Theoretische Informatik I

Duale Hochschule Baden-Württemberg – Lörrach
Studiengang Informatik – TIF21

Januar 2022–März 2022

Teil III

Algorithmen

Übersicht

Definition
Intuitive Algorithmen
Registermaschinen
Formale Algorithmen
Imperative Programme
Komplexität
Berechenbarkeit

1 Definition

2 Komplexität

3 Berechenbarkeit

Übersicht

Definition
Intuitive Algorithmen
Registermaschinen
Formale Algorithmen
Imperative Programme
Komplexität
Berechenbarkeit

1 Definition
- Intuitive Algorithmen
- Registermaschinen
- Formale Algorithmen
- Imperative Programme

Intuition

Intuition

Ein **Algorithmus** ist eine präzise, endliche Verarbeitungsvorschrift, wie die Instanzen einer Klasse von Problemen gelöst werden.
Genauer: Wie wird eine Eingabe auf eine Ausgabe abgebildet.

Wir bemerken, dass Algorithmen insbesondere Abbildungen modellieren. Da Algorithmen nicht terminieren müssen, modellieren sie partielle Abbildungen.

Beispiel

Wir möchten die natürlichen Zahlen 1 bis 5 addieren. Ein Algorithmus hierzu lautet:

»Nimm die Zahl 1, addiere dazu die Zahl 2, addiere dazu die Zahl 3, addiere dazu die Zahl 4, addiere dazu die Zahl 5«.

In Pseudocode:

Beispiel

```
Summe := 1 + 2 + 3 + 4 + 5
```

Beispiel

Allgemeiner könnten wir sagen:

»Wähle ein n, wähle als Summe die Zahl 1, gehe in Gedanken die Zahlen 2 bis n durch und addiere zur Summe immer die aktuelle Zahl«.

In Pseudocode:

Beispiel

```
Lies n ein;
Setze Summe := 1;
Wiederhole für i := 2, ..., n
  Setze Summe := Summe + i;
Gib Summe aus
```

Übersicht

Definition
Intuitive Algorithmen
Registermaschinen
Formale Algorithmen
Imperative Programme
Komplexität
Berechenbarkeit

1 Definition
- Intuitive Algorithmen
- Registermaschinen
- Formale Algorithmen
- Imperative Programme

Registermaschinen

Definition
Intuitive Algorithmen
Registermaschinen
Formale Algorithmen
Imperative Programme
Komplexität
Berechenbarkeit

Um den formalen Algorithmenbegriff präzisieren zu können, führen wir nun die Registermaschinen ein.

Eine Registermaschine besteht aus einer Reihe von Registern, die mit Zahlen aus $\mathbb{N}_0$ durchnummeriert sind. In jedem Register steht eine Zahl aus $\mathbb{N}_0$. Auf diesen Registern können zwei Operationen ausgeführt werden.

Registermaschinen
Syntax und Semantik

Definition

Die **Menge der Registermaschinenprogramme** RMP ist induktiv definiert:

- $a_i \in RMP$ für $i \in \mathbb{N}_0$.
  Das Programm erhöht den Inhalt des Registers i um 1.
- $s_i \in RMP$ für $i \in \mathbb{N}_0$.
  Das Programm erniedrigt den Inhalt des Registers i um 1, falls der Inhalt vorher größer 0 war (andernfalls bleibt er 0).
- Mit $\rho, \varrho \in RMP$ sind auch folgende Ausdrücke Elemente von RMP:
  - $\rho\varrho$ (**Konkatenation**)
    Das Programm wird ausgeführt, indem zuerst ρ und danach ϱ ausgeführt wird.
  - $(\rho)_i$ für $i \in \mathbb{N}_0$ (**Iteration**)
    Das Programm wird ausgeführt, indem zuerst geprüft wird, ob das Register i den Wert 0 hat. Falls nicht, so wird ρ solange ausgeführt, bis das Register i den Wert 0 hat.

Registermaschinen
Eingabe und Ausgabe

Die Eingabe eines Registermaschinenprogramms besteht aus einer Belegung der Register. Wir wollen vereinbaren, dass nur endlich viele Register Werte ungleich 0 haben.

Nach Setzen der Eingabe lassen wir das Programm laufen. Falls es zuende gelaufen ist, erhalten wir eine Ausgabe.

Die Ausgabe eines Registermaschinenprogramms besteht aus einer Belegung der Register. Da zur Eingabezeit nur endlich viele Register Werte ungleich 0 hatten und das Programm nur endlich viele Register veränderte (ein Programm hat nur Zugriff auf endlich viele Register), haben auch bei der Ausgabe nur endlich viele Register Werte ungleich 0.

