Einführung:

Neben den Problemen und Fragen der praktischen Informatik, wo es in erster Linie um die Wahl der geeigneten Algorithmen und Datenstrukturen und die geeigneten Programmiersprachen geht, stellt sich der **nachdenkliche Informatiker** auch grundsätzliche Fragen wie zum Beispiel:

- Die vielen Computer, die bisher entwickelt wurden und in Zukunft entwickelt werden, unterscheiden sich sehr stark. Gibt es trotzdem etwas Gemeinsames, was allen Computern zu Grunde liegt?
- Gibt es trotz der vielen technischen Möglichkeiten einen zumindest theoretisch denkbaren *minimalen Computer*, der ohne Rücksicht auf Zeitbedarf und Speicherbedarf **alle** Algorithmen im Prinzip auszuführen imstande ist?
- Gibt es für jedes Problem, das sich mit Mitteln von Informatik und Mathematik beschreiben lässt, einen Lösungsalgorithmus? Oder gibt es Probleme, für die sich nachweislich kein Algorithmus angeben lässt, d.h. die niemals mit Hilfe eines Computers gelöst werden können, egal wie schnell und leistungsfähig der Computer auch sei?
- Wie kann man Algorithmen hinsichtlich ihrer Qualität, insbesondere ihrer Laufzeit, miteinander vergleichen, ohne sich auf einen speziellen Computertyp zu beziehen?
- Kann man die Richtigkeit von Programmen durch andere Programme feststellen lassen oder sogar *beweisen*?

Fragen dieser Art sind Gegenstand der *Theoretischen Informatik*. Dieses Gebiet der Informatik steht der Mathematik sehr nahe und hat auch diverse Bereiche der Mathematik befruchtet.

Geschichte

Die Idee, einen Algorithmus oder ein Rezept zur Durchführung irgendeiner Aufgabe zu haben, exisiert schon seit tausenden von Jahren. Über viele Jahre hinweg glaubte man folgendes: Wenn irgendein Problem präzise dargelegt werden könnte, dann gibt es — ausreichendes Bemühen vorausgesetzt — am Ende stets eine Lösung oder es könnte schließlich bewiesen werden, dass keine Lösung existiert. Man glaubte also, es gäbe kein Problem, das wirklich so schwierig ist, dass es prinzipiell nicht gelöst werden könnte.
Einer der berühmtesten Vertreter dieses Glaubens war der Mathematiker DAVID HILBERT (1862-1943). Er sagte einst:

> „Ein bestimmtes mathematisches Problem muss notwendigerweise einer exakten Lösung zugänglich sein, entweder in Form einer direkten Antwort auf eine gestellte Frage, oder durch den Beweis seiner Unlösbarkeit und dem damit verbundenen notwendigen Scheitern eines jeden Versuchs...
>
> Das, was uns am meisten reizt, wenn wir uns der Lösung mathematischer Probleme zuwenden, ist genau das, was wir innerlich hören: hier ist das Problem, suche die Lösung; du kannst sie durch reines Nachdenken finden; denn in der Mathematik gibt es nichts, was nicht erkannt werden kann.“

Hilberts Ziel war es, ein mathematisches System zu ersinnen, in dem alle Probleme präzise als Aussagen formulierbar sind, die entweder wahr oder falsch sind. Seine Vorstellung war es, einen Algorithmus zu finden, der zu einer gegebenen Aussage in einem formalen System entscheiden könnte, ob die Aussage richtig oder falsch ist. Hätte Hilbert dieses Ziel erreicht, dann hätte jedes wohldefinierte Problem einfach durch eine Algorithmusausführung gelöst werden können. Die Wahrheit einer Aussage in einem formalen System zu entscheiden, dieses Problem ist unter dem Namen Hilbertsches *Entscheidungsproblem* bekannt geworden.
Die 30er Jahre brachten eine Reihe von Untersuchungen, die, leider entgegen Hilberts Vorstellungen, zeigten, dass das Entscheidungsproblem nicht berechenbar ist. D.h., es gibt keinen Algorithmus, der dies könnte. In gewisser Weise haben dadurch die Mathematiker Glück gehabt, denn sie wären alle durch einen solchen Algorithmus arbeitslos geworden.
Die ersten Nachrichten dieser Entdeckung kamen 1931 auf, als KURT GÖDEL sein berühmtes *Unvollständigkeitstheorem* veröffentlichte. U.a. besagt es, dass es keinen Algorithmus gibt, der als Eingabe irgendeine Aussage über die natürlichen Zahlen erhält, und dessen Ausgabe feststellt, ob diese Aussage wahr oder falsch ist. Ähnliche Ergebnisse fanden ALONZO CHURCH, STEPHEN KLEENE, EMIL POST, ALAN TURING u.a.
Bemerkenswert ist noch, dass Ergebnisse über Probleme, die nicht mit dem Computer lösbar sind, gerade in den 30er Jahren aufkamen, als es noch gar keine Computer gab.

