Informatik II: Algorithmen und Daten-Strukturen

— Sommersemester 2012 —

DHBW Stuttgart

Prof. Dr. Karl Stroetmann

27. Mai 2013

Inhaltsverzeichnis

1 Einführung **4**
- 1.1 Motivation . . . 4
  - 1.1.1 Überblick . . . 4
- 1.2 Algorithmen und Programme . . . 6
- 1.3 Eigenschaften von Algorithmen und Programmen . . . 6
- 1.4 Literatur . . . 7

2 Grenzen der Berechenbarkeit **8**
- 2.1 Das Halte-Problem . . . 8
- 2.2 Unlösbarkeit des Äquivalenz-Problems . . . 11

3 Die $\mathcal{O}$-Notation **13**
- 3.1 Motivation . . . 13
- 3.2 Fallstudie: Effiziente Berechnung der Potenz . . . 18
- 3.3 Der Hauptsatz der Laufzeit-Funktionen . . . 22

4 Der Hoare-Kalkül **29**
- 4.1 Vor- und Nachbedingungen . . . 29
  - 4.1.1 Spezifikation von Zuweisungen . . . 30
  - 4.1.2 Die Abschwächungs-Regel . . . 32
  - 4.1.3 Zusammengesetze Anweisungen . . . 32
  - 4.1.4 Alternativ-Anweisungen . . . 33
  - 4.1.5 Schleifen . . . 35
- 4.2 Der Euklid'sche Algorithmus . . . 35
  - 4.2.1 Nachweis der Korrektheit des Euklid'schen Algorithmus . . . 35
  - 4.2.2 Maschinelle Programm-Verifikation . . . 40
- 4.3 Symbolische Programm-Ausführung . . . 42

4 Abstrakte Daten-Typen und elementare Daten-Stukturen **49**
- 4.1 Abstrakte Daten-Typen . . . 49
- 4.2 Darstellung abstrakter Daten-Typen in *Java* . . . 51
- 4.3 Implementierung eines Stacks mit Hilfe eines *Arrays* . . . 54
- 4.4 Eine Listen-basierte Implementierung von Stacks . . . 56
- 4.5 Auswertung arithmetischer Ausdrücke . . . 58
  - 4.5.1 Ein einführendes Beispiel . . . 59
  - 4.5.2 Ein Algorithmus zur Auswertung arithmetischer Ausdrücke . . . 61
- 4.6 Nutzen abstrakter Daten-Typen . . . 66

5 Sortier-Algorithmen **68**
- 5.1 Sortieren durch Einfügen . . . 69
  - 5.1.1 Komplexität . . . 70
- 5.2 Sortieren durch Auswahl . . . 72

5.2.1 Komplexität . . . 74
5.2.2 Eine feldbasierte Implementierung . . . 74
5.3 Sortieren durch Mischen . . . 76
5.3.1 Komplexität . . . 79
5.3.2 Eine feldbasierte Implementierung . . . 82
5.3.3 Eine nicht-rekursive Implementierung von *Sortieren durch Mischen* . . . 84
5.4 Der *Quick-Sort*-Algorithmus . . . 86
5.4.1 Komplexität . . . 87
5.4.2 Eine feldbasierte Implementierung von *Quick-Sort* . . . 91
5.4.3 Korrektheit . . . 94
5.4.4 Mögliche Verbesserungen . . . 96
5.5 Timsort . . . 98
5.5.1 Bewertung . . . 111

7 Mengen und Abbildungen 128
7.1 Der abstrakte Daten-Typ der *Abbildung* . . . 128
7.2 Geordnete binäre Bäume . . . 130
7.2.1 Implementierung geordneter binärer Bäume in *Java* . . . 133
7.2.2 Analyse der Komplexität . . . 138
7.3 AVL-Bäume . . . 143
7.3.1 Implementierung von AVL-Bäumen in *Java* . . . 147
7.3.2 Analyse der Komplexität . . . 152
7.4 Tries . . . 155
7.4.1 Einfügen in Tries . . . 157
7.4.2 Löschen in Tries . . . 158
7.4.3 Implementierung in *Java* . . . 159
7.5 Hash-Tabellen . . . 163
7.6 Mengen und Abbildungen in Java . . . 170
7.6.1 Das Interface `Collection<E>` . . . 170
7.6.2 Anwendungen von Mengen . . . 175
7.6.3 Die Schnittstelle `Map<K,V>` . . . 176
7.6.4 Anwendungen . . . 179
7.7 Das Wolf-Ziege-Kohl-Problem . . . 180
7.7.1 Die Klasse `ComparableSet` . . . 180
7.7.2 Die Klasse `ComparableList` . . . 189
7.7.3 Lösung des Wolf-Ziege-Kohl-Problems in *Java* . . . 192

8 Prioritäts-Warteschlangen 197
8.1 Definition des ADT *PrioQueue* . . . 197
8.2 Die Daten-Struktur *Heap* . . . 199
8.3 Implementierung in *Java* . . . 202
8.3.1 Implementierung der Methode *change* . . . 204
8.3.2 Prioritäts-Warteschlangen in *Java* . . . 212

9 Daten-Kompression 218
9.1 Der Algorithmus von Huffman . . . 219
9.1.1 Implementierung in *Java* . . . 223
9.2 Optimalität des Huffman'schen Kodierungsbaums . . . 227

10 Graphentheorie **232**
10.1 Die Berechnung kürzester Wege 232
10.1.1 Ein naiver Algorithmus zur Lösung des kürzeste-Wege-Problems 233
10.1.2 Der Algorithmus von Moore 234
10.1.3 Der Algorithmus von Dijkstra 236
10.1.4 Implementierung in *Java* 237
10.1.5 Komplexität 240

11 Die Monte-Carlo-Methode **237**
11.1 Berechnung der Kreiszahl π 237
11.2 Theoretischer Hintergrund 239
11.3 Erzeugung zufälliger Permutationen 241

Kapitel 1

Einführung

1.1 Motivation

Im ersten Semester haben wir gesehen, wie sich Probleme durch die Benutzung von Mengen und Relationen formulieren und lösen lassen. Eine Frage blieb dabei allerdings unbeantwortet: Mit welchen Datenstrukturen lassen sich Mengen und Relationen am besten darstellen und mit welchen Algorithmen lassen sich die Operationen, mit denen wir in der Mengenlehre gearbeitet haben, am effizientesten realisieren? Die Vorlesung *Algorithmen und Datenstrukturen* beantwortet diese Frage sowohl für die Datenstrukturen Mengen und Relationen als auch für einige andere Datenstrukturen, die in der Informatik eine wichtige Rolle spielen.

1.1.1 Überblick

Die Vorlesung *Algorithmen und Datenstrukturen* beschäftigt sich mit dem Design und der Analyse von Algorithmen und den diesen Algorithmen zugrunde liegenden Daten-Strukturen. Im Detail werden wir die folgenden Themen behandeln:

1. Unlösbarkeit des Halte-Problems

   Zu Beginn der Vorlesung zeigen wir die Grenzen der Berechenbarkeit auf und beweisen, dass es praktisch relevante Funktionen gibt, die sich nicht durch Programme berechnen lassen. Konkret werden wir zeigen, dass es kein SetlX-Programm gibt, dass für eine gegebene SetlX-Funktion f und ein gegebenes Argument s entscheidet, ob der Aufruf $f(s)$ terminiert.

2. Komplexität von Algorithmen

   Um die Komplexität von Algorithmen behandeln zu können, führen wir zwei Hilfsmittel aus der Mathematik ein.

   (a) *Rekurrenz-Gleichungen* sind die diskrete Varianten der Differential-Gleichungen. Diese Gleichungen treten bei der Analyse des Rechenzeit-Verbrauchs rekursiver Funktionen auf.

   (b) Die *$\mathcal{O}$-Notation* wird verwendet, um das Wachstumsverhalten von Funktionen kompakt beschreiben zu können. Sie bieten die Möglichkeit, bei der Beschreibung des des Rechenzeit-Verbrauchs eines Algorithmus von unwichtigen Details abstrahieren zu können.

3. Abstrakte Daten-Typen

   Beim Programmieren treten bestimmte Daten-Strukturen in ähnlicher Form immer wieder auf. Diesen Daten-Strukturen liegen sogenannte *abstrakte Daten-Typen* zugrunde. Als konkretes Beispiel stellen wir in diesem Kapitel den abstrakten Daten-Typ *Stack* vor.

Dieser Teil der Vorlesung überlappt sich mit der Vorlesung zur Sprache *Java*, denn abstrakte Datentypen sind eine der Grundlagen der Objekt-orientierten Programmierung.

4. Sortier-Algorithmen

   Sortier-Algorithmen sind die in der Praxis mit am häufigsten verwendeten Algorithmen. Da Sortier-Algorithmen zu den einfacheren Algorithmen gehören, bieten Sie sich als Einstieg in die Theorie der Algorithmen an. Wir behandeln im einzelnen die folgenden Sortier-Algorithmen:

   (a) Sortieren durch Einfügen (engl. *insertion sort*),

   (b) Sortieren durch Auswahl (engl. *min sort*),

   (c) Sortieren durch Mischen (engl. *merge sort*),

   (d) Den *Quick-Sort*-Algorithmus von C. A. R. Hoare.

5. Hoare-Kalkül

   Die wichtigste Eigenschaft eines Algorithmus' ist seine Korrektheit. Der *Hoare-Kalkül* ist ein Verfahren, mit dessen Hilfe die Frage der Korrektheit eines Algorithmus' auf die Frage der Gültigkeit logischer Formeln reduziert werden kann. An dieser Stelle werden wir eine Brücke zu der im ersten Semester vorgestellten Logik schlagen.

6. Abbildungen

   Abbildungen (in der Mathematik auch als Funktionen bezeichnet) spielen nicht nur in der Mathematik sondern auch in der Informatik eine wichtige Rolle. Wir behandeln die verschiedene Daten-Strukturen, mit denen sich Abbildungen realisieren lassen. Im einzelnen besprechen wir binäre Bäume, AVL-Bäume und Hash-Tabellen.

7. Prioritäts-Warteschlangen

   Die Daten-Struktur der Prioritäts-Warteschlangen spielt einerseits bei der Simulation von Systemen und bei Betriebssystemen eine wichtige Rolle, andererseits benötigen wir diese Datenstruktur bei der Diskussion graphentheoretischer Algorithmen.

8. Graphen

   Graphen spielen in vielen Bereichen der Informatik eine wichtige Rolle. Beispielsweise basieren die Navigationssysteme, die heute in fast allen Autos zu finden sind, auf dem Algorithmus von Dijkstra zur Bestimmung des kürzesten Weges. Wir werden diesen Algorithmus in der Vorlesung herleiten.

9. Monte-Carlo-Simulation

   Viele interessante Fragen aus der Wahrscheinlichkeits-Theorie lassen sich aufgrund ihrer Komplexität nicht analytisch lösen. Als Alternative bietet sich an, durch Simulation eine approximative Lösung zu gewinnen. Als konkretes Beispiel werden wir zeigen, wie komplexe Wahrscheinlichkeiten beim Poker-Spiel durch Monte-Carlo-Simulationen bestimmt werden können.