Registermaschinen

Beispiel

Wir wollen die Zahlen in Register 0 und Register 1 addieren und das Ergebnis in Register 2 speichern. Die Ausgabewerte in den Registern 0 und 1 sollen mit ihren Eingabewerten übereinstimmen. Ein Programm dafür lautet:

$$(s_0 a_2 a_3)_0 (s_3 a_0)_3 (s_1 a_2 a_3)_1 (s_3 a_1)_3$$

Wir verwenden hierbei Register 3 als Hilfsregister.
Wir gehen davon aus, dass die Anfangsbelegung von Register 2 und Register 3 jeweils 0 ist.

Registermaschinen

Beispiel (fortgesetzt)

Wir wollen nun das Programm $(s_0 a_2 a_3)_0 (s_3 a_0)_3 (s_1 a_2 a_3)_1 (s_3 a_1)_3$ auf die Eingabe 3 und 2 anwenden.

Register	Start	$(s_0 a_2 a_3)_0$			$(s_3 a_0)_3$		
		1. Lauf	2. Lauf	3. Lauf	1. Lauf	2. Lauf	3. Lauf
3	0	**1**	**2**	**3**	**2**	**1**	**0**
2	0	**1**	**2**	**3**	3	3	3
1	2	2	2	2	2	2	2
0	3	**2**	**1**	**0**	**1**	**2**	**3**

Register		$(s_1 a_2 a_3)_1$		$(s_3 a_1)_3$		Ende
		1. Lauf	2. Lauf	1. Lauf	2. Lauf	
3	0	**1**	**2**	**1**	**0**	0
2	3	**4**	**5**	5	5	5
1	2	**1**	**0**	**1**	**2**	2
0	3	3	3	3	3	3

Wir erhalten als Ergebnis also 5.

Übersicht

Definition
Intuitive Algorithmen
Registermaschinen
Formale Algorithmen
Imperative Programme
Komplexität
Berechenbarkeit

1 Definition
- Intuitive Algorithmen
- Registermaschinen
- Formale Algorithmen
- Imperative Programme

Formale Algorithmen

Definition
Intuitive Algorithmen
Registermaschinen
Formale Algorithmen
Imperative Programme
Komplexität
Berechenbarkeit

Definition

- Eine **Klasse von Problemen** besteht aus einer Menge von Eingaben E, einer Menge von Ausgaben A und einem Zusammenhang von Eingaben und Ausgaben.
- Ein (formaler) **Algorithmus** zur Lösung einer Klasse von Problemen ist ein Registermaschinenprogramm, das jede Eingabe $e \in E$ nach endlicher Zeit in die entsprechende Ausgabe $a \in A$ überführt.

Beachte, dass Eingaben und Ausgaben geeignet als Zahlen aus $\mathbb{N}_0$ in die Register zu kodieren sind.

In der theoretischen Informatik verwendet man für Überlegungen meist Turingmaschinen anstatt Registermaschinen. Man kann aber beweisen, dass sich alles, was sich mit einer Turingmaschine bewerkstelligen lässt, auch mit einer Registermaschine erreichen lässt, und umgekehrt.

Churchsche These

Churchsche These

Die Menge der durch eine Turingmaschine berechenbaren Abbildungen [also die formalen Algorithmen] stimmt genau mit der Menge der intuitiv berechenbaren Abbildungen [also den intuitiven Algorithmen] überein.

Übersicht

1 Definition
- Intuitive Algorithmen
- Registermaschinen
- Formale Algorithmen
- Imperative Programme

Imperative Programme

Wenn wir eine Lösungsvorschrift für ein echtes Problem angeben, wollen wir normalerweise kein Registermaschinenprogramm und keine Turingmaschine angeben; stattdessen verwenden wir meistens imperative Programmiersprachen. Wir definieren im Folgenden einen sehr einfachen Vertreter.

Syntax

Definition

Die **Menge der imperativen Programme** IP ist induktiv definiert:

- $\varepsilon \in IP$ (**Leere Anweisung**).
  Häufig schreiben wir anstelle von ε auch gar nichts.
- `Variable := Term` $\in IP$ wobei $Variable$ eine Variable und $Term$ ein Term ist (**Zuweisung**).
- Mit $\mathscr{S}, \mathscr{T} \in IP$ und einer logischen Formel $\mathscr{B}$ sind auch folgende Ausdrücke Elemente von IP:
  - $\mathscr{S}$ `;` $\mathscr{T}$ (**Konkatenation**)
  - `FALLS` $\mathscr{B}$ `DANN` $\mathscr{S}$ `SONST` $\mathscr{T}$ `SLLAF` (**Auswahl**)
  - `SOLANGE` $\mathscr{B}$ `FÜHRE AUS` $\mathscr{S}$ `EGNALOS` (**Schleife**)