Die Church-Turing-These

Um festzustellen, was ein Algorithmus kann oder nicht kann, muss erst einmal der Begriff Algorithmus geklärt werden. Dazu gab es verschiedene Ansätze, die hier nur andeutungsweise angegeben werden:

- KURT GÖDEL: Ein Algorithmus ist eine Folge von Regeln zur Bildung mathematischer Funktionen aus einfacheren mathematischen Funktionen.
- ALONZO CHURCH: Er verwendete einen ähnlichen Formalismus, den er λ-Kalkül nannte.
- EMIL POST: Er ersann einen Mechanismus, der Symbole manipuliert und den er *Produktionssysteme* nannte.
- STEPHEN KLEENE: Er definierte eine Klasse mathematischer Objekte, die er *rekursive Funktionen* nannte.
- ALAN TURING: Ein Algorithmus ist das, was auf der nach ihm benannten *Turing-Maschine* ausführbar ist (siehe unten).

Überraschenderweise stellte sich im Laufe der Zeit heraus, dass alle auf den ersten Blick so verschiedenen Ansätze gleichwertig sind. D.h., wenn Etwas durch einen Algorithmus, der auf die eine Art beschrieben wurde, berechnet werden kann, dann kann er auch in einer der anderen Versionen berechnet werden. Als Folge dieser Gleichwertigkeit wurde die folgende Aussage eine weitverbreitete Annahme:

> Alle vernünftigen Definitionen von „Algorithmus“, soweit sie bekannt sind oder die jemals irgendwer aufgestellt hat, sind gleichwertig und gleichbedeutend.

Diese Annahme ist als *Church-Turing-These* bekannt geworden. Diese These ist kein mathematischer Satz und sie kann auch nicht bewiesen werden. Bis heute ist aber kein einleuchtendes Gegenbeispiel erbracht worden. Die Church-Turing-These behauptet lediglich, dass wir glauben, eine vernünftige Definiton für das gefunden zu haben, was wir Algorithmus nennen. Mittlerweile ist man so von ihrer Gültigkeit überzeugt, dass mathematische Beweise, die sich auf sie stützen, akzeptiert werden.
Zweite Fassung der Church-Turing-These:

> Turingmaschinen sind formale Modelle von Algorithmen, und kein Berechnungsverfahren kann „algorithmisch“ genannt werden, das nicht von einer Turingmaschine ausführbar ist.

Dritte Fassung der Church-Turing-These:

> Jedes algorithmische Problem, das in *irgendeiner* Programmiersprache programmiert und auf *irgendeinem* dafür geeigneten Computer ausgeführt werden kann (sogar auf Computern, die noch nicht gebaut sind, aber prinzipiell gebaut werden könnten), und selbst wenn es unbeschränkt viel Zeit und Speicherplatz für immer größere Eingaben benötigt — jedes solche Problem ist auch durch eine Turing-Maschine lösbar!
>
> Und alles, was mit einer Turing-Maschine nicht berechenbar ist, lässt sich überhaupt nicht berechnen.

