



Frohen Mutes macht sich Algatha an die neue Aufgabe. Und da Software-Entwicklung für sie bisher in erster Linie ein Handwerk war, spricht eigentlich nichts dagegen, die große Aufgabe einfach langsam Schritt für Schritt anzugehen und die ersten paar Zeilen Quelltext immer mehr zu erweitern. Ob eine Aufgabe gelöst wird, hängt am persönlichen Engagement und Durchhaltevermögen!

In bester »Geht nicht? Gibt's nicht!«-Stimmung fällt Algathas Blick auf dem Nachhauseweg auf eine mit Graffiti verzierte Mauer. Ein provokantes »Dieser Satz ist falsch.« bringt einen kleinen Stein in's Rollen. Die schnelle bejahende Antwort wird sofort im nächsten Augenblick negiert – doch auch das ist nicht richtig. Welche Aussage macht dieser Satz also eigentlich? Die bloße Erkenntnis, dass es noch etwas anderes als wahr und falsch gibt, verwandelt den kleinen Stein in einen ausgewachsenen Erdrutsch bezüglich Algathas Zuversicht. Wenn schon so ein mickriger Satz nicht vernünftig aufgelöst werden kann, dann ist es vielleicht in noch größerem Ausmaß möglich, dass die im vorigen Kapitel identifizierten Teilprobleme gar nicht gelöst werden können!

Von leichter Panik ergriffen fällt Algathas Blick auf eine andere Stelle der Wand, an der ein »Time is on my side« für bessere Laune wirbt. Eine noch größere Unruhe ergreift Algatha: Selbst wenn die Probleme lösbar sind, ist die Zeit leider nicht auf ihrer Seite. Die in der Zukunft anfallende Datenmenge wirft die Frage auf, ob diese Probleme überhaupt in akzeptabler Zeit gelöst werden können.

Nach diesem kurzen Dämpfer gewinnt Algatha schnell ihre Selbstbeherrschung zurück und nimmt sich für den nächsten Tag vor, die grundsätzliche Lösbarkeit von Problemen und Fragen der Laufzeiteffizienz genauer anzuschauen.

2 Machbarkeit und Effizienz

*Nikola Tesla: Nothing is impossible, Mr. Angier.
What you want is simply expensive.
(The Prestige, 2006)*

Aufbau des Kapitels	
2.1	Der Algorithmusbegriff 17
2.2	Grenzen der Algorithmik 21
2.3	Laufzeitüberlegungen 23
2.4	Schwierigkeit von Problemen 32
2.5	Zutaten für effiziente Algorithmen 36
	Übungsaufgaben 38

2.1 Der Algorithmusbegriff

Der Algorithmusbegriff ist ein zentrales Thema dieses Buches. Aus diesem Grund sind die folgenden klaren Definitionen und Notationen notwendig.

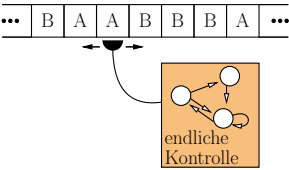
Definition 2.1: Algorithmus

Ein **Algorithmus** ist eine Vorschrift zur Bewältigung einer Aufgabe als Folge von Aktionen, wobei die folgenden Bedingungen gelten:

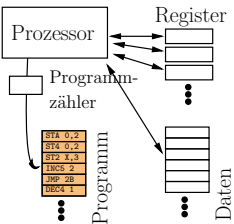
►Algorithmus

- Der Algorithmus lässt sich in endlicher Form beschreiben und ist aus Elementen von endlich vielen, durch die ausführende Maschine definierten Kommandobefehlen aufgebaut.
- Der Algorithmus hat genau ein Startaktion und nach der Ausführung eines jeden Befehls ist die Menge der Folgeaktionen klar durch die Beschreibung und die bisher durchgeführten Aktionen definiert.
- Die Eingabe des Algorithmus ist eine Folge von Daten, die auch leer oder unendlich sein kann. Aber zu jedem Zeitpunkt während der Ausführung ist die bisher betrachtete Datenmenge endlich.

Diese Definition ist noch sehr allgemein und wird erst konkret, wenn ein Maschinenmodell betrachtet wird, auf dem der Algorithmus ausgeführt wird. Das Maschinenmodell definiert die zur Verfügung stehenden Operationen und Kontrollkonstrukte. Die üblichen Maschinenmodelle in diesem Zusammenhang sind:



Schematisches Beispiel einer Turing-Maschine.



Schematische Darstellung einer Registermaschine.

- Turing-Maschine: Als Speicher stehen endlich viele Zustände eines endlichen Automaten sowie ein potentiell unendlich langer Speicherband zur Verfügung. Als Operationen kann der Lesekopf bewegt sowie der Inhalt des Speicherbands gelesen und beschrieben werden. Die Kontrolle ist in einem endlichen Automaten abgelegt – dieser stellt quasi den implementierten Algorithmus dar. Grundsätzlich ist dieses Modell genauso mächtig wie gängige Computer. Als theoretisches Maschinenmodell können wir es sogar mit potentiell beliebig großer Parallelität ausstatten – d.h. nach einer Operation können viele Folgeoperationen gleichzeitig durchgeführt werden. Wir werden ganz kurz auf die Bedeutung der Turing-Maschine für den Algorithmusbegriff noch am Ende von Abschnitt 2.3 eingehen.
- Konkretes Hardware-nahes Maschinenmodell (Registermaschine): In einer Assemblersprache werden die konkreten Anweisungen für den Prozessor beschrieben. Ähnlich wie bei der Turing-Maschine sind die technischen Beschränkungen wie Anzahl der verfügbaren Register offensichtlicher Teil des Modells und jedes algorithmische Detail muss darauf abgebildet werden. Dadurch ist einerseits die Beschreibung des Algorithmus mit allen weiterführenden Betrachtungen in jedem Fall exakt – es kann kein aufwendiger Teil eines Algorithmus in hochsprachlichen Anweisungen »versteckt« werden –, andererseits ist die oft leicht kryptische Schreibweise schlecht lesbar und der fehlende Zwang zur strukturierten Programmierung deckt sich nicht mit heute aktuellen Programmiersprachen.
- Virtuelle Maschine einer Hochsprache (z.B. Java): Eine konkrete Programmiersprache mit all ihren Konstrukten zur Ablaufkontrolle und ihren Basisbefehlen wird benutzt. Speicher wird in der üblichen Form als Variable deklariert und man macht sich an dieser Stelle kaum Gedanken darüber, wie viel Speicher letztendlich die CPU zur Verfügung stellt, da die virtuelle Maschine diese Details verbirgt. Weiterführende Überlegungen der Effizienz und des Ressourcenbedarfs müssen allerdings teilweise sehr sorgfältig durchgeführt werden, was ein Nachteil der hochsprachlichen virtuellen Maschine ist.

Ein Vergleich der Maschinenmodelle mit den Anforderungen der Algorithmusdefinition zeigt, dass alle drei Maschinen als Grundlage für die Beschreibung eines Algorithmus benutzt werden können (vgl. Tabelle 2.1.)

Letztlich muss man sich bei der Beschreibung von Algorithmen für eines der Maschinenmodelle entscheiden. Im vorliegenden Buch ist dies die hochsprachliche virtuelle Maschine, die einerseits die Vorteile der strukturierten Programmierung mit sich bringt und andererseits sehr nah an den aktuellen Programmiersprachen ist. Dies zwingt uns allerdings nicht, alle Algorithmen tatsächlich in einer Programmiersprache wie Java zu präsentieren. Vielmehr halten wir eine etwas abstraktere Beschreibung in Java-nahem Pseudo-Code für besser geeignet für das grundlegende Verständnis

Tabelle 2.1: Vergleich der Maschinenmodelle mit der Algorithmusdefinition

Definition	Turing-Maschine	Registermaschine MIX	Java-VM
endliche Programmgröße	Anzahl der Zustände im endlichen Automaten	Assemblerprogramm bestimmter Größe	Java-Programm bestimmter Größe
endliche Kommandozahl	Lesen, Schreiben, Lesekopf bewegen	LDA, LDX, STA, ADD, SUB, MUL, DIV, ...	Java-Grundbefehle
eine Startoperation	Anfangszustand	Beginn des Programms	Beginn des Programms
Folgeoperationen definiert	durch Übergangsrelation bestimmt, bei Nichtdeterminismus sogar mehrere parallel	sequentieller Ablauf und Sprünge	sequentieller Ablauf und Strukturkommandos while , if-then-else , ...
Eingabe: Datenfolge	auf dem Band	Folge von Interaktionen mit Speicher/peripheren Geräten	Folge von Interaktionen mit Speicher/peripheren Geräten

von Abläufen, da wir uns auf das Wesentliche konzentrieren und Details der Java-Syntax ausblenden können. Die Notation des Pseudo-Codes ist kompakt im Anhang A dargestellt. Die ausprogrammierte Umsetzung als Java-Quelltext kann dem Online-Begleitmaterial entnommen werden.