Ziel der Vorlesung ist nicht primär, dass Sie möglichst viele Algorithmen und Daten-Strukturen kennen lernen. Vermutlich wird es eher so sein, dass Sie viele der Algorithmen und Daten-Strukturen, die Sie in dieser Vorlesung kennen lernen werden, später nie gebrauchen können. Worum geht es dann in der Vorlesung? Das wesentliche Anliegen ist es, Sie mit den *Denkweisen* vertraut zu machen, die bei der Konstruktion und Analyse von Algorithmen verwendet werden. Sie sollen in die Lage versetzt werden, algorithmische Lösungen für komplexe Probleme selbstständig zu entwickeln und zu analysieren. Dabei handelt es sich um einen kreativen Prozeß, der sich nicht in einfachen Kochrezepten einfangen läßt. Wir werden in der Vorlesung versuchen, den Prozess an Hand verschiedener Beispiele zu demonstrieren.

1.2 Algorithmen und Programme

Gegenstand der Vorlesung ist die Analyse von Algorithmen, nicht die Erstellung von Programmen. Es ist wichtig, dass die beiden Begriffe "*Algorithmus*" und "*Programm*" nicht verwechselt werden. Ein *Algorithmus* ist seiner Natur nach zunächst einmal ein abstraktes Konzept, das ein Vorgehen beschreibt um ein gegebenes Problem zu lösen. Im Gegensatz dazu ist ein *Programm* eine konkrete Implementierungen eines Algorithmus. Bei einer solchen Implementierung muss letztlich jedes Detail festgelegt werden, sonst könnte das Programm nicht vom Rechner ausgeführt werden. Bei einem Algorithmus ist das nicht notwendig: Oft wollen wir nur einen Teil eines Vorgehens beschreiben, der Rest interessiert uns nicht, weil beispielsweise ohnehin klar ist, was zu tun ist. Ein Algorithmus läßt also eventuell noch Fragen offen.

In Lehrbüchern werden Algorithmen oft mit Hilfe von *Pseudo-Code* dargestellt. Syntaktische hat Pseudo-Code eine ähnliche Form wie ein Programm. Im Gegensatz zu Programmen kann Pseudo-Code aber auch natürlich-sprachlichen Text beinhalten. Sie sollten sich aber klar machen, dass *Pseudo-Code* genau so wenig ein Algorithmus ist, wie ein Programm ein Algorithmus ist, denn auch der *Pseudo-Code* ist ein konkretes Stück Text, wohingen der Algorithmus eine abstrakte Idee ist. Allerdings bietet der Pseudo-Code dem Informatiker die Möglichkeit, einen Algorithmus auf der Ebene zu beschreiben, die zur Beschreibung am zweckmäßigisten ist, denn man ist nicht durch die Zufälligkeiten der Syntax einer Programmier-Sprache eingeschränkt.

Konzeptuell ist der Unterschied zwischen einem Algorithmus und einem Programm vergleichbar mit dem Unterschied zwischen einer philosophischen Idee und einem Text, der die Idee beschreibt: Die Idee selbst lebt in den Köpfen der Menschen, die diese Idee verstanden haben. Diese Menschen können dann versuchen, die Idee konkret zu fassen und aufzuschreiben. Dies kann in verschiedenen Sprachen und mit verschiedenen Worten passieren, es bleibt die selbe Idee. Genauso kann ein Algorithmus in verschiedenen Programmier-Sprachen kodiert werden, es bleibt der selbe Algorithmus.

Nachdem wir uns den Unterschied zwischen einem Algorithmus und einem Programm diskutiert haben, überlegen wir uns, wie wir Algorithmen beschreiben können. Zunächst einmal können wir versuchen, Algorithmen durch natürliche Sprache zu beschreiben. Natürliche Sprache hat den Vorteil, dass Sie sehr ausdrucksstark ist: Was wir nicht mit natürlicher Sprache ausdrücken können, können wir überhaupt nicht ausdrücken. Der Nachteil der natürlichen Sprache besteht darin, dass die Bedeutung nicht immer eindeutig ist. Hier hat eine Programmier-Sprache den Vorteil, dass die Semantik wohldefiniert ist. Allerdings ist es oft sehr mühselig, einen Algorithmus vollständig auszukodieren, denn es müssen dann Details geklärt werden, die für das Prinzip vollkommen unwichtig sind. Es gibt noch eine dritte Möglichkeit, Algorithmen zu beschreiben und das ist die Sprache der Mathematik. Die wesentlichen Elemente dieser Sprache sind die Prädikaten-Logik und die Mengen-Lehre. In diesem Skript werden wir die Algorithmen in dieser Sprache beschreiben. Um diese Algorithmen dann auch ausprobieren zu können, müssen wir sie in eine Programmier-Sprache übersetzen. Hier bietet sich SetlX an, denn diese Programmier-Sprache stellt die Daten-Strukturen Mengen und Funktionen, die in der Mathematik allgegenwärtig sind, zur Verfügung. Sie werden sehen, dass es in SetlX möglich ist, die Algorithmen auf einem sehr hohen Abstraktions-Niveau darzustellen. Eine Implementierung der Algorithmen in `C++` oder *Java* ist erheblich aufwendiger.

1.3 Eigenschaften von Algorithmen und Programmen

Bevor wir uns an die Konstruktion von Algorithmen machen, sollten wir uns überlegen, durch welche Eigenschaften Algorithmen charakterisiert werden und welche dieser Eigenschaften erstrebenswert sind.

1. Algorithmen sollen *korrekt* sein.
2. Algorithmen sollen *effizient* sein.
3. Algorithmen sollen möglichst **einfach** sein.

Die erste dieser Forderung ist so offensichtlich, dass sie oft vergessen wird: Das schnellste Programm nutzt nichts, wenn es falsche Ergebnisse liefert. Nicht ganz so klar ist die letzte Forderung. Diese Forderung hat einen ökonomischen Hintergrund: Genauso wie die Rechenzeit eines Programms Geld kostet, so kostet auch die Zeit, die Programmierer brauchen um ein Programm zu erstellen und zu warten, Geld. Aber es gibt noch zwei weitere Gründe für die dritte Forderung:

1. Für einen Algorithmus, dessen konzeptionelle Komplexität hoch ist, ist die Korrektheit nicht mehr einsehbar und damit auch nicht gewährleistbar.

2. Selbst wenn der Algorithmus an sich korrekt ist, so kann doch die Korrektheit der Implementierung nicht mehr sichergestellt werden.

1.4 Literatur

Ergänzend zu diesem Skript möchte ich die folgende Literatur empfehlen.

1. *Alfred V. Aho*, *John E. Hopcraft*, and *Jeffrey D. Ullman*: *Data Structures and Algorithms*, Addison-Wesley, 1987.

   Dieses Buch gehört zu den Standardwerken über Algorithmen. Die Algorithmen werden dort auf einem hohen Niveau erklärt.

2. *Frank M. Carrano* and *Janet J. Prichard*: *Data Abstraction and Problem Solving with Java*, Addison-Wesley, 2003.

   In diesem Buch sind die Darstellungen der Algorithmen sehr breit und verständlich. Viele Algorithmen sind graphisch illustriert. Leider geht das Buch oft nicht genug in die Tiefe, so wird zum Beispiel die Komplexität von Algorithmen kaum untersucht.

3. *Thomas H. Cormen*, *Charles E. Leiserson*, *Ronald L. Rivest*, and *Clifford Stein*: *Introduction to Algorithms*, MIT Press, 2001.

   Aufgrund seiner Ausführlichkeit eignet sich dieses Buch sehr gut zum Nachschlagen von Algorithmen. Die Darstellungen der Algorithmen sind eher etwas knapper gehalten, dafür wird aber auch die Komplexität analysiert.

4. *Robert Sedgewick*: *Algorithms in Java*, Pearson, 2002.

   Dieses Buch liegt in der Mitte zwischen den Büchern von Carrano und Cormen: Es ist theoretisch nicht so anspruchsvoll wie das von Cormen, enthält aber wesentlich mehr Algorithmen als das Buch von Carrano. Zusätzlich wird die Komplexität der Algorithmen diskutiert.

5. *Heinz-Peter Gumm und Manfred Sommer*, *Einführung in die Informatik*, Oldenbourg Verlag, 2006.

   Dieses Buch ist eine sehr gute Einführung in die Informatik, die auch ein umfangreiches Kapitel über Algorithmen und Datenstrukturen enthält. Die Darstellung der Algorithmen ist sehr gelungen.

Kapitel 2

Grenzen der Berechenbarkeit

In jeder Disziplin der Wissenschaft wird die Frage gestellt, welche Grenzen die verwendeten Methoden haben. Wir wollen daher in diesem Kapitel beispielhaft ein Problem untersuchen, bei dem die Informatik an ihre Grenzen stößt. Es handelt sich um das Halte-Problem.

2.1 Das Halte-Problem

Das Halte-Problem ist die Frage, ob eine gegebene Funktion für eine bestimmte Eingabe terminiert. Wir werden zeigen, dass dieses Problem nicht durch ein Programm gelöst werden kann. Dazu führen wir folgende Definition ein.

Definition 1 (Test-Funktion) *Ein String t ist eine* Test-Funktion *mit Namen n wenn t die Form*

$$n \texttt{ := procedure(x) } \{ \cdots \}$$

hat, und sich als Definition einer SetlX*-Funktion parsen läßt. Die Menge der Test-Funktionen bezeichnen wir mit TF. Ist $t \in TF$ und hat den Namen n, so schreiben wir* $\texttt{name}(t) = n$. □

Beispiele:

1. s_1 = "`simple := procedure(x) { return 0; }`"
   s_1 ist eine (sehr einfache) Test-Funktion mit dem Namen `simple`.
2. s_2 = "`loop := procedure(x) { while (true) { x := x + 1; } }`"
   s_2 ist eine Test-Funktion mit dem Namen `loop`.
3. s_3 = "`hugo := procedure(x) { return ++x; }` "
   s_3 ist keine Test-Funktion, denn da SetlX den Präfix-Operator "`++`" nicht unterstützt, läßt sich der String s_3 nicht fehlerfrei parsen.

Um das Halte-Problem übersichtlicher formulieren zu können, führen wir noch drei zusätzliche Notationen ein.

Notation 2 ($\rightsquigarrow$, $\downarrow$, $\uparrow$) *Ist n der Name einer C-Funktion und sind a_1, $\cdots$, a_k Argumente, die vom Typ her der Deklaration von n entsprechen, so schreiben wir*

$$n(a_1, \cdots, a_k) \rightsquigarrow r$$

wenn der Aufruf $n(a_1, \cdots, a_k)$ das Ergebnis r liefert. Sind wir an dem Ergebnis selbst nicht interessiert, sondern wollen nur angeben, daß ein Ergebnis existiert, so schreiben wir

$$n(a_1, \cdots, a_k) \downarrow$$

und sagen, dass der Aufruf $n(a_1, \cdots, a_k)$ terminiert. *Terminiert der Aufruf* $n(a_1, \cdots, a_k)$ *nicht, so schreiben wir*

$$n(a_1, \cdots, a_k) \uparrow$$

und sagen, dass der Aufruf $n(a_1, \cdots, a_k)$ divergiert. □

Beispiele: Legen wir die Funktions-Definitionen zugrunde, die wir im Anschluß an die Definition des Begriffs der Test-Funktion gegeben haben, so gilt:

1. `simple("emil")` $\rightsquigarrow 0$
2. `simple("emil")` $\downarrow$
3. `loop("hugo")` $\uparrow$

Das *Halte-Problem* für SetlX-Funktionen ist die Frage, ob es eine `SetlX`-Funktion

`stops := procedure`$(t,\ a)\ \{\cdots\}$

gibt, die als Eingabe eine Testfunktion t und einen String a erhält und die folgende Eigenschaft hat:

1. $t \notin TF \quad \Leftrightarrow \quad$ `stops`$(t, a) \rightsquigarrow 2$.