Syntax

Beispiel

```
FALLS (x > 0)
  DANN d := (y*x)
  SONST d := (-1*(y*x))
SLLAF
```

Beispiel

```
Produkt := 0;
h := b;
SOLANGE (h > 0) FÜHRE AUS
  Produkt := (Produkt + a);
  h := (h - 1)
EGNALOS
```

Semantik

Leere Anweisung

- Der erste Basisfall ist die leere Anweisung:

$$\varepsilon$$

- Die Bedeutung: Tue nichts.
- Danach ist der Algorithmus beendet.

Semantik

Zuweisung

- Der zweite Basisfall ist die Zuweisung:

```
Variable := Term
```

- Die Bedeutung: Weise der Variablen $Variable$ den ausgewerteten Term $Term$ zu.
- Danach ist der Algorithmus beendet.

Konkatenation

- Der erste zusammengesetzte Fall ist die Konkatenation:

$$\mathscr{S}\,;\ \mathscr{T}$$

- Die Bedeutung: Führe erst $\mathscr{S}$ aus und danach $\mathscr{T}$.
- Danach ist der Algorithmus beendet.

Auswahl

- Der zweite zusammengesetzte Fall ist die Auswahl:

$$\texttt{FALLS } \mathscr{B} \texttt{ DANN } \mathscr{S} \texttt{ SONST } \mathscr{T} \texttt{ SLLAF}$$

- Die Bedeutung: Prüfe die Bedingung $\mathscr{B}$. Wenn sie zutrifft, führe $\mathscr{S}$ aus. Wenn sie nicht zutrifft, führe $\mathscr{T}$ aus.
- Danach ist der Algorithmus beendet.

Semantik

Schleife

- Der dritte zusammengesetzte Fall ist die Schleife:

  $$\texttt{SOLANGE } \mathscr{B} \texttt{ FÜHRE AUS } \mathscr{S} \texttt{ EGNALOS}$$

- Die Bedeutung: Prüfe die Bedingung $\mathscr{B}$. Wenn sie zutrifft, führe $\mathscr{S}$ aus. Prüfe die Bedingung $\mathscr{B}$. Wenn sie zutrifft, führe $\mathscr{S}$ aus ...
  Tue dies solange, bis die Bedingung $\mathscr{B}$ nicht zutrifft.
- Danach ist der Algorithmus beendet.

Die Bedingung muss nicht unbedingt falsch werden. In dem Falle terminiert der Algorithmus nicht.

Andere Schleifentypen

In vielen Algorithmen verwendet man häufig noch andere Schleifenkonstrukte, um das Verständnis zu fördern.

Zum einen gibt es Zählschleifen der Form

$$\texttt{FÜR n := 1 BIS 100 FÜHRE AUS } \mathscr{S} \texttt{ RÜF},$$

die beim Lesen den Sinn der Schleife verdeutlichen. Diese Art der Schleife kann aber problemlos in eine Schleife der Art

$$\texttt{SOLANGE } \mathscr{B} \texttt{ FÜHRE AUS } \mathscr{S} \texttt{ EGNALOS}$$

überführt werden.

Zum anderen gibt es den Typ

$$\texttt{FÜHRE AUS } \mathscr{S} \texttt{ BIS GILT } \mathscr{B} \texttt{ ERHÜF}.$$

Auch dieser lässt sich in unseren Basistyp umwandeln.

Übersicht

1 Definition

2 Komplexität

3 Berechenbarkeit

Komplexität

Bei Algorithmen interessieren wir uns für die Laufzeit und den Speicherbedarf. Für die Laufzeit betrachten wir die Anzahl an charakteristischen Anweisungen, etwa Vergleiche oder Zuweisungen. Für den Speicherbedarf betrachten wir die Anzahl der belegten Speicherstellen.

Im folgenden beschränken wir uns auf die Laufzeit, die Betrachtungen für den Speicherbedarf bei Algorithmen laufen analog.

Vergleichen wir nun zwei Algorithmen, um die Teilsummen von Feldern zu berechnen (das Konstrukt `GIB AUS` ist eine Ausgabeanweisung, die den Zustand unseres Programmes nicht verändert).