Während die Turing-Maschine eher für theoretische Überlegungen herangezogen wird (Hopcroft et al., 2003), sind die anderen beiden Maschinenmodelle durchaus gebräuchlich in Lehrbüchern. Knuth (1997, 1998a,b) verfolgt beispielsweise mit dem Prozessormodell MIX einen Assembler-nahen Ansatz, während Sedgewick (2003) oder Weiss (2007) ihre Algorithmen in Java präsentieren.

Beispiel 2.2:

Algorithmus 2.1 zeigt ein Beispiel für den benutzten Pseudo-Code. Links sind die Einzelschritte detailliert beschrieben, während rechts die logischen Schritte und Phasen in Orange auf einer höheren Abstraktionsebene erläutert werden. Für eine Beschreibung der Notation siehe: Anhang A. □

Definition 2.3: Determiniert und deterministisch

Ein Algorithmus heißt genau dann **determiniert**, wenn er für dieselbe Eingabe immer dasselbe Ergebnis liefert. Ein Algorithmus heißt ferner **deterministisch**, wenn sein innerer Ablauf immer identisch ist, es also zu jedem Zeitpunkt immer nur eine Folgeoperation gibt. Gilt dies nicht, so kann der Algorithmus

►determiniert

►deterministisch

- **randomisiert** sein, falls (Pseudo-)Zufallszahlen benutzt werden und diese den Ablauf unabhängig von den Eingaben beeinflussen,

►randomisiert

www

EinfTextsuche.java

Algorithmus 2.1 Ineffizienter Algorithmus zur Textsuche als Beispiel für die Pseudo-Code-Notation

```
EINFACHE-TEXTSUCHE(Gesamttext T[], Suchtext S[])
  Rückgabewert: Startpositionen index
1  anzahl ← 0
2  for startpos ← 0, ..., T.länge - S.länge
3  do ⌈ i ← 1
4      while i ≤ S.länge und S[i] = T[startpos + i]
5      do ⌊ i ← i + 1
6      if i = S.länge + 1
7      then anzahl ← anzahl + 1
8      ⌊ ⌊ index[anzahl] ← startpos
9  return index
```

► nicht-deterministisch

► asynchron

- nicht-deterministisch sein, falls durch eine Anweisung »rate richtig« parallel alle Möglichkeiten durchprobiert werden, oder
- asynchron sein, falls in parallelen Berechnungen unterschiedliche Bearbeitungs- oder Kommunikationszeiten den Ablauf und das Ergebnis beeinflussen.

Im Rahmen dieses Buches ist Nicht-Determinismus nur von marginalem Interesse und Asynchronität kommt überhaupt nicht vor. Die anderen Begriffe werden uns noch genauer beschäftigen.

Definition 2.4: Terminiert

► terminiert

► terminiert stets

Ein Algorithmus terminiert auf eine Eingabe *x* genau dann, wenn er nach endlich vielen Schritten anhält. Er terminiert stets, wenn er auf alle möglichen Eingaben terminiert.

Definition 2.5: Korrektheit

► partiell korrekt

► total korrekt

Ein Algorithmus heißt partiell korrekt, wenn in allen Fällen, in denen der Algorithmus terminiert, seine Ausgabe der Spezifikation (in der Problembeschreibung) genügt. Falls der Algorithmus zusätzlich für alle Eingaben terminiert, heißt er total korrekt.

► klassischer Algorithmus

► randomisierter Algorithmus

► Approximationsalgorithmus

► probabilistischer Algorithmus

► Heuristik

Mit diesen Eigenschaften lassen sich die Algorithmen in verschiedene Klassen einteilen. Klassische Algorithmen sind immer korrekt, determiniert und terminieren stets. Randomisierte Algorithmen dürfen zusätzliche Zufallszahlen benutzen, sind aber trotzdem korrekt und terminieren stets. Approximationsalgorithmen zeichnen sich dadurch aus, dass sie eine garantiert gute Näherung der Lösung liefern – meist gibt es einen Beweis für den maximalen Fehler. Probabilistische Algorithmen besitzen keine Garantie für die Lösungsqualität, sondern können sich mit einer gewissen Wahrscheinlichkeit »irren«. Und schließlich zeichnet sich die Heuristik dadurch aus, dass sie genau dann gute Ergebnisse liefert, wenn die Eingaben zu einer

Tabelle 2.2: Klassifikation von Algorithmen (nach Rawlins, 1992)

	korrekt	terminiert	Zufallszahlen
klassischer Algorithmus	ja	ja	nein
randomisierter Algorithmus	ja	ja	ja
Approximationsalgorithmus	nah dran	ja	manchmal
probabilistischer Algorithmus	meist	ja	manchmal
Heuristik	nicht sicher	nicht sicher	manchmal

Grundannahme der Heuristik passen. Die verschiedenen Algorithmen sind im Vergleich in Tabelle 2.2 dargestellt.

In den weiteren Kapiteln werden Beispiele für die verschiedenen Algorithmenarten vorgestellt. Eine Quicksortvariante (Abschnitt 7.3) und die Skipliste aus Abschnitt 4.2 stehen für randomisierte Algorithmen. Als Approximationsalgorithmus lernen wir die Rundreise-Approximation in Abschnitt 5.4 kennen. Beispiel für die probabilistischen Algorithmen ist der evolutionäre Ansatz zur Lösung des Rundreiseproblems in Abschnitt 12.2. Ein Beispiel für eine Heuristik wird ebenfalls für das Rundreiseproblem (Abschnitt 5.4) vorgestellt, aber auch für ein anderes Problem in der Übungsaufgabe 5.8.



Das ist ja soweit alles ganz nett, doch was bedeutet das jetzt für meine Frage, ob unsere Probleme lösbar sind?

Definition 2.6: Lösbarkeit eines Problems

Ein Problem wird als **lösbar** bezeichnet, wenn es einen Algorithmus gibt, der für jede Instanz des Problems in endlicher Zeit eine Lösung berechnet.

► lösbares Problem



Eng verwandt mit unserem Begriff der »Lösbarkeit« ist der Begriff der »Berechenbarkeit« aus der theoretischen Informatik. Dort geht es zunächst um die Berechnung mathematischer Funktionen auf abstrakten Maschinenmodellen. Auch hier sei auf entsprechende Lehrbücher verwiesen (Schöning, 2008; Davis, 1982).

2.2 Grenzen der Algorithmik

In den folgenden Kapiteln wird eine Vielfalt an algorithmischen Techniken »aus dem Ärmel geschüttelt«, die dem Leser vermeintlich Unmögliches nahe bringt. Es könnte also leicht der Eindruck entstehen, dass grundsätzlich alles algorithmisch gelöst werden kann. Daher wollen wir uns an dieser Stelle kurz den Grenzen der Algorithmik nähern und mit dem sog. **Halteproblem** ein unlösbares Problem betrachten.

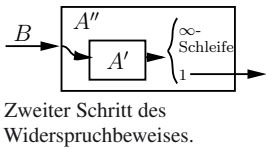
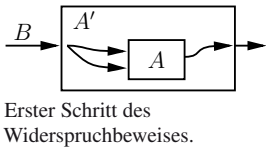
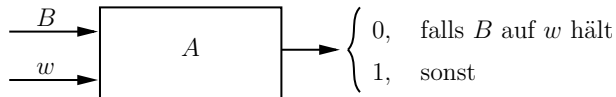
► Halteproblem

Satz 2.7:

Es existiert kein Algorithmus, der für einen beliebigen Algorithmus und eine beliebige Eingabe entscheiden kann, ob der Algorithmus auf der Eingabe terminiert.

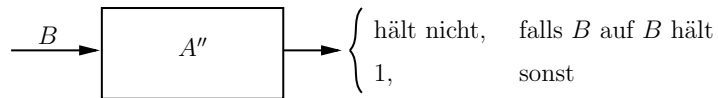
Beweis 2.8:

Wir beweisen den Satz per Widerspruch, indem wir annehmen, dass es einen solchen Algorithmus A gäbe, der für alle Algorithmen B und beliebige Eingabe w entscheidet, ob B auf w hält oder nicht.



Gehen wir ferner davon aus, dass der zu testende Algorithmus B in einer geeigneten Kodierung vorliegt – z.B. als Java-Bytecode, der dann von der virtuellen Maschine auch ausgeführt werden könnte. Zur Vereinfachung können wir also davon ausgehen, dass sowohl B als auch die Eingabe w in binärer Form vorliegen. Dann können wir einen Algorithmus A' konstruieren, der eine einzelne binäre Eingabe B erhält und diese sowohl als Algorithmus B als auch als Eingabe $w = B$ interpretiert. Das ist so machbar, da w beliebig war und somit auch für das spezielle $w = B$ gilt.