   Der Aufruf `stops(`t`, `a`)` liefert genau dann den Wert 2 zurück, wenn t keine Test-Funktion ist.

2. $t \in TF \wedge$ `name`$(t) = n \wedge n(a) \downarrow \quad \Leftrightarrow \quad$ `stops`$(t, a) \rightsquigarrow 1$.

   Der Aufruf `stops(`t`, `a`)` liefert genau dann den Wert 1 zurück, wenn t eine Test-Funktion mit Namen n ist und der Aufruf $n(a)$ terminiert.

3. $t \in TF \wedge$ `name`$(t) = n \wedge n(a) \uparrow \quad \Leftrightarrow \quad$ `stops`$(t, a) \rightsquigarrow 0$.

   Der Aufruf `stops(`t`, `a`)` liefert genau dann den Wert 0 zurück, wenn t eine Test-Funktion mit Namen n ist und der Aufruf $n(a)$ nicht terminiert.

Falls eine SetlX-Funktion `stops` mit den obigen Eigenschaften existiert, dann sagen wir, dass das Halte-Problem für SetlX entscheidbar ist.

Theorem 3 (Turing, 1936) *Das Halte-Problem ist unentscheidbar.*

Beweis: Zunächst eine Vorbemerkung. Um die Unentscheidbarkeit des Halte-Problems nachzuweisen müssen wir zeigen, dass etwas, nämlich eine Funktion mit gewissen Eigenschaften nicht existiert. Wie kann so ein Beweis überhaupt funktionieren? Wie können wir überhaupt zeigen, dass irgendetwas nicht existiert? Die einzige Möglichkeit zu zeigen, dass etwas nicht existiert ist indirekt: Wir nehmen also an, dass eine Funktion `stops` existiert, die das Halte-Problem löst. Aus dieser Annahme werden wir einen Widerspruch ableiten. Dieser Widerspruch zeigt uns dann, dass eine Funktion `stops` mit den gewünschten Eigenschaften nicht existieren kann. Um zu einem Widerspruch zu kommen, definieren wir den String *turing* wie in Abbildung 2.1 gezeigt.

Mit dieser Definition ist klar, dass *turing* eine Test-Funktion mit dem Namen "`alan`" ist:

$$turing \in TF \ \wedge \ \texttt{name}(turing) = \texttt{alan}.$$

Damit sind wir in der Lage, den String *Turing* als Eingabe der Funktion `stops` zu verwenden. Wir betrachten nun den folgenden Aufruf:

`stops(`*turing*`, `*turing*`);`

Offenbar ist *turing* eine Test-Funktion. Daher können nur zwei Fälle auftreten:

$$\texttt{stops}(turing, turing) \rightsquigarrow 0 \quad \vee \quad \texttt{stops}(turing, turing) \rightsquigarrow 1.$$

Diese beiden Fälle analysieren wir nun im Detail:

```
1  turing := "alan := procedure(x) {
2                  result := stops(x, x);
3                  if (result == 1) {
4                      while (true) {
5                          print("... looping ...");
6                      }
7                  }
8                  return result;
9              };"
```

Abbildung 2.1: Die Definition des Strings *turing*.

1. `stops`(*turing*, *turing*) $\rightsquigarrow 0$.

   Nach der Spezifikation von `stops` bedeutet dies

   $$\texttt{alan}(\textit{turing}) \uparrow$$

   Schauen wir nun, was wirklich beim Aufruf `alan(`*turing*`)` passiert: In Zeile 2 erhält die Variable `result` den Wert 0 zugewiesen. In Zeile 3 wird dann getestet, ob `result` den Wert 1 hat. Dieser Test schlägt fehl. Daher wird der Block der `if`-Anweisung nicht ausgeführt und die Funktion liefert als nächstes in Zeile 8 den Wert 0 zurück. Insbesondere terminiert der Aufruf also, im Widerspruch zu dem, was die Funktion `stops` behauptet hat.

   Damit ist der erste Fall ausgeschlossen.

2. `stops`(*turing*, *turing*) $\rightsquigarrow 1$.

   Aus der Spezifikation der Funktion `stops` folgt, dass der Aufruf `alan`(*turing*) terminiert:

   $$\texttt{alan}(\textit{turing}) \downarrow$$

   Schauen wir nun, was wirklich beim Aufruf `alan(`*turing*`)` passiert: In Zeile 2 erhält die Variable `result` den Wert 1 zugewiesen. In Zeile 3 wird dann getestet, ob `result` den Wert 1 hat. Diesmal gelingt der Test. Daher wird der Block der `if`-Anweisung ausgeführt. Dieser Block besteht aber nur aus einer Endlos-Schleife, aus der wir nie wieder zurück kommen. Das steht im Widerspruch zu dem, was die Funktion `stops` behauptet hat.

   Damit ist der zweite Fall ausgeschlossen.

Insgesamt haben wir also in jedem Fall einen Widerspruch erhalten. Also ist die Annahme, dass die SetlX-Funktion `stops` das Halte-Problem löst, falsch. Insgesamt haben wir gezeigt, dass es keine SetlX-Funktion geben kann, die das Halte-Problem löst. □

Bemerkung: Der Nachweis, dass das Halte-Problem unlösbar ist, wurde 1936 von Alan Turing (1912 – 1954) [Tur36] erbracht. Turing hat das Problem damals natürlich nicht für die Sprache SetlX gelöst, sondern für die heute nach ihm benannten *Turing-Maschinen*. Eine Turing-Maschine ist abstrakt gesehen nichts anderes als eine Beschreibung eines Algorithmus. Turing hat also gezeigt, dass es keinen Algorithmus gibt, der entscheiden kann, ob ein gegebener anderer Algorithmus terminiert.

Bemerkung: An dieser Stelle können wir uns fragen, ob es vielleicht eine andere Programmier-Sprache gibt, in der wir das Halte-Problem dann vielleicht doch lösen könnten. Wenn es in dieses Programmier-Sprache Unterprogramme gibt, und wenn wir dort Programm-Texte als Argumente von Funktionen übergeben können, dann ist leicht zu sehen, dass der obige Beweis der Unlösbarkeit des Halte-Problems sich durch geeignete syntaktische Modifikationen auch auf die andere Programmier-Sprache übertragen läßt.

Aufgabe 1: Wir nennen eine Menge X *abzählbar*, wenn es eine Funktion

$$f : \mathbb{N} \to X$$

gibt, so dass es für alle $x \in X$ ein $n \in \mathbb{N}$ gibt, so dass x das Bild von n unter f ist:

$$\forall x \in X : \exists n \in \mathbb{N} : x = f(n).$$

Zeigen Sie, dass die Potenz-Menge $2^{\mathbb{N}}$ der natürlichen Zahlen $\mathbb{N}$ nicht abzählbar ist.

Hinweis: Gehen Sie ähnlich vor wie beim Beweis der Unlösbarkeit des Halte-Problems. Nehmen Sie an, es gäbe eine Funktion f, die die Teilmengen von $\mathbb{N}$ aufzählt:

$$\forall x \in 2^{\mathbb{N}} : \exists n \in \mathbb{N} : x = f(n).$$

Definieren Sie eine Menge `Cantor` wie folgt:

$$\texttt{Cantor} := \big\{n \in \mathbb{N} \mid n \notin f(n)\big\}.$$

Versuchen Sie nun, einen Widerspruch herzuleiten.

2.2 Unlösbarkeit des Äquivalenz-Problems

Es gibt noch eine ganze Reihe anderer Funktionen, die nicht berechenbar sind. In der Regel werden wir den Nachweis, dass eine bestimmt Funktion nicht berechenbar ist, dadurch führen, dass wir zunächst annehmen, dass die gesuchte Funktion doch implementierbar ist. Unter dieser Annahme konstruieren wir dann eine Funktion, die das Halte-Problem löst, was im Widerspruch zu dem am Anfang dieses Abschnitts bewiesen Sachverhalts steht. Dieser Widerspruch zwingt uns zu der Folgerung, dass die gesuchte Funktion nicht implementierbar ist. Wir werden dieses Verfahren an einem Beispiel demonstrieren. Vorweg benötigen wir aber noch eine Definition.

Definition 4 ($\simeq$) *Es seien n_1 und n_2 Namen zweier* SetlX*-Funktionen und $a_1, \cdots, a_k$ seien Argumente, mit denen wir diese Funktionen füttern können. Wir definieren*

$$n_1(a_1, \cdots, a_k) \simeq n_2(a_1, \cdots, a_k)$$

g.d.w. einer der beiden folgen Fälle auftritt:

1. $n_1(a_1, \cdots, a_k)\uparrow \quad \wedge \quad n_2(a_1, \cdots, a_k)\uparrow$
2. $\exists r : \Big(n_1(a_1, \cdots, a_k) \rightsquigarrow r \quad \wedge \quad n_2(a_1, \cdots, a_k) \rightsquigarrow r\Big)$

   In diesem Fall sagen wir, dass die beiden Funktions-Aufrufe $n_1(a_1, \cdots, a_k) \simeq n_2(a_1, \cdots, a_k)$ partiell äquivalent *sind.* □

Wir kommen jetzt zum *Äquivalenz-Problem*. Die Funktion *equal*, die die Form

```
equal := procedure(p1, p2, a) { ... }
```

hat, möge folgender Spezifikation genügen:

1. $p_1 \notin TF \ \vee \ p_2 \notin TF \quad \Leftrightarrow \quad \texttt{equal}(p_1, p_2, a) \rightsquigarrow 2.$
2. Falls
   (a) $p_1 \in TF \ \wedge \ \texttt{name}(p_1) = n_1,$
   (b) $p_2 \in TF \ \wedge \ \texttt{name}(p_2) = n_2$ und
   (c) $n_1(a) \simeq n_2(a)$

   gilt, dann muß gelten:
   $$\texttt{equal}(p_1, p_2, a) \rightsquigarrow 1.$$
3. Ansonsten gilt
   $$\texttt{equal}(p_1, p_2, a) \rightsquigarrow 0.$$

Wir sagen, dass eine Funktion, die der eben angegebenen Spezifikation genügt, das *Äquivalenz-Problem* löst.

Theorem 5 (Rice, 1953) Das Äquivalenz-Problem ist unlösbar.