Komplexität

Beispiel

```
Eingabe: n, Feld der Länge n

FÜR i := 1 BIS n FÜHRE AUS
  Summe := 0;
  FÜR j := 1 BIS i FÜHRE AUS
    Summe := Summe + Feld[j]
  RÜF;
  GIB AUS Summe
RÜF
```

Komplexität

Beispiel

```
Eingabe: n, Feld der Länge n

Summe := 0;
FÜR i := 1 BIS n FÜHRE AUS
  Summe := Summe + Feld[i];
  GIB AUS Summe
RÜF
```

Komplexität

Im ersten Algorithmus werden

$$\sum_{i=1}^{n}\left[1+\sum_{j=1}^{i}1\right]=n+\sum_{i=1}^{n}i=n+\frac{n\cdot(n+1)}{2}=\frac{n^2+3n}{2}$$

Zuweisungen ausgeführt.

Im zweiten Algorithmus werden $1+n$ Zuweisungen ausgeführt.

Der zweite Algorithmus ist offensichtlich der effizientere. Für einen groben Vergleich ist es allerdings unerheblich, ob beim ersten Algorithmus $\frac{n^2+3n}{2}$ oder $\frac{n^2}{2}$ Zuweisungen ausgeführt werden. Der quadratische Summand wächst so schnell (für wachsendes n), dass der lineare Anteil nicht ins Gewicht fällt (für große n). Ebenso ist es für den Vergleich unerheblich, ob $\frac{n^2}{2}$ oder n^2 Zuweisungen ausgeführt werden. Ein konstanter Faktor fällt für unsere Betrachtungen ebenfalls nicht ins Gewicht.

Asymptotisches Verhalten

Wir interessieren uns bei Laufzeitanalysen von Algorithmen nur für das asymptotische Verhalten; das ist das Verhalten bei wachsender Eingabegröße. Die Größe der Eingabe (etwa die Anzahl der zu sortierenden Elemente oder die Anzahl der Knoten in einem Graphen oder die Länge eines Textes) setzen wir in Bezug zur Anzahl der Zuweisungen oder Vergleiche. Wir tun dies durch eine Funktion von $\mathbb{N}$ nach $\mathbb{N}$.

Wir interessieren uns weiterhin nur für den »dominanten Term« in der Laufzeit (etwa den höchsten Exponenten bei einem Polynom); schlussendlich interessieren wir uns nämlich nur dafür, ob ein Algorithmus etwa eine logarithmische, lineare, quadratische, kubische oder exponentielle Laufzeit besitzt. Als Hilfsmittel dafür dienen uns die Landausymbole.

Vorbereitung

Zum besseren Verständnis für den Formalismus der Landausymbole betrachten wir zunächst folgende Mengen:

Es sei $x \in \mathbb{N}$.

$$M(x) := \{y \in \mathbb{N} \mid y \text{ ist durch } x \text{ teilbar}\}$$

Für ein $x \in \mathbb{N}$ erhalten wir mit $M(x)$ also eine Menge von Elementen aus $\mathbb{N}$. Zwei Beispiele:

$$M(3) = \{3, 6, 9, 12, ...\}$$
$$M(5) = \{5, 10, 15, 20, ...\}.$$

Bei den Landausymbolen definieren wir nun für ein $g \in Abb(\mathbb{N}, \mathbb{N})$ eine Menge von Elementen aus $Abb(\mathbb{N}, \mathbb{N})$.

Anders formuliert: Für eine Abbildung $g : \mathbb{N} \to \mathbb{N}$ definieren wir eine Menge von Abbildungen $: \mathbb{N} \to \mathbb{N}$.

Landausymbole

Es sei $g : \mathbb{N} \to \mathbb{N}$.

$$\mathrm{O}(g) := \{f : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{N} : \exists k \in \mathbb{N} : \forall n \geq k : f(n) \leq c \cdot g(n)\}$$
$$\Omega(g) := \{f : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{N} : \exists k \in \mathbb{N} : \forall n \geq k : c \cdot f(n) \geq g(n)\}$$
$$\Theta(g) := \mathrm{O}(g) \cap \Omega(g)$$

Es gilt etwa $n \mapsto 25 \in \mathrm{O}(n \mapsto n^2)$.

Wenn $f \in \mathrm{O}(g)$ gilt, dann wächst g nicht langsamer als f.
Wenn $f \in \Omega(g)$ gilt, dann wächst f nicht langsamer als g.
Wenn $f \in \Theta(g)$ gilt, dann wachsen f und g etwa gleichschnell.

Bei dieser Betrachtung fallen alle »Terme geringerer Ordnung« weg ebenso wie multiplikative Konstanten.

Im Beispiel gilt für die Laufzeit f_1 des ersten Algorithmus $f_1 \in \Theta(n \mapsto n^2)$, und für die Laufzeit f_2 des zweiten Algorithmus $f_2 \in \Theta(n \mapsto n)$.