In der Variante A'' führen wir einfach A' wie oben aus und gehen genau dann in eine unendliche Schleife, wenn A' die Ausgabe **true** erzeugt. Damit haben wir den folgenden Algorithmus konstruiert:



Wenn man nun allerdings A'' auf sich selbst als Eingabe anwendet, dann geht A'' mit der Eingabe A'' genau dann in eine unendliche Schleife, wenn A'' in endlich vielen Schritten auf A'' hält. Dies ist ein logischer Widerspruch und die Annahme zu Beginn des Beweises war falsch. ■

Der Kern des Beweises ist ganz analog geführt zu den bekannten Paradoxien wie dem einführenden »Dieser Satz ist falsch.«, der in einer streng logischen Welt nicht existieren dürfte. Genau dasselbe gilt für einen Algorithmus, der das Halteproblem löst.



Der Beweis für das Halteproblem wurde zum ersten Mal von Turing (1936) für die Turing-Maschine als Maschinenmodell geführt. Tatsächlich gilt diese Unentscheidbarkeit noch für wesentlich mehr Eigenschaften von Algorithmen, wie Rice (1953) gezeigt hat. Eng damit verknüpft ist auch der Unvollständigkeitssatz von Gödel (1931), der die zwingende Unvollständigkeit bzw. Widersprüchlichkeit von hinreichend mächtigen formalen Systemen bewiesen hat.

Obiger Satz steht allerdings nicht im Konflikt mit einer Untersuchung eines fest gegebenen Algorithmus, in welcher bewiesen wird, dass dieser Algorithmus terminiert. Der Satz bedeutet lediglich, dass es kein automatisierbares Verfahren für einen solchen Beweis gibt und für bestimmte Algorithmen die Termination auch (noch) nicht bekannt ist.



So weit, so gut. Aber hat dies irgendetwas mit den fünf Basisproblemen meiner Anwendung zu tun? Wenn ich es mir recht überlege, sollten die alle lösbar sein, denn zu jedem dieser Probleme gibt es nur eine endliche Anzahl möglicher Kandidaten als Lösung. Ein Algorithmus, der diese alle der Reihe nach durchprobiert, findet sicher die gesuchte Lösung.

Neben der Tatsache, dass manche Probleme sich überhaupt nicht durch einen Algorithmus lösen lassen, birgt allein die Problemgröße eine weitere Grenze der Algorithmik: Sind wir immer in der Lage, durch Aufzählen aller Möglichkeiten, ein Problem zu lösen?

Beispiel 2.9:

Für das Rundreiseproblem (Definition 1.19) erhalten wir alle möglichen Rundreisen, wenn wir alle Permutationen erzeugen. Für n Städte sind dies $n! = 1 \cdot 2 \cdot 3 \cdots n$ Stück. Also bei $n = 101$ Städten rund $9,426 \cdot 10^{159}$ Permutationen – wenn wir Symmetrien und zyklische Verschiebungen jeder Rundreise vermeiden, bleiben es immer noch $\frac{1}{2} \cdot 100! = 4,666 \cdot 10^{157}$ Rundreisen. Unter der Annahme, wir könnten 10^6 Rundreisen pro Sekunde testen, folgt eine Laufzeit von etwa $8,878 \cdot 10^{151}$ Jahren. $\square$

Das bedeutet einerseits, dass die im Weiteren vorgestellten Algorithmen grundsätzlich sehr geschickt vorgehen müssen, aber andererseits auch, dass den Laufzeitbetrachtungen der Algorithmen eine besondere Bedeutung als beschränkender Faktor in der Algorithmik zufällt.

2.3 Laufzeitüberlegungen

Möchte man die Laufzeit eines Algorithmus angeben, gibt es drei verschiedene Ansätze mit zunehmendem Abstraktionsgrad:

- Echtzeitmessungen – diese sind nur bedingt für grundsätzliche Vergleiche und Einordnungen von Algorithmen geeignet, da sie abhängig von der genutzten Hardware unterschiedlich ausfallen können,
- exakte Anzahl der Rechenschritte/Schlüsselvergleiche/... – können sowohl für Beispiele berechnet als auch experimentell ermittelt werden, oder

- die sog. **asymptotische Laufzeitkomplexität**, die ausgehend von der exakten Anzahl lediglich eine Einordnung bzgl. einer wachsenden Problemgröße n vornimmt.

Zunächst schauen wir uns an, wie die exakte Anzahl der Schritte mithilfe der folgenden Basisregeln ermittelt werden kann:

- eine Sequenz von Anweisungen resultiert in der Summe der Laufzeiten der Anweisungen,
- bei einer Verzweigung kann man zunächst mit dem aufwändigeren Teil rechnen und
- in Schleifen wird der Aufwand der Anweisungen für einen Schleifendurchlauf mit der Anzahl der Durchläufe multipliziert.
Bei einer For-Schleife zählt zusätzlich für die Zählvariable bei jeder Iteration eine Zuweisung und das Inkrementieren des Werts sowie der Vergleich, ob das Ende der Schleife erreicht ist (d.h. allein für die Zählvariable 3 Operationen in jedem Schleifendurchlauf) – ferner wird die Zählvariable initialisiert und der Vergleich wird auch beim Abbruch der Schleife durchgeführt (d.h. 2 Operationen außerhalb der Schleife).

```

for  $i \leftarrow 1, \dots, n$ 
do [ something
=
 $i \leftarrow 1$ 
while  $i \leq n$ 
do [ something
     $i \leftarrow i + 1$ 

```

ein Schritt am Anfang
 ein Schritt am Ende
 drei Schritte pro Iteration

Beispiel 2.10:

Für den Beispiyalgorithmus 2.2 werden alle Schritte gezählt – dabei zählt jede Zuweisung, jede Rechenoperation und jeder Vergleich. Vereinfachend wird hier auch angenommen, dass jeder dieser Schritte den gleichen Aufwand besitzt, also gleich lange dauert. Damit erhalten wir 1 Operation für die Initialisierung von s , 2 Operationen für die Initialisierung und die Abbruchüberprüfung von k , in jedem Schleifendurchlauf der For-Schleife 3 Operationen für die Zählvariable (2 für das Inkrementieren von k und 1 für den Vergleich auf das Schleifenende) sowie 2 Operationen für $s + k$ bzw. die Zuweisung dieses Wertes zu s . Insgesamt ergibt sich:

$$T(n) = 1 + (2 + \underbrace{\sum_{k=1}^n (3 + 2)}_{=n \cdot 5}) = 5 \cdot n + 3$$

□

Algorithmus 2.2 Erster Algorithmus für die Laufzeitanalyse

EIN-LAUFZEITBEISPIEL(n)

Rückgabewert: –

```

1  $s \leftarrow 0$ 
2 for  $k \leftarrow 1, \dots, n$ 
3 do [  $s \leftarrow s + k$  ]

```

wird n Mal durchlaufen

Beispiel 2.11:

Im Beispiyalgorithmus 2.3 ist die Schrittzahl im Körper der äußeren Schleife nicht mehr jedes Mal gleich. Daher kann man entweder grob abschätzen, indem man mit dem Maximum für jeden Schleifendurchlauf rechnet, oder – wie folgt – die Schrittzahl genau ausrechnen:

$$\begin{aligned}
 T(n) &= 1 + \left(2 + \sum_{k=1}^n \left(3 + \underbrace{\left(2 + \sum_{m=k}^n (3+2) \right)}_{=(n-k+1) \cdot 5} \right) \right) \\
 &= 3 + \sum_{k=1}^n (5 + (n-k+1) \cdot 5) \\
 &= 3 + \underbrace{\sum_{k=1}^n 5}_{=5 \cdot n} + 5 \cdot \underbrace{\sum_{k=1}^n (n-k+1)}_{=n+(n-1)+\dots+1} \\
 &= 3 + 5 \cdot n + 5 \cdot \underbrace{\sum_{j=1}^n j}_{=\frac{n \cdot (n+1)}{2}} \\
 &= 3 + \frac{15}{2} \cdot n + \frac{5}{2} \cdot n^2
 \end{aligned}$$

□

Beispiel 2.12:

Für den Beispiyalgorithmus 2.4 lässt sich die exakte Schrittzahl nicht mehr so einfach bestimmen, da die Anzahl der Schleifendurchläufe unklar

