus tests"
(W. F. Dowling, Drexel University, Philadelphia, 1989)

<u>Beweis:</u>

Definition: Ein Computerprogramm heisst *Virus*, gewenn es, nachdem es gestartet wird, das Betriebssystem (BS) ändert.

Definition: Ein Computerprogramm *P*, gestartet mit Input *x* in der Umgebung des BS, *verbreitet ein Virus*, gewenn es das BS ändert; sonst heisst es *sicher*.

Abmachung: Es existiert mindestens ein Virus, nennen wir es *V*.

Angenommen, $\exists$ sichere Virus-Test-Machine *VTM*, die zu jedem Paar (Programm *P*, Input *x*) entscheiden kann, ob *P* sicher ist mit *x* oder nicht:

$$(P;x) \mapsto [[VTM]] \mapsto \begin{cases} "sicher", \text{ falls } P \text{ sicher mit } x \\ "nicht\ sicher", \text{ sonst} \end{cases}$$

$\Rightarrow$ $\exists$ auch Maschine *DEVIL*:

$$P \mapsto [[DEVIL]] \mapsto \begin{cases} "I\ love\ you", \text{ falls } VTM(P, GN(P)) = "nicht\ sicher" \\ \text{verändert BS mit Hilfe von } V, \text{ sonst} \end{cases}$$

Ist nun *DEVIL* sicher mit Input *GN(DEVIL)* oder nicht???

- Falls *DEVIL* sicher mit *GN(DEVIL)*
  $\Rightarrow$ *DEVIL* schreibt *I love you*
  $\Rightarrow$ *VTM(DEVIL,GN(DEVIL))="nicht sicher"*
  $\Rightarrow$ *DEVIL* ist nicht sicher mit Input *GN(DEVIL)*. WIDERSPRUCH!

- Falls *DEVIL* nicht sicher mit *GN(DEVIL)*
  $\Rightarrow$ Entweder verändert *DEVIL* das BS mit *V* (dann ist *VTM(DEVIL;GN(DEVIL))="sicher",* d.h. *DEVIL* ist sicher mit *GN(DEVIL)*) oder *DEVIL* schreibt „*I love you*“ (dann muss der Aufruf von *VTM* das Virus verursachen, d.h. *VTM* ist selber nicht sicher). In beiden Fällen resultiert ein WIDERSPRUCH!

□

Richardsons Taschenrechner-Sabotage

Sei E eine Menge von Termen über $\mathbb{R}$ mit höchstens einer Variablen x, z.B.

$$E = \left\{ \frac{3}{5},\ x,\ 0.2x^2,\ \sqrt{x},\ \frac{2}{x-1}, ... \right\}$$

Sei E^* die Menge der Funktionen, repräsentiert durch die Terme aus E. (Ist z.B. der Term $A = x^2$ in E, so ist die Funktion $A(x) = x^2$ in E^*.)

Annahmen über E^*:

- Die Funktion $id(x) = x$ ist enthalten.
- Die Funktionen $r(x) = \frac{m}{n}$ sind enthalten $\forall \frac{m}{n} \in \mathbb{Q}$.
- E^* ist abgeschlossen unter Addition, Subtraktion, Multiplikation und Komposition.

$A(x) \equiv B(x)$ soll bedeuten, dass beide Definitionsmengen übereinstimmen und zudem $\forall x \in D_A = D_B : A(x) = B(x)$.

Mögliche Prämissen für E^*:

(1) E^* enthält $\log(2)$, π, e^x und $\sin(x)$.

(2) E^* enthält die Betragsfunktion

(3) E^* enthält eine (totale) Funktion $\varphi(x)$, so dass für keine Funktion $f(x)$ in E^* gilt: $f'(x) = \varphi(x)$.

<u>Sätze von D. Richardson (1968):</u>

E^* erfüllt (1) $\Rightarrow$ Es kann algorithmisch NICHT entschieden werden, ob zu einer beliebigen Funktion $A(x)$ aus E^* ein $x \in \mathbb{R}$ existiert, so dass $A(x) < 0$ ist.