Beweis: Wir führen den Beweis indirekt und nehmen an, dass es doch eine Implementierung der Funktion `equal` gibt, die das Äquivalenz-Problem löst. Wir betrachten die in Abbildung 2.2 angegeben Implementierung der Funktion `stops`.

```
1   stops := procedure(p, a) {
2       f := "loop := procedure(x) {           \n"
3          + "    while (true) { x := x + x; } \n"
4          + "    return 0;                    \n"
5          + "};                               \n";
6       e := equal(f, p, a);
7       if (e == 2) {
8           return 2;
9       } else {
10          return 1 - e;
11      }
12  }
```

Abbildung 2.2: Eine Implementierung der Funktion `stops`.

Zu beachten ist, dass in Zeile 2 die Funktion `equal` mit einem String aufgerufen wird, der eine Test-Funktion ist, und zwar mit dem Namen `loop`. Diese Test-Funktion hat die folgende Form:

```
loop := procedure(x) { while (1) { x := x + x; } };
```

Es ist offensichtlich, dass die Funktion `loop` für kein Ergebnis terminiert. Ist also das Argument p eine Test-Funktion mit Namen n, so liefert die Funktion `equal` immer dann den Wert 1, wenn $n(a)$ nicht terminiert, andernfalls muß sie den Wert 0 zurück geben. Damit liefert die Funktion `stops` aber für eine Test-Funktion p mit Namen n und ein Argument a genau dann 1, wenn der Aufruf $n(a)$ terminiert und würde folglich das Halte-Problem lösen. Das kann nicht sein, also kann es keine Funktion `equal` geben, die das Äquivalenz-Problem löst. □

Die Unlösbarkeit des Äquivalenz-Problems und vieler weiterer praktisch interessanter Problem folgen aus einem 1953 von Henry G. Rice [Ric53] bewiesenen Satz.

Kapitel 3

Die $\mathcal{O}$-Notation

In diesem Kapitel stellen wir die O-Notation vor. Diese beiden Begriffe benötigen wir, um die Laufzeit von Algorithmen analysieren zu können. Die Algorithmen selber stehen in diesem Kapitel noch im Hintergrund.

3.1 Motivation

Wollen wir die Komplexität eines Algorithmus abschätzen, so wäre ein mögliches Vorgehen wie folgt: Wir kodieren den Algorithmus in einer Programmiersprache und berechnen, wieviele Additionen, Multiplikationen, Zuweisungungen, und andere elementare Operationen bei einer gegebenen Eingabe von dem Programm ausgeführt werden. Anschließend schlagen wir im Prozessor-Handbuch nach, wieviel Zeit die einzelnen Operationen in Anspruch nehmen und errechnen daraus die Gesamtlaufzeit des Programms.[1] Dieses Vorgehen ist aber in zweifacher Hinsicht problematisch:

1. Das Verfahren ist sehr kompliziert.
2. Würden wir den selben Algorithmus anschließend in einer anderen Programmier-Sprache kodieren, oder aber das Programm auf einem anderen Rechner laufen lassen, so wäre unsere Rechnung wertlos und wir müßten sie wiederholen.

Der letzte Punkt zeigt, dass das Verfahren dem Begriff des Algorithmus, der ja eine Abstraktion des Programm-Begriffs ist, nicht gerecht wird. Ähnlich wie der Begriff des Algorithmus von bestimmten Details einer Implementierung abstrahiert brauchen wir zur Erfassung der rechenzeitlichen Komplexität eines Algorithmus einen Begriff, der von bestimmten Details der Funktion, die die Rechenzeit für ein gegebenes Programm berechnet, abstrahiert. Wir haben drei Forderungen an den zu findenden Begriff.

- Der Begriff soll von konstanten Faktoren abstrahieren.
- Der Begriff soll von *unwesentlichen Termen* abstrahieren.

  Nehmen wir an, wir hätten ein Programm, dass zwei $n \times n$ Matrizen multipliziert und wir hätten für die Rechenzeit $T(n)$ dieses Programms in Abhängigkeit von n die Funktion

  $$T(n) = 3 \cdot n^3 + 2 \cdot n^2 + 7$$

  gefunden. Dann nimmt der *proportionale Anteil* des Terms $2 \cdot n^2 + 7$ an der gesamten Rechenzeit mit wachsendem n immer mehr ab. Zur Verdeutlichung haben wir in einer Tabelle die Werte des proportionalen Anteils für $n = 1,\ 10,\ 100,\ 1000,\ 10\,000$ aufgelistet:

[1] Da die heute verfügbaren Prozessoren fast alle mit *Pipelining* arbeiten, werden oft mehrere Befehle gleichzeitig abgearbeitet. Da gleichzeitig auch das Verhalten des Caches eine wichtige Rolle spielt, ist die genaue Berechnung der Rechenzeit faktisch unmöglich.

n	$\dfrac{2 \cdot n^2 + 7}{3 \cdot n^3 + 2 \cdot n^2 + 7}$
1	0.75000000000000
10	0.06454630495800
100	0.00662481908150
1000	0.00066622484855
10 000	6.6662224852 e -05

- Der Begriff soll das *Wachstum* der Rechenzeit abhängig von *Wachstum* der Eingaben erfassen. Welchen genauen Wert die Rechenzeit für kleine Werte der Eingaben hat, spielt nur eine untergeordnete Rolle, denn für kleine Werte der Eingaben wird auch die Rechenzeit nur klein sein.

Wir bezeichnen die Menge der positiven reellen Zahlen mit $\mathbb{R}_+$

$$\mathbb{R}_+ := \{x \in \mathbb{R} \mid x > 0\}.$$

Wir bezeichnen die Menge aller Funktionen von $\mathbb{N}$ nach $\mathbb{R}_+$ mit $\mathbb{R}_+^{\mathbb{N}}$, es gilt also:

$$\mathbb{R}_+^{\mathbb{N}} = \big\{f \mid f \text{ ist Funktion der Form } f : \mathbb{N} \to \mathbb{R}_+\big\}.$$

Definition 6 ($\mathcal{O}(f)$) *Es sei eine Funktion $f \in \mathbb{R}_+^{\mathbb{N}}$ gegeben. Dann definieren wir die Menge der Funktionen, die asymptotisch das gleiche Wachstumsverhalten haben wie die Funktion f, wie folgt:*

$$\mathcal{O}(f) \;:=\; \big\{g \in \mathbb{R}_+^{\mathbb{N}} \mid \exists k \in \mathbb{N}\colon \big(\exists c \in \mathbb{R}\colon \forall n \in \mathbb{N}\colon n \geq k \to g(n) \leq c \cdot f(n)\big)\big\}. \qquad \square$$

Was sagt die obige Definition aus? Zunächst kommt es auf kleine Werte des Arguments n nicht an, denn die obige Formel sagt ja, dass $g(n) \leq c \cdot f(n)$ nur für die n gelten muss, für die $n \geq k$ ist. Außerdem kommt es auf Proportionalitäts-Konstanten nicht an, denn $g(n)$ muss ja nur kleinergleich $c \cdot f(n)$ sein und die Konstante c können wir beliebig wählen. Um den Begriff zu verdeutlichen, geben wir einige Beispiele.

Beispiel: Es gilt

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \in \mathcal{O}(n^3).$$

Beweis: Wir müssen eine Konstante c und eine Konstante k angeben, so dass für alle $n \in \mathbb{N}$ mit $n \geq k$ die Ungleichung

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \leq c \cdot n^3$$

gilt. Wir setzen $k := 1$ und $c := 12$. Dann können wir die Ungleichung

$$1 \leq n \tag{3.1}$$

voraussetzen und müssen zeigen, dass daraus

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \leq 12 \cdot n^3 \tag{3.2}$$

folgt. Erheben wir beide Seiten der Ungleichung (3.1) in die dritte Potenz, so sehen wir, dass

$$1 \leq n^3 \tag{3.3}$$

gilt. Diese Ungleichung multiplizieren wir auf beiden Seiten mit 7 und erhalten:

$$7 \leq 7 \cdot n^3 \tag{3.4}$$

Multiplizieren wir die Ungleichung (3.1) mit $2 \cdot n^2$, so erhalten wir

$$2 \cdot n^2 \leq 2 \cdot n^3 \tag{3.5}$$

Schließlich gilt trivialerweise

$$3 \cdot n^3 \leq 3 \cdot n^3 \tag{3.6}$$

Die Addition der Ungleichungen (3.4), (3.5) und (3.6) liefert nun

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \leq 12 \cdot n^3$$

und das war zu zeigen. □

Beispiel: Es gilt $n \in \mathcal{O}(2^n)$.

Beweis: Wir müssen eine Konstante c und eine Konstante k angeben, so dass für alle $n \geq k$

$$n \leq c \cdot 2^n$$

gilt. Wir setzen $k := 0$ und $c := 1$. Wir zeigen

$$n \leq 2^n \quad \text{für alle } n \in \mathbb{N}$$

durch vollständige Induktion über n.

1. **I.A.**: $n = 0$

   Es gilt $0 \leq 1 = 2^0$.

2. **I.S.**: $n \mapsto n + 1$

   Einerseits gilt nach Induktions-Voraussetzung

   $$n \leq 2^n, \tag{1}$$

   andererseits haben wir

   $$1 \leq 2^n. \tag{2}$$

   Addieren wir (1) und (2), so erhalten wir

   $$n + 1 \leq 2^n + 2^n = 2^{n+1}.$$ □

   Bemerkung: Die Ungleichung $1 \leq 2^n$ hätten wir eigentlich ebenfalls durch Induktion nachweisen müssen.

Aufgabe 2: Zeigen Sie

$$n^2 \in \mathcal{O}(2^n).$$

Wir zeigen nun einige Eigenschaften der $\mathcal{O}$-Notation.

Satz 7 (Reflexivität) *Für alle Funktionen $f: \mathbb{N} \to \mathbb{R}_+$ gilt*

$$f \in \mathcal{O}(f).$$

Beweis: Wählen wir $k := 0$ und $c := 1$, so folgt die Behauptung sofort aus der Ungleichung

$$\forall n \in \mathbb{N}: f(n) \leq f(n).$$ □

Satz 8 (Abgeschlossenheit unter Multiplikation mit Konstanten)
Es seien $f, g: \mathbb{N} \to \mathbb{R}_+$ und $d \in \mathbb{R}_+$. Dann gilt

$$g \in \mathcal{O}(f) \Rightarrow d \cdot g \in \mathcal{O}(f).$$

Beweis: Aus $g \in \mathcal{O}(f)$ folgt, dass es Konstanten $c' \in \mathbb{R}_+$, $k' \in \mathbb{N}$ gibt, so dass

$$\forall n \in \mathbb{N}: \big(n \geq k' \to g(n) \leq c' \cdot f(n)\big)$$

gilt. Multiplizieren wir die Ungleichung mit d, so haben wir

$$\forall n \in \mathbb{N}: \big(n \geq k' \to d \cdot g(n) \leq d \cdot c' \cdot f(n)\big)$$

Setzen wir nun $k := k'$ und $c := d \cdot c'$, so folgt

$$\forall n \in \mathbb{N}: \big(n \geq k \to d \cdot g(n) \leq c \cdot f(n)\big)$$

und daraus folgt $d \cdot g \in \mathcal{O}(f)$. □.