Algorithmus 2.3 Zweiter Algorithmus für die Laufzeitanalyse

WEITERES-LAUFZEITBEISPIEL(n)**Rückgabewert:** –

```

1   $s \leftarrow 0$ 
2  for  $k \leftarrow 1, \dots, n$ 
3  do  $\lceil$  for  $m \leftarrow k, \dots, n$   $\rceil$  Durchläufe hängen  $\left. \vphantom{\sum_{k=1}^n} \right\}$  wird  $n$  Mal durchlaufen
4   $\quad \lfloor$  do  $\lfloor s \leftarrow s + m$   $\rfloor$  von  $k$  ab

```

Algorithmus 2.4 Dritter Algorithmus für die Laufzeitanalyse

LETZTES-LAUFZEITBEISPIEL(n)**Rückgabewert:** –

```

1  while  $n > 0$   $\rfloor$  1 Vergleich (pro Iteration und beim Abbruch)  $\left. \vphantom{\sum_{k=1}^n} \right\}$  höchstens
2  do  $\lceil s \leftarrow s \cdot n$   $\rfloor$  1 Operation und 1 Zuweisung  $\left. \vphantom{\sum_{k=1}^n} \right\}$   $n + 1$ 
3  if  $n$  ist gerade  $\rfloor$  1 Vergleich  $\left. \vphantom{\sum_{k=1}^n} \right\}$  Iterationen
4  then  $\lfloor n \leftarrow n - 3$   $\rfloor$  1 Operation und 1 Zuweisung
5   $\quad \lfloor$  else  $\lfloor n \leftarrow n + 1$   $\rfloor$ 

```

ist. Leicht sieht man jedoch ein, dass durch die Anweisung in Zeilen 4 und 5 n wechselweise gerade und ungerade ist. Dadurch wird n maximal auf $n + 1$ erhöht und nach jeweils zwei Iterationen wird n in der Summe um 2 verringert. Dadurch lässt sich die Anzahl der Iterationen nach oben durch $n + 1$ abschätzen. Und wie im vorherigen Beispiel lässt sich zumindest für die obere Grenze die Schrittzahl ermitteln.

$$T(n) \leq 1 + \underbrace{\sum_{k=1}^{n+1} 6}_{=6 \cdot (n+1)} = 6 \cdot n + 7$$

□



Das ist ja klasse, dass ich die Schrittzahl so direkt ausrechnen kann. Das macht Spaß! Allerdings funktioniert das wohl nur bei kleinen Algorithmen...

Für die Laufzeitbestimmung sind zwei Dimensionen eines Problems interessant: einerseits die Größe des Problems – also z. B. die Anzahl n der Schlüssel beim Sortierproblem – und andererseits die konkret möglichen Probleminstanzen für diese Größe – beispielsweise die möglichen Permutationen der Schlüsselmenge $\{1, \dots, n\}$ (liegt die Schlüsselmenge sortiert, umgekehrt sortiert oder völlig unsortiert vor?).

In der Regel hängt die Schwierigkeit für die Lösung eines Problems nicht nur von der Größe n ab. Vielmehr gibt es einfache und schwierige Probleminstanzen gleicher Größe. Um zu asymptotischen Aussagen zu kommen, müssen wir in der folgenden Definition den Laufzeitbegriff für eine Problemgröße genauer fassen.

Definition 2.13: Best-, Worst- und Average-Case

Betrachtet man für ein Problem immer diejenige Probleminstanz einer Größe, die die geringste Laufzeit hat, spricht man vom **Best-Case** (oder günstigsten Fall). Betrachtet man die längste Laufzeit, handelt es sich um den **Worst-Case** (oder ungünstigsten Fall). Untersucht man alle Probleminstanzen derselben Größe und ermittelt die durchschnittliche Laufzeit, bezeichnet man dies als **Average-Case**.

- Best-Case
- Worst-Case
- Average-Case

Beispiel 2.14:

In Bild 2.1 werden verschiedene Laufzeiten dreier Algorithmen für mehrere Probleminstanzen angezeigt. Es sind mit drei Linien jeweils der Best-Case, der Average-Case und der Worst-Case markiert. Je nachdem welchen der drei Fälle wir als Entscheidungsgrundlage heranziehen, würden wir jeweils einen anderen Algorithmus wählen. □

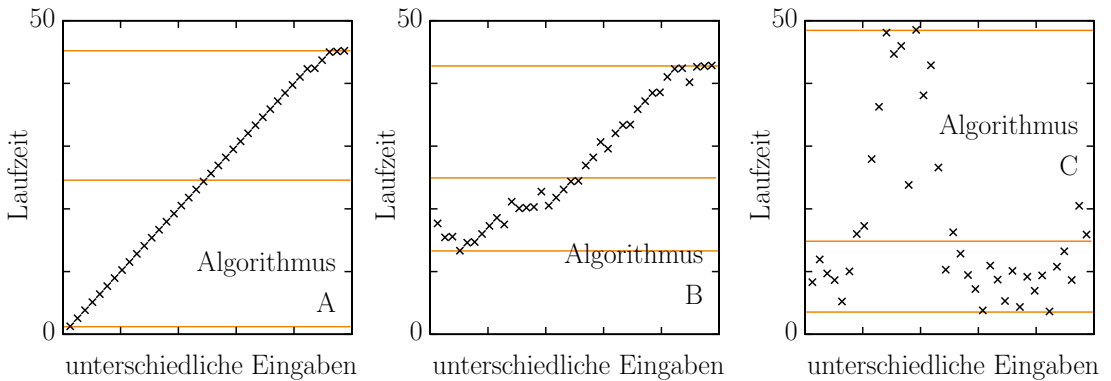


Bild 2.1: Die Laufzeit von drei Algorithmen wird für mehrere Probleminstanzen angezeigt. Die drei farbigen Linien markieren den Worst-Case (oben), Average-Case (mittig) und Best-Case (unten). Beim Vergleich der Best-Case Ergebnisse schneidet Algorithmus A am besten ab. Für den besten Average-Case wählen wir Algorithmus C. Algorithmus B wird verwendet, wenn der Worst-Case möglichst kleine Laufzeit liefern soll.



Das ist ja spannend. Je nachdem, was wir betrachten, sind jeweils andere Algorithmen gut!

Für einen konkreten Algorithmus lässt sich die Best-Case, Average-Case und Worst-Case Laufzeit als Funktion $T : \mathbb{N} \rightarrow \mathbb{R}^+$ darstellen, wobei $n \in \mathbb{N}$ die Problemgröße ist. Dabei soll im Weiteren nur die »Stärke« des Wachstums mit zunehmender Problemgröße interessieren. Typische Funktionen mit unterschiedlich starkem Wachstum sind in Bild 2.2 dargestellt. Ebenso gibt Tabelle 2.3 einen Einblick in das Wachstum der Funktionen, indem die Anzahl der benötigten Dezimalziffern angegeben wird – hier sogar für noch stärker wachsende Funktionen.

Da uns häufig lediglich die Zuordnung zu diesen Funktionen ausreicht, führen wir die sogenannten **Landau-Symbole** O (Groß-O), Ω (Omega) und Θ (Theta) ein, welche eine grobe Einordnung in Funktionsklassen bzgl. des asymptotischen Wachstums erlauben.

► Landau-Symbole

Definition 2.15: Landau-Notation

Für $f : \mathbb{N} \rightarrow \mathbb{N}$ werden die folgenden Funktionsklassen definiert.

$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{N} \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

$$\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{N} \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

$$\Theta(f) = O(f) \cap \Omega(f).$$

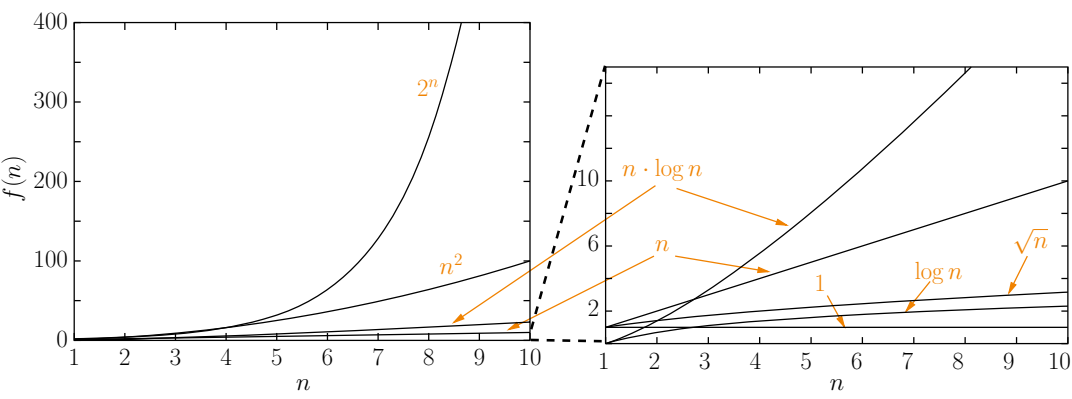
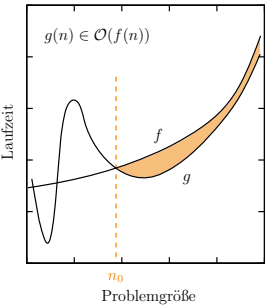


Bild 2.2: Bei der Laufzeitanalyse von Algorithmen treten häufig die hier angeführten Funktionen auf, die vergleichend dargestellt sind.