E^* erfüllt (1)&(2) $\Rightarrow$ Es kann algorithmisch NICHT entschieden werden, ob für eine beliebge Funktion $A(x) \in E^*$ gilt: $A(x) \equiv 0$

E^* erfüllt (1)&(2)&(3) $\Rightarrow$ Es kann algorithmisch NICHT entschieden werden, ob zu einer beliebigen Funktion $A(x)$ aus E^* eine Funktion $f(x)$ aus E^* existiert, so dass $f'(x) \equiv A(x)$.

Halteproblem und die Methode der Reduktion

Satz: Das Halteproblem ist algorithmisch unlösbar (Turing, 1936)

Gibt es eine TM, die bei Input (x, y) „1" berechnet, falls die TM mit Code *x* und Input *y* nach endlich vielen Schritten anhält und sonst „0"?

Für Mathematiker soll die Frage noch etwas präziser gestellt werden:

Sei H := {(x, y) / die TM mit Code *x* und Input *y* hält nach endlich vielen Schritten an} die zur Diskussion stehende Sprache. Nun stellt sich die Frage, ob H <u>entscheidbar</u> ist, d.h. ob eine TM existiert, die die charakteristische Funktion X von H berechnet, wobei X((x, y))=1, gewenn $(x, y) \in$ H und =0, sonst.

Die Antwort darauf lautet „Nein!" Insbesondere kann es kein Betriebssystem geben, das von jedem Paar (Programm , Input) entscheiden kann, ob das Programm mit diesem Input jemals anhält oder nicht - und zwar ohne das Programm mit dem Input zu starten!

Die folgende Beweisskizze zeigt zwar den Kernpunkt des Beweises, blendet aber absichtlich gewisse sehr technische Details aus:

Zum Beweis nehmen wir an, es gäbe doch eine TM, die dieses leistet. Dann muss es natürlich auch eine TM geben, die zu jeder Zahl $x \in \mathbb{N}$ entscheidet, ob die TM mit Code *x* und Input *x* jemals anhält oder nicht. (Das ist ja lediglich ein Spezialfall der eingangs gestellten Frage!). Nennen wir diese TM „HALT-TEST". D.h.:

$$\text{Input x} \mapsto \boxed{\text{HALT-TEST}} \mapsto \begin{cases} 1, \text{ falls TM mit Code x und Input x nach endlich vielen Schritten anhält} \\ 0, \text{ falls die TM mit Code x und Input x nie anhält} \end{cases}$$

Wenn diese Maschine tatsächlich existieren sollte, dann können wir sie leicht zu einer anderen Maschine umbauen, die ich „SPIELVERDERBER“ nennen möchte:

$$\text{Input x} \mapsto \boxed{\text{SPIELVERDERBER}} \mapsto \begin{cases} \text{geht in } \infty\text{-Schleife über, falls HALT-TEST mit Input x 1 liefert} \\ \text{hält an, falls HALT-TEST mit Input x 0 liefert} \end{cases}$$

Sei b der Code der TM SPIELVERDERBER. Wie reagiert SPIELVERDERBER auf den Input b ?

SPIELVERDERBER mit Input b geht in eine Unendlichschleife über $\Leftrightarrow$ HALT-TEST liefert bei Input b den Output „1“ $\Leftrightarrow$ Die TM mit Code b und Input b hält nach endlich vielen Schritten an. $\Leftrightarrow$ SPIELVERDERBER hält bei Input b nach endlich vielen Schritten an.

SPIELVERDERBER müsste also, falls HALT-TEST tatsächlich existieren würde, bei Input b gleichzeitig in eine Unendlichschleife übergehen und nach endlich vielen Schritten anhalten. Dieser Widerspruch macht klar, dass HALT-TEST nicht existieren kann. Das Halteproblem ist also algorithmisch unentscheidbar!