Satz 9 (Abgeschlossenheit unter Addition) *Es seien $f, g, h: \mathbb{N} \to \mathbb{R}_+$. Dann gilt*

$$f \in \mathcal{O}(h) \wedge g \in \mathcal{O}(h) \;\to\; f + g \in \mathcal{O}(h).$$

Beweis: Aus den Voraussetzungen $f \in \mathcal{O}(h)$ und $g \in \mathcal{O}(h)$ folgt, dass es Konstanten $k_1, k_2 \in \mathbb{N}$ und $c_1, c_2 \in \mathbb{R}$ gibt, so dass

$$\forall n \in \mathbb{N}\colon \big(n \geq k_1 \rightarrow f(n) \leq c_1 \cdot h(n)\big) \quad \text{und}$$
$$\forall n \in \mathbb{N}\colon \big(n \geq k_2 \rightarrow g(n) \leq c_2 \cdot h(n)\big)$$

gilt. Wir setzen $k := \max(k_1, k_2)$ und $c := c_1 + c_2$. Für $n \geq k$ gilt dann

$$f(n) \leq c_1 \cdot h(n) \text{ und } g(n) \leq c_2 \cdot h(n).$$

Addieren wir diese beiden Gleichungen, dann haben wir für alle $n \geq k$

$$f(n) + g(n) \leq (c_1 + c_2) \cdot h(n) = c \cdot h(n).$$ □

Satz 10 (Transitivität) *Es seien $f, g, h\colon \mathbb{N} \to \mathbb{R}_+$. Dann gilt*

$$f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h) \;\rightarrow\; f \in \mathcal{O}(h).$$

Beweis: Aus $f \in \mathcal{O}(g)$ folgt, dass es $k_1 \in \mathbb{N}$ und $c_1 \in \mathbb{R}$ gibt, so dass

$$\forall n \in \mathbb{N}\colon \big(n \geq k_1 \rightarrow f(n) \leq c_1 \cdot g(n)\big)$$

gilt und aus $g \in \mathcal{O}(h)$ folgt, dass es $k_2 \in \mathbb{N}$ und $c_2 \in \mathbb{R}$ gibt, so dass

$$\forall n \in \mathbb{N}\colon \big(n \geq k_2 \rightarrow g(n) \leq c_2 \cdot h(n)\big)$$

gilt. Wir definieren $k := \max(k_1, k_2)$ und $c := c_1 \cdot c_2$. Dann haben wir für alle $n \geq k$:

$$f(n) \leq c_1 \cdot g(n) \text{ und } g(n) \leq c_2 \cdot h(n).$$

Die zweite dieser Ungleichungen multiplizieren wir mit c_1 und erhalten

$$f(n) \leq c_1 \cdot g(n) \text{ und } c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n).$$

Daraus folgt aber sofort $f(n) \leq c \cdot h(n)$. □

Satz 11 (Grenzwert-Satz) *Es seien $f, g\colon \mathbb{N} \to \mathbb{R}_+$. Außerdem existiere der Grenzwert*

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

Dann gilt $f \in \mathcal{O}(g)$.

Beweis: Es sei

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

Nach Definition des Grenzwertes gibt es dann eine Zahl $k \in \mathbb{N}$, so dass für alle $n \in \mathbb{N}$ mit $n \geq k$ die Ungleichung

$$\left| \frac{f(n)}{g(n)} - \lambda \right| \leq 1$$

gilt. Multiplizieren wir diese Ungleichung mit $g(n)$, so erhalten wir

$$|f(n) - \lambda \cdot g(n)| \leq g(n).$$

Daraus folgt wegen

$$f(n) \leq \big|f(n) - \lambda \cdot g(n)\big| + \lambda \cdot g(n)$$

die Ungleichung

$$f(n) \leq g(n) + \lambda \cdot g(n) = (1 + \lambda) \cdot g(n).$$

Definieren wir $c := 1 + \lambda$, so folgt für alle $n \geq k$ die Ungleichung $f(n) \leq c \cdot g(n)$. □

Wir zeigen die Nützlichkeit der obigen Sätze an Hand einiger Beispiele.

Beispiel: Es sei $k \in \mathbb{N}$. Dann gilt

$$n^k \in \mathcal{O}(n^{k+1}).$$

Beweis: Es gilt

$$\lim_{n\to\infty} \frac{n^k}{n^{k+1}} = \lim_{n\to\infty} \frac{1}{n} = 0.$$

Die Behauptung folgt nun aus dem Grenzwert-Satz. □

Beispiel: Es sei $k \in \mathbb{N}$ und $\lambda \in \mathbb{R}$ mit $\lambda > 1$. Dann gilt

$$n^k \in \mathcal{O}(\lambda^n).$$

Beweis: Wir zeigen, dass

$$\lim_{n\to\infty} \frac{n^k}{\lambda^n} = 0 \tag{3.7}$$

ist, denn dann folgt die Behauptung aus dem Grenzwert-Satz. Nach dem Satz von L'Hospital können wir den Grenzwert wie folgt berechnen

$$\lim_{n\to\infty} \frac{n^k}{\lambda^n} = \lim_{x\to\infty} \frac{x^k}{\lambda^x} = \lim_{x\to\infty} \frac{\frac{d\,x^k}{dx}}{\frac{d\,\lambda^x}{dx}}$$

Die Ableitungen können wir berechnen, es gilt:

$$\frac{d\,x^k}{dx} = k \cdot x^{k-1} \quad \text{und} \quad \frac{d\,\lambda^x}{dx} = \ln(\lambda) \cdot \lambda^x.$$

Berechnen wir die zweite Ableitung so sehen wir

$$\frac{d^2\,x^k}{dx^2} = k \cdot (k-1) \cdot x^{k-2} \quad \text{und} \quad \frac{d^2\,\lambda^x}{dx^2} = \ln(\lambda)^2 \cdot \lambda^x.$$

Für die k-te Ableitung gilt analog

$$\frac{d^k\,x^k}{dx^k} = k \cdot (k-1) \cdot \cdots \cdot 1 \cdot x^0 = k! \quad \text{und} \quad \frac{d^k\,\lambda^x}{dx^k} = \ln(\lambda)^k \cdot \lambda^x.$$

Wenden wir also den Satz von L'Hospital zur Berechnung des Grenzwertes k mal an, so sehen wir

$$\lim_{x\to\infty} \frac{x^k}{\lambda^x} = \lim_{x\to\infty} \frac{\frac{d\,x^k}{dx}}{\frac{d\,\lambda^x}{dx}} = \lim_{x\to\infty} \frac{\frac{d^2\,x^k}{dx^2}}{\frac{d^2\,\lambda^x}{dx^2}} = \cdots = \lim_{x\to\infty} \frac{\frac{d^k\,x^k}{dx^k}}{\frac{d^k\,\lambda^x}{dx^k}} = \lim_{x\to\infty} \frac{k!}{\ln(\lambda)^k \lambda^x} = 0.$$

□

Beispiel: Es gilt $\ln(n) \in \mathcal{O}(n)$.

Beweis: Wir benutzen Satz 11 und zeigen mit der Regel von L'Hospital, dass

$$\lim_{n\to\infty} \frac{\ln(n)}{n} = 0$$

ist. Es gilt

$$\frac{d\,\ln(x)}{dx} = \frac{1}{x} \quad \text{und} \quad \frac{d\,x}{dx} = 1.$$

Also haben wir

$$\lim_{n\to\infty} \frac{\ln(n)}{n} = \lim_{x\to\infty} \frac{\frac{1}{x}}{1} = \lim_{x\to\infty} \frac{1}{x} = 0.$$

□

Aufgabe 3: Zeigen Sie $\sqrt{n} \in \mathcal{O}(n)$.

Beispiel: Es gilt $2^n \in \mathcal{O}(3^n)$, aber $3^n \notin \mathcal{O}(2^n)$.

Beweis: Zunächst haben wir

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0.$$

Den Beweis, dass $3^n \notin \mathcal{O}(2^n)$ ist, führen wir indirekt und nehmen an, dass $3^n \in \mathcal{O}(2^n)$ ist. Dann muss es Konstanten c und k geben, so dass für alle $n \geq k$ gilt

$$3^n \leq c \cdot 2^n.$$

Wir logarithmieren beide Seiten dieser Ungleichung und finden

$$
\begin{array}{lll}
 & \ln(3^n) & \leq \ln(c \cdot 2^n) \\
\leftrightarrow & n \cdot \ln(3) & \leq \ln(c) + n \cdot \ln(2) \\
\leftrightarrow & n \cdot \big(\ln(3) - \ln(2)\big) & \leq \ln(c) \\
\leftrightarrow & n & \leq \dfrac{\ln(c)}{\ln(3) - \ln(2)}
\end{array}
$$

Die letzte Ungleichung müßte nun für beliebig große natürliche Zahlen n gelten und liefert damit den gesuchten Widerspruch zu unserer Annahme.

Aufgabe 4:

1. Es sei $b \geq 1$. Zeigen Sie $\log_b(n) \in \mathcal{O}(\ln(n))$.
2. $3 \cdot n^2 + 5 \cdot n + \sqrt{n} \in \mathcal{O}(n^2)$
3. $7 \cdot n + \big(\log_2(n)\big)^2 \in \mathcal{O}(n)$
4. $\sqrt{n} + \log_2(n) \in \mathcal{O}\left(\sqrt{n}\right)$
5. $n^n \in \mathcal{O}\big(2^{2^n}\big)$.

   Hinweis: Die letzte Teilaufgabe ist schwer!

3.2 Fallstudie: Effiziente Berechnung der Potenz

Wir verdeutlichen die bisher eingeführten Begriffe an einem Beispiel. Wir betrachten ein Programm zur Berechnung der Potenz m^n für natürliche Zahlen m und n. Abbildung 3.1 zeigt ein naives Programm zur Berechnung von m^n. Die diesem Programm zu Grunde liegende Idee ist es, die Berechnung von m^n nach der Formel

$$m^n = \underbrace{m \cdot \ldots \cdot m}_{n}$$

durchzuführen.

```
1  power := procedure(m, n) {
2      r := 1;
3      for (i in {1 .. n}) {
4          r := r * m;
5      }
6      return r;
7  };
```

Abbildung 3.1: Naive Berechnung von m^n für $m, n \in \mathbb{N}$.