Minimale Rechnermodelle

Was ist — zumindest prinzipiell — das Gemeinsame aller Computer und Algorithmen? Dazu sind im Laufe der Geschichte der Informatik mehrere Modelle entwickelt worden. Die Wichtigsten, die Zählmaschine und die Turing-Maschine seien hier vorgestellt:

Die Zählmaschine

Die *Zählmaschine* mit wahlfreiem Zugriff (*random access machine*) ist ein abstraktes Rechnermodell, das unseren heutigen Computern am ehesten ähnelt. Die Bestandteile sind

- ein Datenspeicher mit den Speicherzellen $S_1, S_2, S_3 \ldots S_n$, wobei $n \in \mathbb{N}$ beliebig groß und jede Speicherzelle eine beliebig große natürliche Zahl aufnehmen kann;
- ein Programmspeicher beliebiger Größe.

Die Einschränkung auf natürliche Zahlen erscheint willkürlich. Es lässt sich aber zeigen, dass man alle anderen Datentypen wie negative Zahlen, rationale Zahlen, Zeichen usw. mit bestimmten Algorithmen auf natürliche Zahlen abbilden kann.
Wenn es einen Programmspeicher gibt, dann muss auch ein Programm ablaufen können. Welche Programmstrukturen sind erlaubt? Es reicht eine minimale Programmiersprache, die lediglich eine While-Schleife bereitstellt und das Erhöhen oder Erniedrigen einer Speicherzelle. Man kann zeigen, dass dies ausreicht, um **alle** Algorithmen ausführen zu können. Genauer genommen müssen die folgenden Anweisungen möglich sein:

`x = 0`	Löschen des Inhalts einer beliebigen Speicherzelle.
`x = x + 1`	Erhöhen einer beliebigen Speicherzelle um 1.
`x = x - 1`	Erniedrigen einer beliebigen Speicherzelle um 1. Wegen der Einschränkung gilt hier, dass für $x = 0$ auch $x - 1 = 0$ gilt.
`while x <> y do`	While-Schleife.

...

`end`

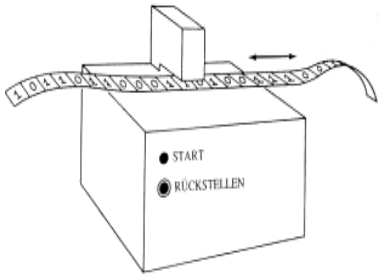
Das erscheint als sehr wenig. Man schämt sich ja fast als Informatiker, dass alles, was man machen kann, mit so wenigen Vorgaben möglich sein soll. Man wird allerdings etwas beruhigt, wenn man weiß, dass man für die Mathematik eigentlich auch nur die natürlichen Zahlen definieren muss und alles Andere sich daraus logisch ergibt. Insofern zeigt sich hier die tiefgehende Beziehung zwischen Mathematik und theoretischer Informatik.
Wie sind aber die „höheren“ Operationen, die man von einer Programmiersprache gewohnt ist, möglich? Dies soll hier nur angedeutet werden oder als Übung durchgeführt werden:

1. Laden einer Speicherzelle mit einer Konstanten $n \in \mathbb{N}$: Übung
2. Zuweisung einer Variablen y an eine Variable x: Übung
3. Erhöhen einer Variablen y um eine Variable x: Übung
4. Addition zweier Variablen x und y: wie in Punkt 2 und 3
5. Subtraktion: Übung
6. Wenn Addition und Subtraktion machbar sind, dann sind es auch Multiplikationen und Divisionen, Restbildung usw.
7. Booelesche Ausdrücke: Setze 0 für `false` und 1 für `true`.
8. Die Anweisung `if expr then s` lässt sich durch eine while-Schleife ersetzen: Übung
9. Die Anweisung `if expr then s1 else s2` lässt sich durch eine while-Schleife ersetzen: Übung
10. Aufrufe von (nichtrekursiven) Prozeduren lassen sich durch Einsetzen des Prozedurrumpfes in das „Hauptprogramm“ beseitigen. Rekursive Prozeduren können immer in nichtrekursive transformiert werden.

Bemerkungen:

- Die For-Schleife reicht nicht aus, da bei ihr die Obergrenze der Laufvariablen von vornherein fest liegt. In Java ist dies allerdings etwas flexibler geregelt.
- Man kann statt der While-Schleife auch eine Struktur wie `if x=0 goto A` benutzen, wobei `A` für irgend eine Anweisung an einer bestimmten Stelle im Programm steht. Mit dieser Goto-Möglichkeit kann man jede While-Schleife nachbilden und umgekehrt, also sind beide Darstellungen gleichwertig.