Landausymbole

Beispiel

$$
\begin{aligned}
\mathrm{O}(n \mapsto n^2) = \{ & n \mapsto 25, \\
& n \mapsto \lceil \log n \rceil + 13, \\
& n \mapsto n, \\
& n \mapsto n + 13, \\
& n \mapsto n \cdot \lceil \log n \rceil + 3 \cdot n + 2, \\
& n \mapsto \left\lceil \frac{1}{2} \cdot n^2 \right\rceil, \\
& n \mapsto \frac{n^2 + 3n}{2}, \\
& n \mapsto n^2, \\
& n \mapsto 3 \cdot n^2 + 4 \cdot n + 2, \\
& n \mapsto 3096 \cdot n^2 + 1044 \cdot n + 3 \cdot \lceil \log n \rceil, \\
& \ldots \}
\end{aligned}
$$

Landausymbole

Beispiel

$$\begin{aligned}\Omega(n \mapsto n^2) = \{&n \mapsto \left\lceil \frac{1}{2} \cdot n^2 \right\rceil, \\ &n \mapsto \frac{n^2 + 3n}{2}, \\ &n \mapsto n^2, \\ &n \mapsto 3 \cdot n^2 + 4 \cdot n + 2, \\ &n \mapsto 3096 \cdot n^2 + 1044 \cdot n + 3 \cdot \lceil \log n \rceil, \\ &n \mapsto n^2 \cdot \lceil \log n \rceil + n, \\ &n \mapsto n^3, \\ &n \mapsto n^5 + 26 \cdot n^2, \\ &n \mapsto 2^n, \\ &n \mapsto 3^n + 25 \cdot n, \\ &...\}\end{aligned}$$

Landausymbole

Beispiel

$$\begin{aligned}\Theta(n \mapsto n^2) = \{&n \mapsto \left\lceil \frac{1}{2} \cdot n^2 \right\rceil, \\ &n \mapsto \frac{n^2 + 3n}{2}, \\ &n \mapsto n^2, \\ &n \mapsto 3 \cdot n^2 + 4 \cdot n + 2, \\ &n \mapsto 3096 \cdot n^2 + 1044 \cdot n + 3 \cdot \lceil \log n \rceil, \\ &...\}\end{aligned}$$

Laufzeiten

Die meisten Algorithmen erhalten nicht nur eine Zahl n als Eingabe. Ein Sortieralgorithmus etwa erhält ein Feld der Länge n, das er sortieren soll. Es gibt Algorithmen, die ein bereits sortiertes Feld sehr viel schneller sortieren können als ein unsortiertes Feld (viel überraschender ist eigentlich, dass es auch Algorithmen gibt, bei denen das nicht so ist). Bei solchen Algorithmen ist es nicht möglich, die Laufzeit in Abhängigkeit von n anzugeben. Stattdessen betrachtet man den besten Fall, durchschnittlichen Fall oder schlimmsten Fall.

Laufzeiten – Optimismus und Pessimismus

Der beste Fall ist derjenige mit Eingabelänge n, bei dem der Algorithmus die kürzestmögliche Zeit benötigt. Bei den meisten Sortieralgorithmen ist dies ein bereits sortiertes Feld.

Der durchschnittliche Fall existiert meist nicht (so wie es auch meist keinen Menschen mit einem Durchschnittseinkommen gibt). Es wird vielmehr über alle möglichen Eingaben der Länge n eine Aussage gemacht: Die Laufzeiten des Algorithmus für all diese Eingaben werden addiert und durch die Anzahl dieser Eingaben geteilt (es gibt nur endlich viele Eingaben der Länge n). Das Ergebnis ist dann die durchschnittliche Laufzeit des Algorithmus für Eingaben der Länge n.

Der schlimmste Fall ist derjenige mit Eingabelänge n, bei dem der Algorithmus die längstmögliche Zeit benötigt. Bei einigen Sortieralgorithmen ist dies ein umgekehrt sortiertes Feld.

Laufzeiten – Optimismus und Pessimismus

Es gibt einen Sortieralgorithmus, dessen Laufzeit im durchschnittlichen Falle in $\Theta(n \mapsto n \cdot \lceil \operatorname{ld} n \rceil)$ liegt, dessen Laufzeit im schlimmsten Falle jedoch in $\Theta(n \mapsto n^2)$ liegt.

Außerdem gibt es einen Sortieralgorithmus, dessen Laufzeit sogar im schlimmsten Falle in $\Theta(n \mapsto n \cdot \lceil \operatorname{ld} n \rceil)$ liegt.