Bedeutung des n_0 in der Definition.



Also nochmal in Worte gefasst: Ein Algorithmus ist beispielsweise in $O(n^2)$, wenn sich seine Laufzeit unter bestimmten Bedingungen nach oben durch n^2 beschränken lässt. Aber da brauche ich erstmal ein paar Beispiele, um das vollständig zu verstehen...

Beispiel 2.16:

Auf dem nebenstehenden Bild sind für kleine Werte n die Funktionen $f(n)$ und $g(n)$ wechselseitig kleiner. Ab der Problemgröße n_0 allerdings bleibt

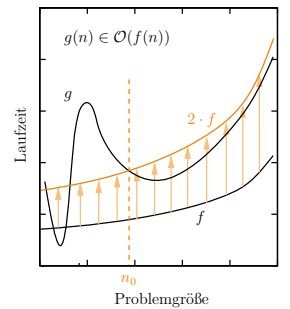
Tabelle 2.3: Anzahl der Dezimalziffern für die ersten Werte unterschiedlich stark wachsender Funktionen.

	Wert n										
	1	2	3	4	5	6	7	8	9	10 ... 1023	
$\lg n$	1	1	1	1	1	1	1	1	1	1	1
n	1	1	1	1	1	1	1	1	1	2	4
$n \cdot \lg n$	1	1	1	1	2	2	2	2	2	2	5
n^2	1	1	1	2	2	2	2	2	2	3	7
2^n	1	1	1	2	2	2	3	3	3	4	308
$n!$	1	1	1	2	3	3	4	5	6	7	2636
2^{n^2}	1	2	3	5	8	11	15	20	25	31	?
2^{2^n}	1	2	3	5	10	20	39	78	155	309	?
$2^{\cdot^{\cdot^{\cdot^2}}}_n$	1	1	2	5	19728	?	?	?	?	?	?

die Funktion $f(n)$ oberhalb der Funktion $g(n)$ – für alle Werte $n \rightarrow \infty$ (hier nur sichtbar bis $n = 50$). Damit gilt $g(n) \in O(f(n))$, da obige Bedingung für $c = 1$ erfüllt ist. Man sagt: $g(n)$ wächst höchstens so stark wie $f(n)$. $\square$

Beispiel 2.17:

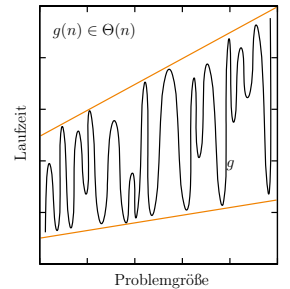
Im zweiten nebenstehenden Bild ist die Funktion $g(n)$ im sichtbaren Bereich im Wesentlichen deutlich langsamer, hat also eine höhere Laufzeit, als die Funktion $f(n)$. Da aber nur das asymptotische Wachstum ohne konstante Faktoren interessant ist, kann durch einen geeigneten Wert $c = 2$ die Funktion $f(n)$ in die Situation des ersten Bilds übertragen werden, womit wieder $g(n) \in O(f(n))$ gezeigt werden kann. $\square$



Bedeutung des c in der Definition.

Beispiel 2.18:

Im dritten Bild ist eine Funktion zu sehen, deren Minimal- und Maximalwerte sich durch die angezeigten linearen Funktionen beschränken lassen. Da die Steigung der linearen Funktionen als konstanter Faktor irrelevant ist und auch die Verschiebung der Funktionen am Nullpunkt das asymptotische Wachstum nicht beeinflusst, gilt $g(n) \in O(n)$ durch die obere Grenze und $g(n) \in \Omega(n)$ durch die untere Grenze. Daraus folgt direkt laut Definition $g(n) \in \Theta(n)$. $\square$



Beispiel für Θ .



Das Symbol O wurde vom Mathematiker Bachmann (1894) eingeführt. Die Bezeichnung »Landau-Notation« geht auf das Buch von Landau (1909) zurück, welches der Verbreitung der Notation Vor-schub geleistet hat. In der Informatik wurde die Notation durch Knuth (1976) um die anderen Symbole erweitert und durchgesetzt.

Bei der asymptotischen Laufzeitanalyse möchte man einen einfachen Ausdruck für das Laufzeitverhalten haben, der eng an den exakten Werten ist. Es gelten die folgenden Regeln.

Lemma 2.19: Komplementäre Symbole

O und Ω sind komplementär zueinander:

$$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n)).$$

Beweis 2.20:

Wir zeigen zunächst die Richtung $\Rightarrow$. Gilt $f(n) \in O(g(n))$, dann existieren Werte $c > 0$ und n_0 , sodass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot g(n)$. Dann lässt sich die Gleichung umstellen, sodass (ebenfalls für alle $n \geq n_0$) gilt: $g(n) \geq \frac{1}{c} \cdot f(n)$. Das bedeutet: Wir können $c' = \frac{1}{c}$ und $n'_0 = n_0$ wählen, wofür dann $g(n) \geq c' \cdot f(n)$ ($n \geq n'_0$) gilt. Damit sind die Bedingungen der Definition von $g(n) \in \Omega(f(n))$ direkt erfüllt. Die Rückrichtung $\Leftarrow$ lässt sich analog zeigen. $\blacksquare$

Korollar 2.21: Reflexivität

$$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n)).$$

Das Korollar folgt direkt aus Lemma 2.19.

Lemma 2.22: Transitivität

$$\begin{aligned} \text{Es gilt: } f(n) \in O(g(n)) \text{ und } g(n) \in O(h(n)) &\Rightarrow f(n) \in O(h(n)) \\ f(n) \in \Omega(g(n)) \text{ und } g(n) \in \Omega(h(n)) &\Rightarrow f(n) \in \Omega(h(n)) \\ f(n) \in \Theta(g(n)) \text{ und } g(n) \in \Theta(h(n)) &\Rightarrow f(n) \in \Theta(h(n)). \end{aligned}$$

Beispiel 2.23:

Sei $f(n) = 5 \cdot n + 17 \cdot \log n$ eine exakte Laufzeit. Dann gilt für $g(n) = n^2$ und $h(n) = n^3$ die Voraussetzung der ersten Gleichung aus Lemma 2.22: $5 \cdot n + 17 \cdot \log n \in O(n^2)$ und $n^2 \in O(n^3)$. Dann folgt aus Lemma 2.22, dass auch $5 \cdot n + 17 \cdot \log n \in O(n^3)$ gilt. $\square$

Beweis 2.24: (der ersten Gleichung des Lemmas)

Sei $c' \in \mathbb{R}^+$ und $n'_0 \in \mathbb{N}$ so gewählt, dass $f(n) \leq c' \cdot g(n)$ für $n \geq n'_0$. Ebenso gilt $g(n) \leq c'' \cdot h(n)$ für $n \geq n''_0$ mit Werten $c'' \in \mathbb{R}^+$ und $n''_0 \in \mathbb{N}$. Dann können wir $n'''_0 = \max\{n'_0, n''_0\}$ wählen und es gilt für $n \geq n'''_0$:

$$f(n) \leq c' \cdot g(n) \leq c' \cdot c'' \cdot h(n).$$

Mit $c''' = c' \cdot c''$ ist also die Definition von $f(n) \in O(h(n))$ erfüllt. $\blacksquare$

Auf die weiteren Beweise der anderen Teile des Lemmas und der folgenden Lemmata verzichten wir.