Das Programm ist offenbar korrekt. Zur Berechnung von m^n werden für positive Exponenten n insgesamt $n - 1$ Multiplikationen durchgeführt. Wir können m^n aber wesentlich effizienter berechnen. Die Grundidee erläutern wir an der Berechnung von m^4. Es gilt

$$m^4 = (m \cdot m) \cdot (m \cdot m).$$

Wenn wir den Ausdruck $m \cdot m$ nur einmal berechnen, dann kommen wir bei der Berechnung von m^4 nach der obigen Formel mit zwei Multiplikationen aus, während bei einem naiven Vorgehen 3 Multiplikationen durchgeführt würden! Für die Berechnung von m^8 können wir folgende Formel verwenden:

$$m^8 = \big((m \cdot m) \cdot (m \cdot m)\big) \cdot \big((m \cdot m) \cdot (m \cdot m)\big).$$

Berechnen wir den Term $(m \cdot m) \cdot (m \cdot m)$ nur einmal, so werden jetzt 3 Multiplikationen benötigt um m^8 auszurechnen. Ein naives Vorgehen würde 7 Multiplikationen benötigen. Wir versuchen die oben an Beispielen erläuterte Idee in ein Programm umzusetzen. Abbildung 3.2 zeigt das Ergebnis. Es berechnet die Potenz m^n nicht durch eine naive $(n-1)$-malige Multiplikation sondern es verwendet das Paradigma

Teile und Herrsche. (engl. *divide and conquer*)

Die Grundidee um den Term m^n für $n \geq 1$ effizient zu berechnen, läßt sich durch folgende Formel beschreiben:

$$m^n = \begin{cases} m^{n/2} \cdot m^{n/2} & \text{falls } n \text{ gerade ist;} \\ m^{n/2} \cdot m^{n/2} \cdot m & \text{falls } n \text{ ungerade ist.} \end{cases}$$

```
1  power := procedure(m, n) {
2      if (n == 0) {
3          return 1;
4      }
5      p := power(m, floor(n / 2));
6      if (n % 2 == 0) {
7          return p * p;
8      } else {
9          return p * p * m;
10     }
11 };
```

Abbildung 3.2: Berechnung von m^n für $m, n \in \mathbb{N}$.

Da es keineswegs offensichtlich ist, dass das Programm in 3.2 tatsächlich die Potenz m^n berechnet, wollen wir dies nachweisen. Wir benutzen dazu die Methode der *Wertverlaufs-Induktion* (engl. *computational induction*). Die Wertverlaufs-Induktion ist eine Induktion über die Anzahl der rekursiven Aufrufe. Diese Methode bietet sich immer dann an, wenn die Korrektheit einer rekursiven Prozedur nachzuweisen ist. Das Verfahren besteht aus zwei Schritten:

1. *Induktions-Anfang.*

   Beim Induktions-Anfang weisen wir nach, dass die Prozedur in allen den Fällen korrekt arbeitet, in denen sie sich nicht selbst aufruft.

2. *Induktions-Schritt*

   Im Induktions-Schritt beweisen wir, dass die Prozedur auch in den Fällen korrekt arbeitet, in denen sie sich rekursiv aufruft. Beim Beweis dieser Tatsache dürfen wir voraussetzen, dass die Prozedur bei jedem rekursiven Aufruf den korrekten Wert produziert. Diese Voraussetzung wird auch als *Induktions-Voraussetzung* bezeichnet.

Wir demonstrieren die Methode, indem wir durch Wertverlaufs-Induktion beweisen, dass gilt:

$$\texttt{power}(m, n) \rightsquigarrow m^n.$$

1. **Induktions-Anfang**.

   Die Methode ruft sich dann nicht rekursiv auf, wenn $n = 0$ gilt. In diesem Fall haben wir

   $$\texttt{power}(m, 0) \rightsquigarrow 1 = m^0.$$

2. **Induktions-Schritt**.

   Der rekursive Aufruf der Prozedur `power` hat die Form $\texttt{power}(m, n/2)$. Also gilt nach Induktions-Voraussetzung

   $$\texttt{power}(m, n/2) \rightsquigarrow m^{n/2}.$$

Danach können in der weiteren Rechnung zwei Fälle auftreten. Wir führen daher eine Fallunterscheidung entsprechend der `if`-Abfrage in Zeile 6 durch:

(a) $n \,\%\, 2 = 0$, n ist also gerade.
Dann gibt es ein $k \in \mathbb{N}$ mit $n = 2 \cdot k$ und also ist $n/2 = k$. In diesem Fall gilt

$$\begin{array}{lcl} \texttt{power}(m,n) & \rightsquigarrow & \texttt{power}(m,k) \cdot \texttt{power}(m,k) \\ & \overset{I.V.}{\rightsquigarrow} & m^k \cdot m^k \\ & = & m^{2\cdot k} \\ & = & m^n. \end{array}$$

(b) $n \,\%\, 2 = 1$, n ist also ungerade.
Dann gibt es ein $k \in \mathbb{N}$ mit $n = 2 \cdot k + 1$ und wieder ist $n/2 = k$. In diesem Fall gilt

$$\begin{array}{lcl} \texttt{power}(m,n) & \rightsquigarrow & \texttt{power}(m,k) \cdot \texttt{power}(m,k) \cdot m \\ & \overset{I.V.}{\rightsquigarrow} & m^k \cdot m^k \cdot m \\ & = & m^{2\cdot k+1} \\ & = & m^n. \end{array}$$

Damit ist der Beweis der Korrektheit abgeschlossen. □

Als nächstes wollen wir die Komplexität des obigen Programms untersuchen. Dazu berechnen wir zunächst die Anzahl der Multiplikationen, die beim Aufruf $\texttt{power}(m,n)$ durchgeführt werden. Je nach dem, ob der Test in Zeile 6 negativ ausgeht oder nicht, gibt es mehr oder weniger Multiplikationen. Wir untersuchen zunächst den schlechtesten Fall (engl. *worst case*). Der schlechteste Fall tritt dann ein, wenn es ein $l \in \mathbb{N}$ gibt, so dass

$$n = 2^l - 1$$

ist, denn dann gilt

$$n/2 = 2^{l-1} - 1 \quad \text{und} \quad n \,\%\, 2 = 1,$$

was wir sofort durch die Probe

$$2 \cdot (n/2) + n \,\%\, 2 = 2 \cdot (2^{l-1} - 1) + 1 = 2^l - 1 = n$$

verifizieren. Folglich ist, wenn n die Form $2^l - 1$ hat, bei jedem rekursiven Aufruf der Exponent n ungerade. Wir nehmen also $n = 2^l - 1$ an und berechnen die Zahl a_n der Multiplikationen, die beim Aufruf von $\texttt{power}(m,n)$ durchgeführt werden.

Zunächst gilt $a_0 = 0$, denn wenn $n = 0$ ist, wird keine Multiplikation durchgeführt. Ansonsten haben wir in Zeile 9 zwei Multiplikationen, die zu den Multiplikationen, die beim rekursiven Aufruf in Zeile 5 anfallen, hinzu addiert werden müssen. Damit erhalten wir die folgende Rekurrenz-Gleichung:

$$a_n = a_{n/2} + 2 \qquad \text{für alle } n \in \left\{2^l - 1 \mid l \in \mathbb{N}\right\} \quad \text{mit } a_0 = 0.$$

Wir definieren $b_l := a_{2^l-1}$ und erhalten dann für die Folge $(b_l)_l$ die Rekurrenz-Gleichung

$$b_l = a_{2^l-1} = a_{(2^l-1)/2} + 2 = a_{2^{l-1}-1} + 2 = b_{l-1} + 2 \qquad \text{für alle } l \in \mathbb{N}.$$

Die Anfangs-Bedingung lautet $b_0 = a_{2^0-1} = a_0 = 0$. Offenbar lautet die Lösung der Rekurrenz-Gleichung

$$b_l = 2 \cdot l \qquad \text{für alle } l \in \mathbb{N}.$$

Diese Behauptung können Sie durch eine triviale Induktion verifizieren. Für die Folge a_n haben wir dann:

$$a_{2^l-1} = 2 \cdot l.$$

Formen wir die Gleichung $n = 2^l - 1$ nach l um, so erhalten wir $l = \log_2(n+1)$. Setzen wir diesen Wert ein, so sehen wir

$$a_n = 2 \cdot \log_2(n+1) \in \mathcal{O}\big(\log_2(n)\big).$$

Wir betrachten jetzt den günstigsten Fall, der bei der Berechnung von `power`(m, n) auftreten kann. Der günstigste Fall tritt dann ein, wenn der Test in Zeile 6 immer gelingt weil n jedesmal eine gerade Zahl ist. In diesem Fall muss es ein $l \in \mathbb{N}$ geben, so dass n die Form

$$n = 2^l$$

hat. Wir nehmen also $n = 2^l$ an und berechnen die Zahl a_n der Multiplikationen, die dann beim Aufruf von `power`(m, n) durchgeführt werden.

Zunächst gilt $a_{2^0} = a_1 = 2$, denn wenn $n = 1$ ist, scheitert der Test in Zeile 6 und Zeile 9 liefert 2 Multiplikationen. Zeile 5 liefert in diesem Fall keine Multiplikation, weil beim Aufruf `power`$(m, 0)$ sofort das Ergebnis in Zeile 4 zurück gegeben wird.

Ist $n = 2^l > 1$, so haben wir in Zeile 7 eine Multiplikation, die zu den Multiplikationen, die beim rekursiven Aufruf in Zeile 5 anfallen, hinzu addiert werden muß. Damit erhalten wir die folgende Rekurrenz-Gleichung:

$$a_n = a_{n/2} + 1 \qquad \text{für alle } n \in \big\{2^l \mid l \in \mathbb{N}\big\} \quad \text{mit } a_1 = 2.$$

Wir definieren $b_l := a_{2^l}$ und erhalten dann für die Folge $(b_l)_l$ die Rekurrenz-Gleichung

$$b_l = a_{2^l} = a_{(2^l)/2} + 1 = a_{2^{l-1}} + 1 = b_{l-1} + 1 \qquad \text{für alle } l \in \mathbb{N},$$

mit der Anfangs-Bedingungen $b_0 = a_{2^0} = a_1 = 2$. Also lösen wir die Rekurrenz-Gleichung

$$b_{l+1} = b_l + 1 \qquad \text{für alle } l \in \mathbb{N} \quad \text{mit } b_0 = 2.$$

Offenbar lautet die Lösung

$$b_l = 2 + l \qquad \text{für alle } l \in \mathbb{N}.$$

Setzen wir hier $b_l = a_{2^l}$, so erhalten wir:

$$a_{2^l} = 2 + l.$$

Formen wir die Gleichung $n = 2^l$ nach l um, so erhalten wir $l = \log_2(n)$. Setzen wir diesen Wert ein, so sehen wir

$$a_n = 2 + \log_2(n) \in \mathcal{O}\big(\log_2(n)\big).$$

Da wir sowohl im besten als auch im schlechtesten Fall das selbe Ergebnis bekommen haben, können wir schließen, dass für die Zahl a_n der Multiplikationen allgemein gilt:

$$a_n \in \mathcal{O}\big(\log_2(n)\big).$$

Bemerkung: Wenn wir nicht die Zahl der Multiplikationen sondern die Rechenzeit ermitteln wollen, die der obige Algorithmus benötigt, so wird die Rechnung wesentlich aufwendiger. Der Grund ist, dass wir dann berücksichtigen müssen, dass die Rechenzeit bei der Berechnung der Produkte in den Zeilen 7 und 9 von der Größe der Faktoren abhängig ist.