Dennoch wird häufig der erste Algorithmus eingesetzt, da die multiplikativen Konstanten, die wir hier ignorieren, bei diesem kleiner sind. In den meisten Fällen ist der erste Algorithmus damit schneller als der zweite. Unsere Lebenserfahrung sagt uns jedoch, dass wir die schlimmstmögliche Eingabe erhalten, wenn es denn mal darauf ankommt.

Wenn wir also einmal in die Verlegenheit kommen, Sortieralgorithmen zum Einsatz in der Steuerung eines Kernreaktor-Kühlkreislaufes auszusuchen, nehmen wir vielleicht doch besser den Algorithmus, der auch im schlimmsten Falle das garantierte Laufzeitverhalten $\Theta(n \mapsto n \cdot \lceil \operatorname{ld} n \rceil)$ hat.

Die Länge von Zahlen

Im folgenden verzichten wir aus Gründen der Übersicht auf die Rundungsklammern.

Der Vollständigkeit halber sei hier noch auf ein Detail (sogar ein exponentiell großes) aufmerksam gemacht, das wir bisher unterschlagen haben. In der Laufzeitanalyse werden Zahlen in der Eingabe meist binär kodiert. Das bedeutet, dass die Zahl 1024 die Länge 11 hat (nein, nicht Länge 10, es ist ja eine 1 mit zehn Nullen).

Die Länge von Zahlen

Diese Tatsache hat allerdings einen gravierenden Einfluss auf folgendes Beispiel zum Quadrieren einer Zahl.

Beispiel

```
Eingabe: n

Produkt := 0;
FÜR i := 1 BIS n FÜHRE AUS
  Produkt := Produkt + n
RÜF;
GIB AUS Produkt
```

Die Länge von Zahlen

Der Algorithmus erhält die Zahl n als Eingabe, die Eingabelänge ist also $\operatorname{ld} n$. Die Laufzeit ist $n \mapsto n + 1$; ausgedrückt in der Eingabelänge ist sie jedoch $\operatorname{ld} n \mapsto n + 1$, also $\operatorname{ld} n \mapsto 2^{\operatorname{ld} n} + 1$. Damit hat dieser Algorithmus exponentielle Laufzeit in der Eingabelänge. Es ist ja auch anschaulich seltsam, dass der Algorithmus doppelt so lange benötigt, wenn eine doppelt so große Zahl quadriert werden soll.

Das ist keineswegs haarspalterisch: Andernfalls gäbe es einen Algorithmus mit polynomieller Laufzeit, der die Primfaktorzerlegung großer Zahlen (und groß beginnt hier etwa ab 1024 Bit) bestimmt.

Und auch für andere schwere Probleme gäbe es dann »effiziente« Algorithmen. Damit wäre die Definition, was ein effizienter Algorithmus ist, beinahe wertlos.

Landausymbole – Ermahnende Worte

Häufig (eigentlich fast immer) findet man die falsche Schreibweise $f_1 \in \Theta(n^2)$, $f_2 \in \Theta(n)$, wobei n^2 die Abbildung $n \mapsto n^2$ und n die Abbildung $n \mapsto n$ sein soll. Dies ist schon syntaktisch falsch, da n^2 und n keine Abbildungen sind. Richtig schreiben wir:
$f_1 \in \Theta(n \mapsto n^2)$, $f_2 \in \Theta(n \mapsto n)$.

Um das Problem zu verdeutlichen, betrachten wir a^b. Hierbei ist es ein essentieller Unterschied, ob wir $a \mapsto a^b$ oder $b \mapsto a^b$ meinen.

Landausymbole – Ermahnende Worte

Daneben gibt es noch die falsche Notation $f_1(n) \in \mathrm{O}(n \mapsto n^2)$. Sobald wir ein n in unsere Funktion einsetzen, ist es keine Funktion mehr, sondern ein Wert. Links steht also ein Wert, rechts eine Menge von Funktionen. Das ist syntaktisch falsch.

Hierfür mag die Bequemlichkeit ein Grund sein. So will man vielleicht bei obigem ersten Algorithmus schreiben:

$$\left(\sum_{i=1}^{n} 1 + \left(\sum_{j=1}^{i} 1\right)\right) = \ldots = \frac{n^2+3n}{2} \in \Theta(n \mapsto n^2).$$

Eine Alternative wäre eine ad-hoc-Funktionsdefinition:

$$L_1(n) := \left(\sum_{i=1}^{n} 1 + \left(\sum_{j=1}^{i} 1\right)\right) = \ldots = \frac{n^2+3n}{2}, L_1 \in \Theta(n \mapsto n^2).$$

Landausymbole – Ermahnende Worte

Viele Autoren schreiben darüber hinaus auch $f = \mathrm{O}(g)$. Auf der linken Seite steht eine Funktion, auf der rechten Seite eine Menge von Funktionen. Das Gleichheitszeichen ist also grober Unsinn (und deswegen hat es sich vermutlich auch durchgesetzt).