Lemma 2.25: Vereinfachung von Ausdrücken

$$\begin{aligned} f(n) \in O(g(n)) &\Rightarrow O((f+g)(n)) = O(g(n)) \\ f(n) \in \Omega(g(n)) &\Rightarrow \Omega((f+g)(n)) = \Omega(f(n)) \\ g(n) \in \Theta(f(n)) &\Rightarrow \Theta(f+g) = \Theta(g) = \Theta(f). \end{aligned}$$

Beispiel 2.26:

In einer Abschätzung nach oben können wir in einer Summe die schwächer wachsenden Summanden entfallen lassen, z.B. bei $f(n) = n$ und $g(n) = n \cdot \log n$ gilt $O(n + n \cdot \log n) = O(n \cdot \log n)$ wegen $f(n) \in O(g(n))$. Analog können die stärker wachsenden Summanden bei einer Abschätzung nach unten entfallen. $\square$

Beispiel 2.27:

In Beispiel 2.11 wurde die exakte Laufzeit $T(n) = 3 + \frac{27}{2} \cdot n + \frac{15}{2} \cdot n^2$ für Algorithmus 2.3 ermittelt. Da $3 + \frac{27}{2} \cdot n \in O(\frac{15}{2} \cdot n^2)$ gilt, folgt $T(n) \in O(\frac{15}{2} \cdot n^2)$. Und damit folgt letztendlich: $T(n) \in O(n^2)$. $\square$

Lemma 2.28: Berechnungsregel

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow f(n) \in \Theta(g(n))$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)).$$

Beispiel 2.29:

Betrachten wir $f(n) = \log n$ und $g(n) = \sqrt{n}$.

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{\log n}{\sqrt{n}} &= \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\underbrace{\frac{1}{2} \cdot n^{-\frac{1}{2}}}_{= \frac{\sqrt{n}}{2 \cdot n}}} && \text{(Regel von l'Hôpital)} \\ &= \lim_{n \rightarrow \infty} \frac{1}{2 \cdot \sqrt{n}} = 0. \end{aligned}$$

Daraus folgt $\log n \in O(\sqrt{n})$. $\square$

Korollar 2.30: Hierarchie der Polynome

$$m \in \mathbb{N} \Rightarrow O(n^m) \subset O(n^{m+1}).$$

Zum Abschluss dieses Abschnitts zu Laufzeitanalysen, wollen wir noch kurz untersuchen, welche Aussagen bei der Kombination von Landau-Symbolen mit Betrachtungen zum Worst-Case, Average-Case etc. entstehen. Wie Tabelle 2.4 zeigt muss man dabei insbesondere unterscheiden, ob man über die exakte Laufzeit oder bereits einen Extremfall spricht. Aussagen wie »der Worst-Case wird großzügig nach oben abgeschätzt« können insbesondere dann sinnvoll sein, wenn bestimmte Eigenschaften, wie etwa die Struktur der gespeicherten Daten beim Mengenproblem, offen gelassen werden.



Prima: jetzt kann ich verschiedene Algorithmen tatsächlich sinnvoll vergleichen. Besonders die Vereinfachungen von der

Tabelle 2.4: Mögliche Aussagen mit der Landau-Notation

	$O(\cdot)$	$\Theta(\cdot)$	$\Omega(\cdot)$
exakte Zeit	obere Schranke, d.h. Beschränkung des Worst-Case	asymptotisch fallen Worst-Case und Best-Case zusammen	untere Schranke, d.h. Beschränkung des Best-Case
Worst-Case	obere Schranke als eventuell großzügige Abschätzung, d.h. der Worst-Case kann auch besser sein	asymptotisch exakte Analyse des Worst-Case	untere Schranke, d.h. der Worst-Case kann noch schlechter sein
Average-Case	Average-Case ist asymptotisch nicht langsamer	asymptotisch exakte Beschreibung des Average-Case	Average-Case ist asymptotisch nicht schneller
Best-Case	obere Schranke, d.h. der Best-Case kann noch besser sein	asymptotisch exakte Analyse des Best-Case	untere Schranke als eventuell großzügige Abschätzung, d.h. der Best-Case kann auch schlechter sein

exakten Zählung der Operationen zu einer grundsätzlichen Aussage gefällt mir gut!
Jetzt frage ich mich nur noch, ob meine Probleme alle gleich schwierig sind?

2.4 Schwierigkeit von Problemen

Im letzten Unterkapitel wurde das Handwerkszeug für theoretische Laufzeitvergleiche von Algorithmen vorgestellt. Dies ist ein sinnvolles Mittel für die Bewertung verschiedener Algorithmen als Lösung für dasselbe Problem. Man kann daraus allerdings kaum Aussagen über die Schwierigkeit von Problemen ableiten.

Stellen wir uns zum Beispiel vor, wir wollen eine Aussage darüber treffen, ob das Sortierproblem schwieriger oder leichter als das Rundreiseproblem ist. Dann könnten wir die besten uns bekannten Algorithmen für beide Probleme heranziehen und diese vergleichen. Da uns bisher nur die Aufzählung aller Möglichkeiten eingefallen ist, wären dies beim Sortierproblem mit n Zahlen insgesamt $n!$ mögliche Anordnungen – ebenso wie beim Rundreiseproblem der Größe n . Mit einem jeweils ganz naiven Algorithmus wären also die Problem etwa gleich schwer. Allerdings können wir uns hier nicht sicher sein, ob es nicht noch schnellere Algorithmen gibt, die uns zu einem anderen Ergebnis des Vergleichs führen.

Letzendlich bleibt diese Unsicherheit, ob es keinen schnelleren Algorithmus geben kann, ohne genauere mathematische Beweise bestehen. Da-

her eignet sich eine derartige Betrachtung der Laufzeit nicht für eine Einordnung der Probleme bezüglich ihrer Schwierigkeit.

Im Weiteren wird die Grundidee eines besser geeigneten Ansatzes vorgestellt, bei dem tatsächlich die Probleme algorithmisch zueinander in Bezug gesetzt werden. Dieses Thema gehört in das Gebiet der Komplexitätstheorie und wird hier nur knapp und anschaulich angerissen. Für die Einordnung von Problemen vereinfachen wir zunächst von der Berechnung der Lösung zu sog. Entscheidungsproblemen.

Definition 2.31: Entscheidungsproblem

Bei einem **Entscheidungsproblem** wird zu einer Probleminstanz und einer Eigenschaft durch einen Algorithmus entschieden, ob eine entsprechende Lösung mit der Eigenschaft existiert.

► Entscheidungsproblem

Beispiel 2.32:

Das Rundreiseproblem lässt sich als Entscheidungsproblem formulieren, indem eine maximal zulässige Länge $maxL$ der Rundreise vorgeben wird. Ein Algorithmus zur Lösung dieses Problems muss genau dann wahr zurückgeben, falls eine Rundreise existiert, die nicht länger als $maxL$ ist. □



Ein Entscheidungsproblem liefert uns zwar nicht die Lösung, die wir eigentlich haben wollen. Doch für die Einordnung unserer Probleme in Klassen bzgl. ihrer Schwierigkeit lässt sich sicher sagen, dass das Finden der Lösung nicht leichter sein kann als das dazugehörige Entscheidungsproblem.

Definition 2.33: Klasse P

Ein Entscheidungsproblem ist genau dann in der **Klasse P** enthalten, wenn es einen klassischen Algorithmus gibt, der die Antwort in polynomieller Zeit berechnet. Dabei bedeutet polynomielle Zeit, dass es ein $k \in \mathbb{N}$ gibt, so dass der Algorithmus in $O(n^k)$ ist.

► Klasse P

Innerhalb der Klasse P ist es irrelevant, ob ein Algorithmus die Laufzeit $O(n^2)$, $O(n^3)$ oder $O(n^8)$ aufweist.

Definition 2.34: Klasse NP

Ein Entscheidungsproblem ist genau dann in der **Klasse NP**, wenn es für sie einen Algorithmus mit Polynomialzeit gibt, der zusätzlich einen Teil der Lösung »raten« darf und dann gleichzeitig für alle geratenen Teillösungen die Eigenschaft überprüft.

► Klasse NP

Trivialerweise ist jedes Problem aus P auch in NP enthalten – es gilt also $P \subseteq NP$, da zusätzlich zur geforderten Polynomialzeit noch geraten werden darf aber nicht muss.



Üblicherweise werden die Klassen P und NP mit Turing-Maschinen definiert. Für die Klasse NP wird dabei der Ratevorgang über nicht-deterministische Übergänge im Zustandsautomaten realisiert. Das N steht dabei für »nicht-deterministisch«, das P für »polynomiell«. Details können Standardlehrbüchern wie dem von Reischuk (1990) entnommen werden.

Derzeit ist nicht bekannt, ob die Klasse NP tatsächlich mehr Probleme als die Klasse P enthält. Es fehlt also ein Beweis, der für ein Problem aus NP zeigt, dass dieses nicht in P sein kann – oder dass $P = NP$ gilt. Für einen entsprechenden Beweis wurde im Jahr 2000 vom Clay-Mathematics-Institute ein Preis über 1 Million US-\$ ausgesetzt. Derzeit sind für die schwierigen Probleme in NP nur klassische Algorithmen mit exponentieller Laufzeit bekannt.