□

Abgesehen von der überaus interessanten Einsicht in die algorithmische Unlösbarkeit eines konkreten Problems bietet das Halteproblem noch einen weiteren Vorteil: Es kann als Basis für weitere Beweise dieser Art verwendet werden. In der Tat kann man von anderen Problemen nachweisen, dass sie algorithmisch unlösbar sind, indem man sie auf das Halteproblem reduziert. Dazu sei hier nur Folgendes gesagt:

Definition:

Ein Problem B heisst auf **das Halteproblem reduzierbar**, in Zeichen: $\mathrm{B} \leq \mathrm{HP}$, wenn eine Reduktionsfunktion *r* existiert, die B auf HP reduziert.

Eine solche Reduktionsfunktion

1. ist TM-berechenbar
2. ordnet jedem Paar (n, x) genau ein y zu.
3. hat die Eigenschaft, dass $(n, x) \in \mathrm{HP} \quad \Leftrightarrow \quad y = r(n, x) \in \mathrm{B}$

So kann man zum Beispiel die algorithmische Unlösbarkeit von Thue’s Wortproblem nachweisen oder auch die Unlösbarkeit von Hilbert Nr. 10 (Matijasevic, 1970), usw.

NP-vollständige Probleme

Eine *nicht-deterministische* TM ist ein 3-Tupel (A, Q, δ), wobei

- Q eine endliche Menge von Zuständen
- I das Inputalphabet
- A das Bandalphabet mit $I \subseteq A$
- $B \in A \setminus I$ das Leerzeichen (Blank)
- q_0 der Startzustand
- q_a der akzeptierende Zustand
- $\boxed{\delta : Q \times A \to P\left(Q \times \left(A \times \{L, R, S\}\right)\right)}$ die Programm-Funktion ist, die dem aktuellen Zustand und dem aktuell gesehenen Zeichen den neuen Zustand und das neue Zeichen samt der Bewegungen zuordnet. (L=Links, R=Rechts, S=Stop)

Ein Funktionswert von δ ist aber ein Element der Potenzmenge von $Q \times \left(A \times \{L, R, S\}\right)$, d.h. es stehen der Maschine zu jedem Zeitpunkt mehrere mögliche Fortsetzungen offen! *Ein* möglicher Berechnungspfad der nicht-det. TM ergibt sich aus der Auswahl von jeweils einer der Aktionen aus $\delta\left(q, \left(a_1, \ldots, a_k\right)\right)$.

P:= Menge aller Sprachen, die von einer deterministischen TM in polynomialer Zeit akzeptiert werden.

NP := Menge aller Sprachen, die von einer nicht-deterministischen TM in polynomialer Zeit akzeptiert werden.

(NP = **N**on deterministic **p**olynomial)

Das *P-NP-Problem*:

$$\boxed{\text{Ist } P \subset NP \text{ oder } P = NP \text{ ?}}$$

(bis heute ungelöst!)

Ein Problem L heisst *NP-vollständig*, gewenn

- $L \in NP$ und
- für jedes $L_1 \in NP$ $L_1 \leq_p L$ gilt.

(Das heisst, dass es eine polynomial-zeitbeschränkte deterministische TM gibt, die eine Funktion $f(x)$ berechnet, so dass $x \in L_1 \Leftrightarrow f(x) \in L$.)

Folgerung:
Kann von einem einzigen *NP-vollständigen* Problem gezeigt werden, dass es zu *P* gehört, dann ist automatisch *P*=*NP* !!!

NP-vollständige Probleme:

1) SAT – das erste als *NP-vollständig* erkannte Problem!
   (S. A. Cook, 1971)

2) *k*-Färbbarkeit von Graphen

3) Partition eines Graphen in Subgraphen mit Hamilton-Wegen

4) CLIQUE (Enthält ein Graph eine Clique der Grösse *k*?)

5) HAMILTONIAN CIRCUIT und H. PATH

6) SUBGRAPH ISOMORPHISM

7) DEGREE CONSTRAINT SPANNING TREE
   (Gibt es einen aufspannenden Baum, dessen Knoten alle Grad $\leq k$ haben?)

8) SHORTEST TOTAL PATH LENGTH SPANNING TREE

9) MAX CUT
   (Gibt es eine Graphenpartition in 2 Subgraphen, so dass die Summe der Gewichte aller Kanten, die einen Subgraphen mit dem anderen verbinden, $\geq k$ ist?)