Landausymbole – Ermahnende Worte

Man hüte sich vor Aussagen wie: »Der Algorithmus `S` hat Laufzeit $O(n \mapsto n^4)$, `T` hat Laufzeit $O(n \mapsto n^8)$, deshalb ist `S` asymptotisch (für große Eingaben) schneller.« Es kann sein, dass `T` eigentlich nur quadratische Laufzeit hat, man hat es nur noch nicht zeigen können.

Aufgrund der Inklusion

$$O(n \mapsto n) \subseteq O(n \mapsto n^2) \subseteq ... \subseteq O(n \mapsto n^{17}) \subseteq ... \subseteq O(n \mapsto n^{52})$$

kann man auch über einen Linearzeitalgorithmus sagen: »Der ist schlecht, der ist in $O(n \mapsto n^{52})$«. Vergleiche sollte man immer nur bei Aussagen über die Zugehörigkeit zu einer »Θ-Klasse« treffen. Leider tun das die wenigsten Autoren.

Effizienz

Aus einer bestimmten theoretischen Sicht sind alle Algorithmen, die eine Laufzeit haben, die durch ein Polynom beschränkt wird, effizient. Alles, was stärker wächst (etwa exponentielles Wachstum), ist ineffizient.

Das ist zwar eine sehr grobe Unterscheidung, sie hat aber ihre Berechtigung: Bei einem Algorithmus, dessen Laufzeit beispielsweise exponentiell in der Eingabegröße wächst, kann schon bei moderater Eingabegröße die Laufzeit nur unter Zuhilfenahme von »ein Mehrfaches der Halbwertszeit von Uran« beschrieben werden.

Effizienz

Wir sollten übrigens zwischen der Effizienz eines Problems und der eines Algorithmus unterscheiden. Die Effizienz eines Algorithmus haben wir eben besprochen. Bei der eines Problems müssen wir alle Algorithmen betrachten, die dieses Problem lösen und den schnellsten untersuchen.

Es kann sonst passieren, dass wir ein Problem, das sich leicht lösen lässt (etwa das Sortieren eines Feldes), als schwer ansehen, da der betrachtete Algorithmus aus Spaß erst exponentiell viele Zuweisungen durchführt, die er nicht benötigt (das ist dann zwar ein dummer Algorithmus, aber er löst das Problem).

Effizienz

In der Tat ist es verhältnismäßig leicht, zu zeigen, dass ein Problem etwa höchstens kubischen Aufwand hat – man muss nur einen Algorithmus angeben, der dies leistet. Schwieriger ist es, zu zeigen, dass ein Problem genau kubischen Aufwand hat. Dazu müsste man alle Algorithmen betrachten, die das Problem lösen – und das sind unendlich viele (wir können beliebig viele sinnlose Anweisungen hinzufügen). Man versucht daher theoretisch zu argumentieren, dass man ein Problem nicht besser lösen kann als in einer bestimmten Größenordnung.

So ist die Ausgabekomplexität eine natürliche untere Schranke für Probleme: Wenn n Elemente eingelesen werden und n^2 Elemente ausgegeben werden müssen, dann liegt die Laufzeit eines beliebigen Algorithmus für dieses Problem in $\Omega(n \mapsto n^2)$.

Übersicht

1 Definition

2 Komplexität

3 Berechenbarkeit

Berechenbarkeit

Definition

Eine Abbildung $f : \mathscr{M} \to \mathscr{N}$ heißt **berechenbar**, wenn es einen Algorithmus gibt, der bei Eingabe von $m \in \mathscr{M}$ die Ausgabe $f(m) \in \mathscr{N}$ liefert. Ein- und Ausgabe sind dabei geeignet zu kodieren.

Berechenbarkeit

Satz

Es gibt abzählbar viele Algorithmen.

Beweis.

Wähle ein Alphabet für die Algorithmen (etwa ASCII, also 128 verschiedene Zeichen). Dann gibt es höchstens 128^n verschiedene Algorithmen der Länge n. Schreibe nun alle Algorithmen systematisch auf: Erst den leeren Algorithmus, dann alle Algorithmen der Länge 1, dann alle der Länge 2, ... Auf diese Weise werden alle Algorithmen erfasst. Wir können also jeden Algorithmus mit einer natürlichen Zahl identifizieren. Daher gibt es nur abzählbar viele Algorithmen. □

Berechenbarkeit

Wir sehen uns nun die Menge $\mathrm{Abb}(\mathbb{N}, \{0,1,2,3,4,5,6,7,8,9\})$ an.