Aufgabe 5: Schreiben Sie eine Prozedur `prod` zur Multiplikation zweier Zahlen. Für zwei natürliche Zahlen m und n soll der Aufruf `prod`(m, n) das Produkt $m \cdot n$ mit Hilfe von Additionen berechnen. Benutzen Sie bei der Implementierung das Paradigma "Teile und Herrsche" und beweisen Sie die Korrektheit des Algorithmus mit Hilfe einer Wertverlaufs-Induktion. Schätzen Sie die Anzahl der Additionen, die beim Aufruf von `prod`(m, n) im schlechtesten Fall durchgeführt werden, mit Hilfe der $\mathcal{O}$-Notation ab.

3.3 Der Hauptsatz der Laufzeit-Funktionen

Im letzten Abschnitt haben wir zur Analyse der Rechenzeit der Funktion *power*() zunächst eine Rekurrenz-Gleichung aufgestellt, diese gelöst und anschließend das Ergebnis mit Hilfe der $\mathcal{O}$-Notation abgeschätzt. Wenn wir nur an einer Abschätzung interessiert sind, dann ist es in vielen Fällen nicht notwendig, die zu Grunde liegende Rekurrenz-Gleichung exakt zu lösen, denn der *Hauptsatz der Laufzeit-Funktionen* (Englisch: *Master Theorem*) [CLRS01] bietet eine Methode zur Gewinnung von Abschätzungen, bei der es nicht notwendig ist, die Rekurrenz-Gleichung zu lösen. Wir präsentieren eine etwas vereinfachte Form dieses Hauptsatzes.

Theorem 12 (Hauptsatz der Laufzeit-Funktionen) Es seien

1. $\alpha, \beta \in \mathbb{N}$ mit $\alpha \geq 1$ und $\beta > 1$,
2. $f : \mathbb{N} \to \mathbb{R}_+$,
3. die Funktion $g : \mathbb{N} \to \mathbb{R}_+$ genüge der Rekurrenz-Gleichung

   $$g(n) = \alpha \cdot g\left(n/\beta\right) + f(n),$$

   wobei der Ausdruck n/β die ganzzahlige Division von n durch β bezeichnet.

Dann können wir in den gleich genauer beschriebenen Situationen asymptotische Abschätzungen für die Funktion $g(n)$ angeben:

1. Falls es eine Konstante $\varepsilon > 0$ gibt, so dass

   $$f(n) \in \mathcal{O}\big(n^{\log_\beta(\alpha)-\varepsilon}\big)$$

   gilt, dann haben wir

   $$g(n) \in \mathcal{O}\left(n^{\log_\beta(\alpha)}\right).$$

2. Falls sowohl $f(n) \in \mathcal{O}\big(n^{\log_\beta(\alpha)}\big)$ als auch $n^{\log_\beta(\alpha)} \in \mathcal{O}\big(f(n)\big)$ gilt, dann folgt

   $$g(n) \in \mathcal{O}\big(\log_\beta(n) \cdot n^{\log_\beta(\alpha)}\big).$$

3. Falls es eine Konstante $\gamma < 1$ und eine Konstante $k \in \mathbb{N}$ gibt, so dass für $n \geq k$

   $$\alpha \cdot f\left(n/\beta\right) \leq \gamma \cdot f(n)$$

   gilt, dann folgt

   $$g(n) \in \mathcal{O}\big(f(n)\big).$$

□

Erläuterung: Ein vollständiger Beweis dieses Theorems geht über den Rahmen einer einführenden Vorlesung hinaus. Wir wollen aber erklären, wie die drei Fälle zustande kommen.

1. Wir betrachten zunächst den ersten Fall. In diesem Fall kommt der asymptotisch wesentliche Anteil des Wachstums der Funktion g von der Rekursion. Um diese Behauptung einzusehen, betrachten wir die homogene Rekurrenz-Gleichung

   $$g(n) = \alpha \cdot g\left(n/\beta\right).$$

   Wir beschränken uns auf solche Werte von n, die sich als Potenzen von β schreiben lassen, also Werte der Form

   $$n = \beta^k \quad \text{mit } k \in \mathbb{N}.$$

   Definieren wir für $k \in \mathbb{N}$ die Folge $\big(b_k\big)_{k\in\mathbb{N}}$ durch

   $$b_k := g\big(\beta^k\big),$$

so erhalten wir für die Folgenglieder b_k die Rekurrenz-Gleichung

$$b_k = g\big(\beta^k\big) = \alpha \cdot g\left(\beta^k/\beta\right) = \alpha \cdot g\big(\beta^{k-1}\big) = \alpha \cdot b_{k-1}.$$

Wir sehen unmittelbar, dass diese Rekurrenz-Gleichung die Lösung

$$b_k = \alpha^k \cdot b_0 \tag{3.8}$$

hat. Aus $n = \beta^k$ folgt sofort

$$k = \log_\beta(n).$$

Berücksichtigen wir, dass $b_k = g(n)$ ist, so liefert Gleichung (3.8) also

$$g(n) = \alpha^{\log_\beta(n)} \cdot b_0. \tag{3.9}$$

Wir zeigen, dass

$$\alpha^{\log_\beta(n)} = n^{\log_\beta(\alpha)} \tag{3.10}$$

gilt. Dazu betrachten wir die folgende Kette von Äquivalenz-Umformungen:

$$\begin{array}{lrcll}
 & \alpha^{\log_\beta(n)} & = & n^{\log_\beta(\alpha)} & \mid\ \log_\beta(\cdot) \\
\Leftrightarrow & \log_\beta\left(\alpha^{\log_\beta(n)}\right) & = & \log_\beta\left(n^{\log_\beta(\alpha)}\right) & \\
\Leftrightarrow & \log_\beta(n) \cdot log_\beta(\alpha) & = & \log_\beta(\alpha) \cdot \log_\beta(n) & \text{wegen } \log_b(x^y) = y \cdot \log_b(x)
\end{array}$$

Da die letzte Gleichung offenbar richtig ist, und wir zwischendurch nur Äquivalenz-Umformungen durchgeführt haben, ist auch die erste Gleichung richtig und wir haben Gleichung (3.10) gezeigt. Insgesamt haben wir damit

$$g(n) = n^{\log_\beta(\alpha)} \cdot b_0$$

gezeigt. Also gilt: Vernachlässigen wir die Inhomogenität f, so erhalten wir die folgende asymptotische Abschätzung:

$$g(n) \in \mathcal{O}\big(n^{\log_\beta(\alpha)}\big).$$

2. Im zweiten Fall liefert die Inhomogenität f einen Beitrag, der genau so groß ist wie die Lösung der homogenen Rekurrenz-Gleichung. Dies führt dazu, dass die Lösung asymptotisch um einen Faktor $\log_\beta(n)$ größer wird. Um das zu verstehen, betrachten wir exemplarisch die Rekurrenz-Gleichung

   $$g(n) = \alpha \cdot g\left(n/\beta\right) + n^{\log_\beta(\alpha)}$$

   mit der Anfangs-Bedingung $g(1) = 0$. Wir betrachten wieder nur Werte $n \in \{\beta^k \mid k \in \mathbb{N}\}$ und setzen daher

   $$n = \beta^k.$$

   Wie eben definieren wir

   $$b_k := g(n) = g\big(\beta^k\big).$$

Das liefert

$$b_k = \alpha \cdot g\left(\beta^k/\beta\right) + \left(\beta^k\right)^{\log_\beta(\alpha)} = \alpha \cdot g(\beta^{k-1}) + \left(\beta^{\log_\beta(\alpha)}\right)^k = \alpha \cdot b_{k-1} + \alpha^k.$$

Nun gilt $b_0 = g(1) = 0$. Um die Rekurrenz-Gleichung $b_k = \alpha \cdot b_{k-1} + \alpha^k$ zu lösen, berechnen wir zunächst die Werte für $k = 1, 2, 3$:

$$\begin{aligned} b_1 &= \alpha \cdot b_0 + \alpha^1 \\ &= \alpha \cdot 0 + \alpha \\ &= 1 \cdot \alpha^1 \end{aligned}$$

$$\begin{aligned} b_2 &= \alpha \cdot b_1 + \alpha^1 \\ &= \alpha \cdot 1 \cdot \alpha^1 + \alpha^2 \\ &= 2 \cdot \alpha^2 \end{aligned}$$

$$\begin{aligned} b_3 &= \alpha \cdot b_2 + \alpha^2 \\ &= \alpha \cdot 2 \cdot \alpha^2 + \alpha^3 \\ &= 3 \cdot \alpha^3 \end{aligned}$$

Wir vermuten hier, dass die Lösung dieser Rekurrenz-Gleichung durch die Formel

$$b_k = k \cdot \alpha^k$$

gegeben wird. Den Nachweis dieser Vermutung führen wir durch eine triviale Induktion:

I.A.: $k = 0$

Einerseits gilt $b_0 = 0$, andererseits gilt $0 \cdot \alpha^0 = 0$.

I.S.: $k \mapsto k + 1$

$$\begin{aligned} b_{k+1} &= \alpha \cdot b_k + \alpha^{k+1} \\ &\overset{IV}{=} \alpha \cdot k \cdot \alpha^k + \alpha^{k+1} \\ &= k \cdot \alpha^{k+1} + \alpha^{k+1} \\ &= (k+1) \cdot \alpha^{k+1}. \end{aligned}$$

Da aus $n = \beta^k$ sofort $k = \log_\beta(n)$ folgt, ergibt sich für die Funktion $g(n)$

$$g(n) = b_k = k \cdot \alpha^k = \log_\beta(n) \cdot \alpha^{\log_\beta(n)} = \log_\beta(n) \cdot n^{\log_\beta(\alpha)}$$

und das ist genau die Form, durch die im zweiten Fall des Hauptsatzes die Funktion $g(n)$ abgeschätzt wird.