10) TSP

11) LONGEST CIRCUIT
    (Gibt es einen circuit der Länge $\geq k$?)

12) MINIMUM EDGE-COST FLOW

13) RUCKSACK
    (Menge von Waren, Gewichte, Wertzuordnung, geg. Höchstgewicht und Mindestwert. Gibt es Submenge von Waren, die Höchstgewicht und Mindestwert einhält?)

…und zahlreiche weitere Probleme aus den Gebieten:

- Mengen und Partitionen
- Datenspeicherung und Kompression
- Scheduling
- Algebra und Zahlentheorie
- Lösbarkeit von Gleichungen
- Spiele und Puzzles
- Logik
- Automatentheorie
- Programmoptimierung

Was vom Traum übrigblieb:

Der Traum von Leibniz kann in einzelnen engen mathematischen Theorien tatsächlich realisiert werden! Zwei dieser Theorien möchte ich im Folgenden kurz ansprechen. Im Anschluss daran werden wir uns noch mit einer interessanten Frage auseinandersetzen, die sich im Zusammenhang entscheidbarer Theorien eigentlich aufdrängt: Wie schnell ist eine entscheidbare Theorie entscheidbar? Genauer: Wie viele Schritte benötigt ein algorithmischer Test, ob eine bestimmte Formel wahr oder falsch ist, mindestens? Fragen dieser Art sind sehr neu, erst in den 60er-Jahren des letzten Jahrhunderts wurden die ersten mathematischen Sätze aus der sog. Komplexitätstheorie bewiesen, der Theorie eben, die nach der Komplexität, dem minimalen Aufwand von Algorithmen fragt.

Betrachten wir nun eine erste entscheidbare Theorie etwas genauer:

Es handelt sich um die **Theorie der dichten Ordnung (ThdO).** Diese mathematische Teildisziplin vereinigt in sich alle Sätze zum Thema „Ordnungsrelation über rationalen Zahlen", also alle Sätze, die eine Aussage über rationale Zahlen machen unter Verwendung der ordnenden Zeichen $=$, $\neq$, $<$, $\leq$, $>$, $\geq$.

Die THdO benutzt also die Zeichen $=$, $\neq$, $<$, $\leq$, $>$,$\geq$

ferner die logischen Zeichen $\forall$, $\exists$, $\neg$, $\wedge$, $\vee$, $\rightarrow$ samt den Hilfszeichen) (,

die rationalen Zahlen als Konstanten

die Buchstaben x, y, z, ... (auch indiziert) als Variabeln

die Axiome der Prädikatenlogik 1. Stufe

und die folgenden theoriespezifischen Axiome:

(1) $\forall x \ (\neg (x<x))$

(2) $\forall x,y \ ((x<y) \ v \ (x=y) \ v \ (y<x))$

(3) $\forall x,y,z \ ((x<y) \wedge (y<z) \rightarrow (x<z))$

(4) $\forall x,y \ ((x<y) \rightarrow \exists z((x<z) \wedge (z<y))$

(5) $\forall x \ \exists y,z \ ((y<x) \wedge (x<z))$

(Das Axiom (4) bedeutet die Dichtheit; es gibt keine Löcher in dem durch das Zeichen „<" definierten Ordnungsgefüge!)

Eine typische Aussage dieser Theorie könnte z.B. so aussehen:

$$\exists y \ \exists x \ (\ ((x\leq y) \wedge (0\leq x) \wedge (x\neq y) \wedge (x\neq 5) \ ) \ v \ ((y\leq x) \wedge (x\leq 0) \wedge (x\neq y) \wedge (y\neq -3) \ ) \ )$$

Die ThdO ist entscheidbar, d.h. es gibt tatsächlich einen Algorithmus, der automatisch von jeder beliebigen in ThdO formulierbaren Aussage entscheidet, ob sie wahr oder falsch ist. Eine genaue und detaillierte Beschreibung dieses Algorithmus würde den Rahmen dieses Votrages sprengen. Ich beschränke mich daher auf ein paar erläuternde Bemerkungen:

Sei also eine beliebige Formel F aus der ThdO gegeben.