Wir beschränken uns auf diese Abbildungen, da etwa beliebige Werte in $\mathbb{R}$ für Rechner problematisch sind – schließlich benötigt man für die Dezimaldarstellung bei den meisten reellen Zahlen unendlich viel Speicher.

Satz

Die Menge $\text{Abb}(\mathbb{N}, \{0,1,2,3,4,5,6,7,8,9\})$ ist überabzählbar.

Beweis.

Sei $f \in \text{Abb}(\mathbb{N}, \{0,1,2,3,4,5,6,7,8,9\})$, also
$f : \mathbb{N} \to \{0,1,2,3,4,5,6,7,8,9\}$.
Fasse eine solche Abbildung als reelle Zahl auf:

$$0, f(1)f(2)f(3)f(4)\ldots$$

Damit erhalten wir alle Zahlen im Intervall [0,1] (manche sogar mehrfach, da die Dezimalbruchentwicklung nicht eindeutig ist, etwa $0,0\overline{9} = 0,1$). Von dieser Menge wissen wir aber, dass sie überabzählbar ist. □

Berechenbarkeit

Wir sehen also: unsere Rechner (und nach der Churchschen These auch wir Menschen) können fast gar nichts. Eine abzählbare Menge verschwindet in einer überabzählbaren fast spurlos.

Das Halteproblem

Um die brennende Relevanz der theoretischen Informatik für die Praxis zu erkennen, betrachten wir folgendes nicht-berechenbare Problem. Es soll gleichzeitig als Ansatz dienen, wie wir die Nicht-Berechenbarkeit von Problemen überhaupt zeigen können.

Das Halteproblem

Gegeben ist ein Algorithmus `S` und eine Eingabe E. Gefragt ist, ob der Algorithmus bei Eingabe von E terminiert oder nicht.

Das Halteproblem

Satz

Das Halteproblem ist nicht berechenbar.

Beweis.

Wir nehmen an, dass das Problem berechenbar sei. Dann gibt es einen Algorithmus `T`, der zwei Eingaben hat: Einen endlichen Zeichenstrom `S` und einen endlichen Zeichenstrom E. `S` behandelt er als Algorithmus und E als Eingabe für `S`. `T` gibt »terminiert« aus, falls `S` mit Eingabe E terminiert, andernfalls gibt `T` »terminiert nicht« aus.

Das Halteproblem

Beweis.

Wir konstruieren nun einen neuen Algorithmus W:

```
Eingabe X

FALLS (T(X, X)="terminiert")
  DANN SOLANGE (1=1) FÜHRE AUS
       EGNALOS
  SONST
SLLAF
```

Das Halteproblem

Beweis.

Nun füttern wir `W` mit der Eingabe `W` und unterscheiden, ob `W` terminiert oder nicht.

- Falls `W` terminiert, dann findet `T` dies heraus (er prüft ja gerade, ob `W` mit der Eingabe `W` terminiert). Also wird `W`

  ```
  SOLANGE (1=1) FÜHRE AUS EGNALOS
  ```

  ausführen, und damit nicht terminieren. Widerspruch.
- Falls `W` nicht terminiert, dann findet `T` dies heraus (er prüft ja gerade, ob `W` mit der Eingabe `W` terminiert). Daher wird `W` den `SONST`-Fall ausführen (also nichts tun) und anschließend terminieren. Widerspruch.

Damit haben wir gezeigt, dass es einen solchen Algorithmus `T` nicht geben kann. □

Das Halteproblem

Zwei kleine Anmerkungen:

Wenn wir einen Algorithmus mit einer Eingabe starten, dann gilt natürlich, dass er entweder terminiert oder nicht terminiert. Wir haben aber gezeigt, dass es keinen Algorithmus geben kann, der für alle Algorithmen ihr Terminierungsverhalten herausfindet. Nach der Churchschen These sind sogar wir Menschen nicht in der Lage, allen Algorithmen anzusehen, ob sie terminieren oder nicht.

Die Aussage dieses Satzes ist nicht, dass ein Algorithmus niemals einen anderen auf Terminierung testen kann. So könnte ein Algorithmus prüfen, ob ein anderer Algorithmus nur aus

```
SOLANGE (1=1) FÜHRE AUS EGNALOS
```

besteht. Diesen Algorithmus kann er problemlos als nicht terminierend erkennen. Aber es gibt Algorithmen, für die er keine Aussage treffen kann, etwa

```
SOLANGE (2=2) FÜHRE AUS EGNALOS.
```