3. Im letzten Fall des Hauptsatzes überwiegt schließlich der Beitrag der Inhomogenität, so dass die Lösung nun asymptotisch durch die Inhomogenität dominiert wird. Wir machen wieder den Ansatz

$$n = \beta^k \quad \text{und} \quad b_k = g\left(\beta^k\right).$$

Wir überlegen uns, wie die Ungleichung

$$\alpha \cdot f\left(n/\beta\right) \leq \gamma \cdot f(n)$$

für $n = \beta^k$ aussieht und erhalten

$$\alpha \cdot f\left(\beta^{k-1}\right) \leq \gamma \cdot f\left(\beta^k\right) \tag{3.11}$$

Setzen wir hier für k den Wert $k-1$ ein, so erhalten wir

$$\alpha \cdot f\left(\beta^{k-2}\right) \leq \gamma \cdot f\left(\beta^{k-1}\right) \tag{3.12}$$

Wir multiplizieren nun die Ungleichung (3.12) mit α und Ungleichung (3.11) mit γ und erhalten die Ungleichungen

$$\alpha^2 \cdot f\left(\beta^{k-2}\right) \leq \alpha \cdot \gamma \cdot f\left(\beta^{k-1}\right) \quad \text{und} \quad \alpha \cdot \gamma \cdot f\left(\beta^{k-1}\right) \leq \gamma^2 \cdot f\left(\beta^{k}\right)$$

Setzen wir diese Ungleichungen zusammen, so erhalten wir die neue Ungleichung

$$\alpha^2 \cdot f\left(\beta^{k-2}\right) \leq \gamma^2 \cdot f\left(\beta^{k}\right)$$

Iterieren wir diesen Prozess, so sehen wir, dass

$$\alpha^i \cdot f\left(\beta^{k-i}\right) \leq \gamma^i \cdot f(\beta^k) \quad \text{für alle } i \in \{1, \cdots k\} \text{ gilt.} \tag{3.13}$$

Wir berechnen nun $g(\beta^k)$ durch Iteration der Rekurrenz-Gleichung:

$$\begin{aligned}
g(\beta^k) &= \alpha \cdot g(\beta^{k-1}) + f(\beta^k) \\
&= \alpha \cdot \left(\alpha \cdot g(\beta^{k-2}) + f(\beta^{k-1})\right) + f(\beta^k) \\
&= \alpha^2 \cdot g(\beta^{k-2}) + \alpha \cdot f(\beta^{k-1}) + f(\beta^k) \\
&= \alpha^3 \cdot g(\beta^{k-3}) + \alpha^2 \cdot f(\beta^{k-2}) + \alpha \cdot f(\beta^{k-1}) + f(\beta^k) \\
&\;\;\vdots \\
&= \alpha^k \cdot g(\beta^0) + \alpha^{k-1} \cdot f(\beta^1) + \cdots + \alpha^1 \cdot f(\beta^{k-1}) + \alpha^0 \cdot f(\beta^k) \\
&= \alpha^k \cdot g(\beta^0) + \sum_{i=1}^{k} \alpha^{k-i} \cdot f(\beta^i)
\end{aligned}$$

Da bei der $\mathcal{O}$-Notation die Werte von f für kleine Argumente keine Rolle spielen, können wir ohne Beschränkung der Allgemeinheit annehmen, dass $g(\beta^0) \leq f(\beta^0)$ ist. Damit erhalten wir dann die Abschätzung

$$\begin{aligned}
g(\beta^k) &\leq \alpha^k \cdot f(\beta^0) + \sum_{i=1}^{k} \alpha^{k-i} \cdot f(\beta^i) \\
&= \sum_{i=0}^{k} \alpha^{k-i} \cdot f(\beta^i) \\
&= \sum_{j=0}^{k} \alpha^{j} \cdot f(\beta^{k-j})
\end{aligned}$$

wobei wir im letzten Schritt den Index i durch $k-j$ ersetzt haben. Berücksichtigen wir nun die Ungleichung (3.13), so erhalten wir die Ungleichungen

$$\begin{aligned}
g(\beta^k) &\leq \sum_{j=0}^{k} \gamma^j \cdot f(\beta^k) \\
&= f(\beta^k) \cdot \sum_{j=0}^{k} \gamma^j \\
&\leq f(\beta^k) \cdot \sum_{j=0}^{\infty} \gamma^j \\
&= f(\beta^k) \cdot \frac{1}{1-\gamma},
\end{aligned}$$

wobei wir im letzten Schritt die Formel für die geometrische Reihe

$$\sum_{j=0}^{\infty} q^j = \frac{1}{1-q}$$

benutzt haben. Ersetzen wir nun β^k wieder durch n, so sehen wir, dass

$$g(n) \leq \frac{1}{1-\gamma} \cdot f(n)$$

gilt und daraus folgt sofort

$$g(n) \in \mathcal{O}\big(f(n)\big).$$ □

Beispiel: Wir untersuchen das asymptotische Wachstum der Folge, die durch die Rekurrenz-Gleichung

$$a_n = 9 \cdot a_{n/3} + n$$

definiert ist. Wir haben hier

$$g(n) = 9 \cdot g(n/3) + n, \quad \text{also} \quad \alpha = 9, \quad \beta = 3, \quad f(n) = n.$$

Damit gilt

$$\log_\beta(\alpha) = \log_3(9) = 2.$$

Wir setzen $\varepsilon := 1 > 0$. Dann gilt

$$f(n) = n \in \mathcal{O}(n) = \mathcal{O}\left(n^{2-1}\right) = \mathcal{O}\left(n^{2-\varepsilon}\right).$$

Damit liegt der erste Fall des Hauptsatzes vor und wir können schließen, dass

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

gilt. □

Beispiel: Wir betrachten die Rekurrenz-Gleichung

$$a_n = a_{n/2} + 2$$

und analysieren das asymptotische Wachstum der Funktion $n \mapsto a_n$ mit Hilfe des Hauptsatzes der Laufzeit-Funktionen. Wir setzen $g(n) := a_n$ und haben also für die Funktion g die Rekurrenz-Gleichung

$$g(n) = 1 \cdot g\left(n/2\right) + 2$$

Wir definieren $\alpha := 1$, $\beta := 2$ und $f(n) = 2$. Wegen

$$\log_\beta(\alpha) = \log_2(1) = 0 \quad \text{und} \quad 2 \in \mathcal{O}(1) = \mathcal{O}(n^0) \quad \text{sowie} \quad n^0 \in \mathcal{O}(2)$$

sind die Voraussetzungen des zweiten Falls erfüllt und wir erhalten

$$a_n \in \mathcal{O}\big(\log_2(n)\big).$$ □

Beispiel: Diesmal betrachten wir die Rekurrenz-Gleichung

$$a_n = 3 \cdot a_{n/4} + n \cdot \log_2(n).$$

Es gilt $\alpha = 3$, $\beta = 4$ und $f(n) = n \cdot \log_2(n)$. Damit gilt

$$log_\beta(\alpha) = \log_4(3) < 1.$$

Damit ist klar, dass die Funktion $f(n) = n \cdot log_2(n)$ schneller wächst als die Funktion $n^{\log_4(3)}$. Damit kann höchstens der dritte Fall des Hauptsatzes vorliegen. Wir suchen also ein $\gamma < 1$, so dass die Ungleichung

$$\alpha \cdot f(n/\beta) \leq \gamma \cdot f(n)$$

gilt. Setzen wir hier die Funktion $f(n) = n \cdot \log_2(n)$ und die Werte für α und β ein, so erhalten wir die Ungleichung

$$3 \cdot n/4 \cdot \log_2(n/4) \leq \gamma \cdot n \cdot \log_2(n),$$

die für durch 4 teilbares n offenbar äquivalent ist zu

$$\frac{3}{4} \cdot \log_2(n/4) \leq \gamma \cdot \log_2(n).$$

Setzen wir $\gamma := \frac{3}{4}$ und kürzen, so geht diese Ungleichung über in die offensichtlich wahre Ungleichung

$$\log_2(n/4) \leq \log_2(n).$$

Damit liegt also der dritte Fall des Hauptsatzes vor und wir können schließen, dass

$$a_n \in \mathcal{O}\left(n \cdot \log_2(n)\right)$$

gilt. □

Aufgabe 6: Benutzen Sie den Hauptsatz der Laufzeit-Funktionen um das asymptotische Wachstum der Folgen $\left(a_n\right)_{n\in\mathbb{N}}$, $\left(b_n\right)_{n\in\mathbb{N}}$ und $\left(c_n\right)_{n\in\mathbb{N}}$ abzuschätzen, falls diese Folgen den nachstehenden Rekurrenz-Gleichungen genügen:

1. $a_n = 4 \cdot a_{n/2} + 2 \cdot n + 3$.
2. $b_n = 4 \cdot b_{n/2} + n^2$.
3. $c_n = 3 \cdot c_{n/2} + n^3$.

Bemerkung: Es ist wichtig zu sehen, dass die drei Fälle des Theorems nicht vollständig sind: Es gibt Situationen, in denen der Hauptsatz nicht anwendbar ist. Beispielsweise läßt sich der Hauptsatz nicht für die Funktion g, die durch die Rekurrenz-Gleichung

$$g(n) = 2 \cdot g(n/2) + n \cdot \log_2(n) \quad \text{mit der Anfangs-Bedingung } g(1) = 0$$

definiert ist, anwenden, denn die Inhomogenität wächst schneller als im zweiten Fall, aber nicht so schnell, dass der dritte Fall vorliegen würde. Dies können wir wie folgt sehen. Es gilt

$$\alpha = 2, \quad \beta = 2 \quad \text{und damit} \quad \log_\beta(\alpha) = 1.$$

Damit der zweite Fall vorliegt, müßte

$$n \cdot \log_2(n) \in \mathcal{O}(n^1)$$

gelten, was sicher falsch ist. Da die Inhomogenität $n \cdot \log_2(n)$ offenbar schneller wächst als der Term n^1, kann jetzt höchstens noch der dritte Fall vorliegen. Um diese Vermutung zu überprüfen, nehmen wir an, dass ein $\gamma < 1$ existiert, so dass die Inhomogenität

$$f(n) := n \cdot \log_2(n)$$

die Ungleichung

$$\alpha \cdot f(n/\beta) \leq \gamma \cdot f(n)$$

erfüllt. Einsetzen von f sowie von α und β führt auf die Ungleichung

$$2 \cdot n/2 \cdot \log_2(n/2) \leq \gamma \cdot n \cdot \log_2(n).$$

Dividieren wir diese Ungleichung durch n und vereinfachen, so erhalten wir

$$\log_2(n) - \log_2(2) \leq \gamma \cdot \log_2(n).$$

Wegen $\log_2(2) = 1$ addieren wir auf beiden Seiten 1 und subtrahieren $\gamma \cdot \log_2(n)$. Dann erhalten wir

$$\log_2(n) \cdot (1 - \gamma) \leq 1,$$

woraus schließlich

$$\log_2(n) \leq \frac{1}{1-\gamma}$$

folgt. Daraus folgt durch Anwenden der Funktion $x \mapsto 2^x$ die Ungleichung

$$n \leq 2^{\frac{1}{1-\gamma}},$$

die aber sicher nicht für beliebige n gelten kann. Damit haben wir einen Widerspruch zu der Annahme, dass der dritte Fall des Hauptsatzes vorliegt. □

Aufgabe 7: Lösen Sie die Rekurrenz-Gleichung

$$g(n) = 2 \cdot g(n/2) + n \cdot \log_2(n) \quad \text{mit der Anfangs-Bedingung } g(1) = 0$$

für den Fall, dass n eine Zweier-Potenz ist. □

Literaturverzeichnis

[AHU87] Alfred V. Aho, John E. Hopcraft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[AVL62] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[BM93] Jon Louis Bentley and M. Douglas McIlroy. Engineering a sort function. *Software - Practice and Experience*, 23(11):1249–1265, 1993.

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines*. Springer-Verlag, 2003.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[CP03] Frank M. Carrano and Janet J. Prichard. *Data Abstraction and Problem Solving with Java*. Prentice Hall, 2003.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[dlB59] René de la Briandais. File Searching Using Variable Length Keys. In *Proceedings of the Western Joint Computer Conference*, pages 195–298, 1959.

[Gur91] Yuri Gurevich. Evolving algebras. *Bull. EATCS*, 43:264–284, 1991.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4:321, 1961.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Ier06] Roberto Ierusalimschy. *Programming in Lua*. Lua.Org, 2nd edition, 2006.

[Moo59] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 83, 1953.

[Sed02] Robert Sedgewick. *Algorithms in Java*. Pearson, 2002.

[Tur36] Alan M. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[WS92] L. Wall and R. L. Schwartz. *Programming Perl.* O'Reilly and Assoc., 1992.