1.) Zuerst reduzieren wir die 6 Zeichen =, ≠, < ,≤ ,> ,≥ auf zwei, indem wir alle Zeichen allein durch die beiden Zeichen ≤, ≠ ausdrücken:

a=b ersetzen wir durch ....
a>b ersetzen wir durch ....
a<b ersetzen wir durch
a≥b ersetzen wir durch ....

2.) Dann wird F in die sog. <u>pränexe Normalform</u> übergeführt, d.h. in eine Form, in der alle Quantoren (∀ , ∃) am Anfang der Formel stehen. Diese Umformung ist ausserordentlich trickreich, aber stets möglich. F nimmt dann also die Gestalt

∀ ∀ ∀ ∀ ∀ ∀ ∀ ∃ . ∃ ..(-------- quantorenfreier Teil ----------) an.

3.) Nun schafft man alle Allquantoren weg, indem man überall ∀ x ersetzt durch ¬ ∃ x ¬ .
4.) Dann wird der quantorenfreie Teil von F in die sog. <u>disjunktive Normalform (DNF</u>) übergeführt, d.h. in die Form

((........) ∧ (........) ∧ (........) .∧.) v ((........) ∧ (........) ∧ (........) .∧.) v (.........

Nun ist der quantorenfreie Teil eine oder-Verbindung von Klammern, die jeweils und-Verbindungen sind.

Unser obiges Beispiel ist bereits in dieser Form!

5.) Schliesslich wird von innen her jeder ∃ -Quantor eliminiert, indem man im quantorenfreien Teil gewisse Ersetzungen und Streichungen vornimmt, die auch wieder besonders trickreich, aber möglich sind. Dabei werden gleichzeitig alle Teile der Formel durch „T" (für „true") oder „F" (für „false") ersetzt.
6.) Am Schluss kann man aus den und- und oder-Verbindungen der Zeichen „T" und „F" den Wahrheitswert der ganzen Formel F ablesen.

Ich hoffe, Ihnen mit diesen vagen Erläuterungen eine wenigstens ungefähr Vorstellung gegeben zu haben von der Möglichkeit, eine beliebige Formel aus ThdO automatisch zu entscheiden. Bei Problemen, die sich in der ThdO formulieren lassen, können wir also in der Tat, wie Leibniz das vorschlug, zurücklehnen und durch blosses Rechnen entscheiden, wer recht hat!

Eine weitere entscheidbare Theorie ist die sog. **Presburger-Arithmetik (PA)**. Hierin sind alle mathematischen Sätze zusammengefasst, die einzig von den natürlichen Zahlen, der Operation"+" und der Ordnungsrelation „<" handeln. Den Beweis der Entscheidbarkeit der PA finden Sie in Presburgers Originalarbeit:

M. Presburger: „Über die Vollständigkeit eines gewissen Systems der Arithmetik der ganzen Zahlen, in welchem die Addition als einzige Operation hervortritt"
Compte-Rendus du 1. Congrès des Mathématiciens des pays Slav., Warsaw, 1929, pp. 92-101.

Nun hatte ich weiter oben angedeutet, dass sich in der 60er-Jahren des letztem Jahrhunderts die Bemühungen der Mathematiker der Komplexität, also dem zeitlichen (und räumlichen!) Aufwand von Algorithmen, zuwandten. Was war geschehen?