Die Turing-Maschine



Sie besteht aus

1. aus einem Schaltwerk mit einer festen Anzahl von Zuständen z_1 bis z_n,
2. einem beidseitig unendlich langen Band (z.B. aus Papier zu denken) als Speicher
3. und einem Schreib-Lese-Kopf.

Das Band ist in Zellen eingeteilt, wobei jede Zelle ein Zeichen eines gegebenen Alphabets aufnehmen kann. Das Band kann zellenweise nach rechts oder links bewegt werden. Man stellt sich allerdings üblicherweise vor, dass der Schreib-Lese-Kopf nach links oder nach rechts läuft.
Zu Beginn enthält das Band die Eingabedaten. Der Kopf steht auf einer Zelle des Bandes. Was passiert nach dem Start der Maschine?

1. Der Schreib-Lese-Kopf liest das Zeichen auf dem Band und gibt es an das Schaltwerk weiter. Aus dem gegenwärtigen Zustand und dem gelesenen Zeichen bestimmt das Schaltwerk aus einer Tabelle
   - ein neues Zeichen,
   - einen neuen Zustand und
   - eine Bewegungsrichtung für den Schreib-Lese-Kopf, die nur links oder rechts lauten kann.
2. Das neue Zeichen wird an der aktuellen Stelle auf das Band geschrieben, der Schreib-Lese-Kopf geht um eine Zelle nach rechts oder links und das Schaltwerk geht in den neuen Zustand über.
3. Einer der Zustände im Schaltwerk muss als Startzustand gesetzt werden.
4. Einer der Zustände im Schaltwerk muss der Stoppzustand sein. Wird dieser Zustand erreicht, stoppt die Maschine.

Diese Maschine ist wirklich sehr primitiv aufgebaut. Man braucht dazu keine komplizierte Mechanik oder Elektronik. Denkbar wäre z.B. eine Turing-Maschine, die aus einer Rolle Toilettenpapier (wegen der hilfreichen Zelleneinteilung) mit weiteren unendlich vielen angeklebten Rollen rechts und links davon, einer Markierung (z.B. ein großer Stein) für die Position des Schreib-Lese-Kopfs und einem Zettel, auf dem die verschiedenen Zustände und Aktionen je nach gelesenem Zeichen, besteht. Als einfaches Alphabet wären die Zeichen 0 und 1 denkbar, wobei man eine 1 durch

einen kleinen Stein auf dem Toilettenpapier darstellen könnte und die 0 durch ein steinfreies Blatt repräsentiert wird. Ein Mensch könnte dann den Ablauf einer solchen „Maschine“ in Ruhe in einer geröllreichen Umgebung durch Steinelegen und Nachgucken auf dem Zettel für die Zustände verfolgen.
Denken wir nun zurück an die Church-Turing-These: Dies heißt doch nun, dass man prinzipiell das Gleiche leisten kann wie der modernste Computer oder wie der jeweils in der Zukunft zu irgendeiner Zeit beste Computer, wenn man sich nur mit einem Blatt Papier zum Aufschreiben der Zustände und Übergänge, genügend bzw. unendlich viel Toilettenpapier und der gleichen Menge an Steinchen bewaffnet. Genügend Zeit zur Ausführung der Algorithmen sollte man sich allerdings auch nehmen.

Gleichwertigkeit

Es lässt sich zeigen, dass die beiden vorgestellten minimalen Rechnermodelle, die Zählmaschine und die Turing-Maschine, gleichwertig sind. D.h., dass eine Turing-Maschine eine Zählmaschine simulieren kann und umgekehrt.
In der theoretischen Informatik wird die Turing-Maschine häufiger benutzt, da sie vom formalen Standpunkt aus einfacher zu handhaben ist. Deswegen beschäftigen wir uns im nächsten Abschnitt etwas ausführlicher mit diesem merkwürdigen „Gerät“.