Die Klasse NP ist dabei noch nicht die schwierigste Klasse – so gibt es etwa Probleme, die tatsächlich nicht mit einer nicht-deterministischen Turingmaschine in polynomieller Zeit lösbar sind. Ein Beispiel wäre die Klasse PSPACE, welche diejenigen Probleme zusammenfasst, die polynomiell viel Speicherplatz benötigen. Beispiele hierfür wären Spiele wie Gobang oder eine Schachvariante, bei der ein Bauer nach einer bestimmten Anzahl an Zügen bewegt werden muss (siehe ebenfalls Reischuk, 1990).

Beispiel 2.35:

Ganz offensichtlich ist das SUCHEN des Mengenproblems – d.h. ist ein Element enthalten oder nicht – in P, da wir nur alle n Elemente kontrollieren müssen und damit eine Laufzeit $O(n)$ erreichen. Offensichtlich ist SUCHEN auch in NP, da wir das Element raten können und anschließend in $\Theta(1)$ überprüfen, ob es das gesuchte ist.

Analog kann man auch argumentieren, dass KÜRZESTER-WEG und RUNDREISE in NP sind, da wir den Weg oder die Rundreise raten und anschließend in $O(n)$ überprüfen, ob sie kurz genug sind. Tatsächlich werden wir in den weiteren Kapiteln sehen, dass KÜRZESTER-WEG sogar in P ist, was nicht für RUNDREISE gilt. $\square$



Ok! Alles, was man raten und dann in vernünftiger Zeit überprüfen kann, ist in NP.

Da $P \subseteq NP$ gilt, stellt sich nun die Frage, wie wir denn nun die richtig schwierigen Probleme in NP charakterisieren können. Dies geschieht in

der folgenden Definition mit einem wichtigen Konzept der Algorithmik – nämlich der Technik, ein Problem auf ein anderes Problem abzubilden.

Definition 2.36: NP-Vollständigkeit

Ein Problem p heißt **NP-vollständig**, wenn es in NP enthalten ist und sich für jedes Problem $p' \in \text{NP}$ jede Problemistanz aus p' durch einen Algorithmus in polynomieller Zeit auf eine Problemistanz aus p abbilden lässt, sodass man von der Lösung für p auf die Lösung für p' schließen kann.

► NP-vollständig

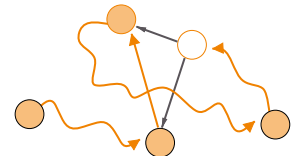
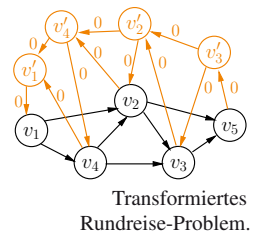
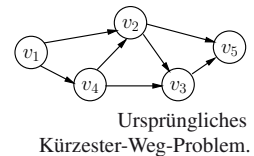
Die Abbildung eines Problems auf ein anderes in polynomieller Zeit bedeutet, dass wir aus der Schwierigkeit des ersten Problems auf die Schwierigkeit des zweiten Problems schließen können. Das zweite Problem kann damit nicht grundsätzlich leichter sein als das erste.

Beispiel 2.37:

An einem Beispiel soll kurz erläutert werden, wie eine solche Abbildung von einem in ein anderes Problem funktioniert. Hierfür soll das Problem KÜRZESTER-WEG (ohne Beschränkung der Allgemeinheit mit nur einem Zielknoten) auf das Problem RUNDREISE abgebildet werden.

Hierfür doppelten wir die Knoten (außer dem Start- und dem Zielknoten) und führen die rechts orange abgebildeten Kanten mit dem Gewicht 0 ein.

Offensichtlich ergibt sich eine Rundreise, wenn man den kürzesten Weg vom Start zum Ziel nimmt und anschließend über die orangefarbenen Kanten die restlichen Knoten »abholt«. Wie man auch leicht einsieht, können orangefarbene Knoten nicht auf dem Weg zum Zielknoten vorkommen, da diese nur dann, wenn sie in der vorgegebenen Reihenfolge durchlaufen werden, zurück zum Startknoten führen. Wird ein orangefarbener Knoten schon früher besucht, kann derjenige Knoten, der auf dem Rückweg direkt davor liegt, nicht mehr besucht werden, da es keine Kante mehr gibt, die von dem Knoten wieder zu einem noch freien Knoten weg führt. Folglich enthält die kürzeste Rundreise auch den kürzesten Weg. □

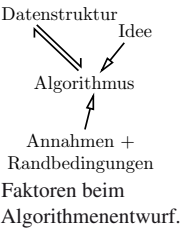


Damit ist RUNDREISE mindestens so schwierig wie KÜRZESTER-WEG. Da die andere Richtung allerdings nicht gezeigt werden kann, liegen die Probleme nicht in derselben Problemklasse – das Problem RUNDREISE fällt in die Klasse der NP-vollständigen Probleme, während KÜRZESTER-WEG in P liegt. Alle anderen hier betrachteten Problem liegen ebenfalls in P.



Die NP-Vollständigkeit wurde als erstes für die Erfüllbarkeit logischer Formeln gezeigt. Für andere Probleme reicht es, zu zeigen, dass sich ein NP-vollständiges Problem darauf reduzieren lässt und dass es in NP ist. Eine umfassende Übersicht liefert das Buch von Garey & Johnson (1979).

2.5 **Zutaten für effiziente Algorithmen**

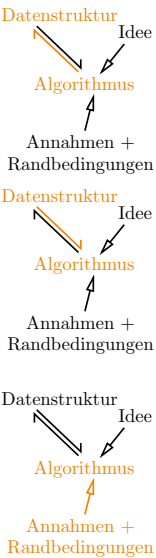


Der Entwurf eines effizienten Algorithmus steht in einer starken Wechselbeziehung zur benutzten Datenstruktur. Zudem wird der Algorithmus stark von der »genialen« algorithmischen Idee sowie von Annahmen und Randbedingungen beeinflusst. Das Ziel ist dabei in der Regel nicht nur ein korrekter Algorithmus, sondern auch ein möglichst sparsamer Umgang mit den Ressourcen Zeit und Speicherplatz.

Die unterschiedlichen Aspekte werden im Weiteren für das folgende dem Sortierproblem verwandten Problem beispielhaft illustriert.

```
www
IMedian.java
```

► **MEDIAN**



Definition 2.38: Medianproblem

Für eine unsortierte Folge von n Objekten mit den Schlüsseln $a_1, \dots, a_n$ ist der **MEDIAN** als dasjenige Element $a_{\pi(m)}$ gesucht, das nach einer Sortierung π genau in der Mitte $m = \lceil \frac{n}{2} \rceil$ der Folge $a_{\pi(1)} \leq \dots \leq a_{\pi(n)}$ stünde.

Das Medianproblem lässt sich einfach dadurch lösen, dass wir das Sortierproblem lösen (vgl. Algorithmus 2.5). Der Sortieralgorithmus bestimmt die benötigte Datenstruktur. Wie wir in den folgenden Kapiteln sehen werden, ist damit die Mediansuche mit Laufzeit $O(n^2)$ oder $O(n \cdot \log n)$ möglich.

Nehmen wir im Gegenzug an, dass die Datenstruktur fest ist – z.B. eine Liste, in der nur sequentiell von vorn nach hinten lesend auf die Elemente zugegriffen wird (oder gar ein Datenstrom, bei dem jedes Element nur einmal gelesen wird). Dann muss ein zur Datenstruktur passender Algorithmus gesucht werden – der Ansatz über das Sortieren funktioniert hier nicht.

Wenn wir allerdings zusätzliches Wissen über Randbedingungen und Annahmen über Eigenschaften der vorliegenden Daten besitzen, lässt sich dieses Wissen natürlich auch beim Algorithmenentwurf berücksichtigen. Gehen wir etwa bei der Mediansuche davon aus, dass nur $k \in \mathbb{N}$ verschiedene Werte vorkommen können, lässt sich der Median durch reines Abzählen der verschiedenen Werte bestimmen. Algorithmus 2.6 (**MEDIAN-ZÄHLEN**) bestimmt zu jedem Schlüsselwert die Häufigkeit des Vorkommens und errechnet daraus den Median. Unter der stark einschränkenden Randbedingung der endlichen Werte, kann man hier eine lineare Laufzeit $\Theta(n + k)$ erreichen. Sind diese Randbedingungen erfüllt, können wir auch bei einem Nur-Lesen-Zugriff auf die Daten den Median schnell (sogar schneller als bei normalem Zugriff durch Sortieren) bestimmen.