Das Jahr 1968 war nicht nur auf der politischen Bühne ein Jahr der Revolution. Auch unter den Mathematikern machte sich eine Unzufriedenheit breit über das damals Erreichte. Man war unzufrieden über die klassischen wissenschaftlichen Paradigmen, das Paradigma etwa, ein Problem für definitiv gelöst zu halten, wenn schon eine Lösung bzw. ein Lösungsalgorithmus existiert. Fragen

nach der Optimalität von Lösungen im Hinblick auf die Anzahl benötigter Operationen machten sich jetzt breit. Wie viele Operationen sind zur Lösung mindestens nötig? Wie viel Speicherplatz (auf einer Turing-Maschine) wird zur Lösung mindestens benötigt? Durch solche Fragestellungen entstand die sog. Komplexitätstheorie, die Theorie, die lange bestehende Algorithmen neu in Frage stellt, ihren Aufwand untersucht und neue, bessere, schnellere Algorithmen für längst bekannte Probleme sucht. Einer der Pioniere dieser Theorie war (und ist) Volker Strassen, gewissermassen der „Rudi Dutschke der Algorithmik". Seine Arbeit aus dem Jahre 1969 „Gaussian elimination is not optimal" gab den Anstoss zu vielen weiteren Sätzen dieser Art. (Es ging bei dieser Arbeit darum zu zeigen, dass die „Gausselimination", ein Algorithmus zur Bearbeitung von Matrizen, nicht optimal im Hinblick auf die Anzahl Operationen ist. Strassen konnte dasselbe Problem mit weniger Operationen(reelle Multiplikationen!) lösen und setzte damit die Frage in Umlauf, was denn die Zeitkomplexität, als die minimale Anzahl Operationen, der Gausselimination sei.) Strassen war sich damals der Bedeutung seiner Forschungen so wenig bewusst, dass er vom seinem Fachkollegen Cook regelrecht zur Publikation gedrängt werden musste.

Die Einfachheit und Natürlichkeit vieler älterer Algorithmen haben deren Optimalität lange als selbstverständlich erscheinen lassen. Daher kam es in den 60er-Jahren einer (mathematische) Revolution gleich, die Frage nach der Optimalität überhaupt zu stellen.

Auch bei unseren beiden oben erwähnten entscheidbaren Theorien stellte sich bald die Frage nach ihrer Komplexität. Die in diesem Zusammenhang wichtigsten Resultate sind:

ThdO: Anzahl Operationen= $n^{O(n)}$

(Das im Exponenten verwendete sog Landau-Symbol „O" bedeutet, dass der Exponent $\leq C \cdot n$ ist für eine gewisse Konstante *C*.) Dabei ist *n* die Länge der Inputformel. Formeln der Länge $n = 10$ ziehen also 10^{10} Berechnungsschritte nach sich.

Anzahl Speicherzellen $= n \cdot lb(n)$

PA:

Derek C. Oppen von der University of Toronto analysierte 1973 einen auf D. C. Cooper (1972) zurückgehenden Algorithmus zur Entscheidung der PA. Die Zeit- und Raumschranken, die er dabei fand, sind allerdings obere Schranken, d.h. sie geben die maximale Anzahl Operationen und die maximale Anzahl Speicherzellen wieder. Im Jahre 1974 lieferten Michael J. Fischer und Michael O. Rabin vom MIT untere Schranken, mithin die eigentliche Komplexität, die minmale Anzahl Operationen bzw. Speicherzellen. Nach diesem Werk ist die PA „super-exponentiell":

Es gibt Formeln der Länge *n*, deren Entscheidung mind. $2^{2^{cn}}$ Schritte und auch Speicherzellen benötigt.

Es ist klar, dass die Frage nach der Komplexität eines Algorithmus von enormer Bedeutung sein kann. Ein Algorithmus nützt uns nichts, wenn sein Aufwand so enorm ist, dass eine automatische Lösung Jahre dauert. Denken wir nur an das Problem der „Dreifärbbarkeit von Graphen". Es gibt 3 mögliche Farben pro Knoten. Da es *n* Knoten sind, sind im Wesentlichen 3^n verschiedene Färbungen zu prüfen. Wenn unser Algorithmus nun einfach alle möglichen Färbungen prüft, dann haben wir ein Komplexitätsproblem! Angenommen nämlich, unser Computer könnte 1 Färbung pro Nanosekunde prüfen; dann würde er 3^n Nanosekunden arbeiten. Bei einem Graphen mit $n = 100$ Knoten wären das bereits 3^{100} Nanosekunden, was ungefähr $1.6 \cdot 10^{31}$ Jahre sind.