Algorithmus 2.5 Medianbestimmung durch Sortieren und Rückgabe des mittleren Elements der sortierten Folge

```
www
MedianSortiert.java
```

```
MEDIAN-SORTIERT(Feld A)
Rückgabewert: Wert des mittleren Elements
1  SORTIERE(A)
2  return A[ $\lceil \frac{A.länge}{2} \rceil$ ]
```

Algorithmus 2.6 Medianbestimmung durch Zählen bei endlichen Wertemen- gen

MEDIAN-ZÄHLEN(Feld A , größter Schlüssel k)

Rückgabewert: Wert des mittleren Elements

```

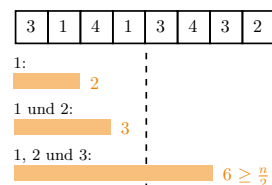
1  for  $i \leftarrow 1, \dots, k$ 
2  do  $\lceil \text{anzahl}[i] \leftarrow 0$  } Zähler initialisieren
3  for  $j \leftarrow 1, \dots, A.\text{länge}$ 
4  do  $\lceil \text{anzahl}[A[j]] \leftarrow \text{anzahl}[A[j]] + 1$  } Anzahl für jeden
                                           Schlüssel bestimmen
5   $\text{anzahlKleiner} \leftarrow 0$ 
6   $i \leftarrow 0$ 
7  while  $\text{anzahlKleiner} < \frac{A.\text{länge}}{2}$ 
8  do  $\lceil i \leftarrow i + 1$ 
9       $\lceil \text{anzahlKleiner} \leftarrow \text{anzahlKleiner} + \text{anzahl}[i]$  } den Median
                                                                    aus den
                                                                    Zählern
                                                                    ableiten
10 return  $i$ 
```

www

MedianCount.java

Beispiel 2.39:

Das nebenstehende Bild zeigt ein Feld der Länge $n = 8$, dessen Einträge $k = 4$ unterschiedliche Schlüssel ($\{1, 2, 3, 4\}$) sind. Der Algorithmus bestimmt für jeden Schlüssel die Anzahl seines Vorkommens. In der While-Schleife werden diese Anzahlen nacheinander addiert und überprüft, ob das jeweilige Ergebnis kleiner n ist. Bei der Schlüsselmenge $\{1, 2, 3\}$ wird die Hälfte der Feldlänge erreicht. Das bedeutet, dass der letzte zur Menge hinzugefügte Schlüssel, hier die 3, der Median sein muss. □



Medianbestimmung durch
Abzählen.

Dass die lineare Laufzeit grundsätzlich auch im allgemeinen Fall möglich ist, werden wir im weiteren Verlauf des Buchs (in Abschnitt 7.4) sehen. Dem dort beschriebenen Algorithmus liegt der letzte der Faktoren des Algorithmenentwurfs zugrunde: eine geniale Idee, wie auf besonders raffinierte Art und Weise die gesuchte Information berechnet werden kann. Solche Ideen fußen häufig auf den sogenannte **Entwurfsparadigmen**, die einen generischen Ansatz für die Entwicklung von Algorithmen beschreiben, oder auch auf mathematisch/theoretischen Überlegungen.

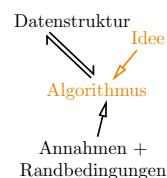
Da jedes Kapitel in diesem Lehrbuch einem speziellen Entwurfsgedanken gewidmet ist, könnte man an dieser Stelle die Kapitel zu den verschiedenen Faktoren des Algorithmenentwurfs wie folgt zuordnen:

Datenstrukturen: Grundsätzliche Datenstrukturen werden in Kapitel 3 und Kapitel 4 vorgestellt. Die spezielle Datenstruktur der Prioritätswarteschlange wird in Kapitel 10 entwickelt.

Annahmen und Randbedingungen: In Kapitel 6 sind Laufzeiterwägungen die treibende Kraft. Die Menge der zu speichernden Daten ist in Kapitel 11 thematisiert. In beiden Kapiteln werden aus den betrachteten Aspekten heraus auch neue Datenstrukturen entwickelt.

Ideen: Große Ideen in Form von Entwurfsparadigmen ziehen sich durch die Algorithmik und sind explizit in den Kapiteln 5, 7 und 8 dargestellt.

► Entwurfsparadigma



Weitere Ideen, wie Algorithmen arbeiten können, werden in den Kapiteln 9 und 12 betrachtet.

Wir haben gesehen, dass alle Probleme aus Kapitel 1 grundsätzlich algorithmisch lösbar sind – es bleibt jedoch die Frage offen, ob dies mit effizienter Laufzeit möglich ist. Als Grundlage für die kommenden Kapitel wurden Begriffe und Notationen zur Beschreibung der Algorithmen und ihrer Laufzeit eingeführt. Am Beispiel des Medianproblems haben wir auch die Faktoren und Wechselwirkungen beim Entwurf eines Algorithmus beleuchtet.

Übungsaufgaben

Aufgabe 2.1: Textsuche

Führen Sie EINFACHE-TEXTSUCHE (Algorithmus 2.1) auf den folgenden Eingaben durch: $T = aababbbaaab$ und $S = aab$. Dokumentieren Sie alle Teilschritte. Wie oft wird der Vergleich von zwei Zeichen in Zeile 4 erfolgreich bzw. erfolglos durchgeführt?

Aufgabe 2.2: Exakte Laufzeit

Betrachten Sie die folgenden beiden Algorithmen.

LINKERALG(n)

Rückgabewert: Wert sum

```

1   $sum \leftarrow 0$ 
2  for  $i \leftarrow 1, \dots, n$ 
3  do for  $k \leftarrow i, \dots, n$ 
4      do  $sum \leftarrow sum + k$ 
5  return  $sum$ 
```

RECHTERALG(n)

Rückgabewert: Wert sum

```

1   $sum \leftarrow 0$ 
2  for  $k \leftarrow 1, \dots, n$ 
3  do  $sum \leftarrow sum + k \cdot k$ 
4  return  $sum$ 
```

- Bestimmen Sie die exakte Anzahl der Zuweisungen $sum \leftarrow \dots$ für beide Algorithmen in Abhängigkeit von n .
- Zeigen Sie, dass beide Algorithmen dasselbe Ergebnis berechnen. Betrachten Sie hierfür zunächst für einen kleinen Wert von n , welche Terme addiert werden.
- Bestimmen Sie die exakte Schrittzahl der beiden Algorithmen, wobei jede Wertzuweisung, jede Addition und jede Multiplikation als ein Schritt zählen – berücksichtigen Sie dabei auch die versteckten Operationen der Schleifen. Zur Vereinfachung der Terme können Sie Satz B.1 (Seite 339) benutzen.

Aufgabe 2.3: Konstante Faktoren

Gegeben seien fünf Algorithmen mit der jeweiligen *Worst-Case* Laufzeit (in Mikrosekunden): $5000 \cdot \log_2 n$, $500 \cdot n$, $50 \cdot n \cdot \log_2 n$, $5 \cdot n^2$ und 2^{n-1} .

- a) Falls eine Stunde Rechenzeit zur Verfügung steht, welche Problemgröße n wäre für die einzelnen Algorithmen noch berechenbar?
- b) Geben Sie für jeden Algorithmus das Intervall der Werte für n an, für die der Algorithmus eine bessere Worst-Case Laufzeit als die anderen Algorithmen besitzt.

Es reicht, wenn Sie die Aufgaben durch »Probieren« lösen.

Aufgabe 2.4: Beweis der Lemmata

Beweisen Sie die zweite und dritte Aussage von Lemma 2.22 und Lemma 2.25, indem sie die Definition der Landau-Notation benutzen.

Aufgabe 2.5: Asymptotische Komplexität

Zeigen Sie die folgenden Aussagen:

- a) $n \in O(20 \cdot n \cdot \log n)$
- b) $40 \cdot n \cdot \log n \in O(n^2)$
- c) $\sqrt{20 \cdot n} \in \Omega(7 \cdot \log n)$
- d) $5 \cdot n^2 + 37 \cdot n \in \Theta(n^2)$.

Aufgabe 2.6: Asymptotische Komplexität und Logarithmen

Zeigen Sie unter Nutzung der Logarithmengesetze (Abschnitt B.3), dass $O(\log_2 n) = O(\ln n)$ gilt.

Aufgabe 2.7: Vergleich von Laufzeiten

Algorithmus A habe polynomielle Laufzeit $T_A(n) = 200 \cdot n^2$. Algorithmus B habe exponentielle Laufzeit $T_B(n) = 2^n$. Für welche Werte von n sind Algorithmus A bzw. Algorithmus B jeweils schneller? Zeigen und beweisen Sie, wie sich die Algorithmen asymptotisch zueinander verhalten.

Algorithmen und Datenstrukturen

Weicker, K.; Weicker, N.

2013, IX, 356 S. 91 Abb. in Farbe., Softcover

ISBN: 978-3-8348-1238-4