

Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik VI

Algorithmen und Datenstrukturen

Vorlesungsmitschrift zur Vorlesung im SS 2004

Prof. Dr.-Ing. H. Ney

Letzte Überarbeitung: 4. Mai 2004
Thomas Deselaers

Inhaltsverzeichnis

1	Grundlagen	8
1.1	Einführung	8
1.1.1	Problemstellungen	8
1.1.2	Aktualität des Themas	9
1.1.3	Ziele der Vorlesung	9
1.1.4	Hinweis auf das didaktische Problem der Vorlesung	9
1.1.5	Vorgehensweise	10
1.1.6	Datenstrukturen	10
1.2	Algorithmen und Komplexität	11
1.2.1	Random-Access-Machine (RAM)	13
1.2.2	Komplexitätsklassen	16
1.2.3	O-Notation für asymptotische Aussagen	17
1.2.4	Regeln für die Laufzeitanalyse	19
1.2.5	Beispiel: Fibonacci-Zahlen	21
1.2.6	Beispiel: Fakultät	24
1.3	Datenstrukturen	24
1.3.1	Datentypen	24
1.3.2	Listen	27
1.3.3	Stacks	37
1.3.4	Queues	39
1.3.5	Bäume	41
1.3.6	Abstrakte Datentypen	52
1.4	Entwurfsmethoden	54
1.4.1	Divide-and-Conquer-Strategie	54
1.4.2	Dynamische Programmierung	57
1.4.3	Memoization	60
1.5	Rekursionsgleichungen	61
1.5.1	Sukzessives Einsetzen	61
1.5.2	Master-Theorem für Rekursionsgleichungen	62
1.5.3	Rekursionsungleichungen	66

2	Sortieren	68
2.1	Einführung	68
2.2	Elementare Sortierverfahren	71
2.2.1	SelectionSort	71
2.2.2	InsertionSort	73
2.2.3	BubbleSort	76
2.2.4	Indirektes Sortieren	80
2.2.5	BucketSort	82
2.3	QuickSort	84
2.3.1	Beweis der Schranken von $\sum_{k=m}^N 1/k$ mittels Integral-Methode	94
2.4	HeapSort	95
2.4.1	Komplexitätsanalyse von HeapSort	99
2.5	Untere und obere Schranken für das Sortierproblem	101
2.6	Schranken für $N!$	104
2.7	MergeSort	107
2.8	Zusammenfassung	107
3	Suchen in Mengen	108
3.1	Problemstellung	108
3.2	Einfache Implementierungen	110
3.2.1	Ungeordnete Arrays und Listen	110
3.2.2	Vergleichsbasierte Methoden	110
3.2.3	Bitvektordarstellung (Kleines Universum)	113
3.2.4	Spezielle Array-Implementierung	114
3.3	Hashing	116
3.3.1	Begriffe und Unterscheidung	116
3.3.2	Hashfunktionen	118
3.3.3	Wahrscheinlichkeit von Kollisionen	119
3.3.4	Offenes Hashing (Hashing mit Verkettung)	122
3.3.5	Geschlossene Hashverfahren (Hashing mit offener Adressierung)	124
3.3.6	Zusammenfassung der Hashverfahren	129
3.4	Binäre Suchbäume	129
3.4.1	Allgemeine binäre Suchbäume	129
3.4.2	Der optimale binäre Suchbaum	142
3.5	Balancierte Bäume	148
3.5.1	AVL-Bäume	148
3.5.2	(a,b) -Bäume	154
3.6	Priority Queue und Heap	156
4	Graphen	158
4.1	Motivation: Wozu braucht man Graphen?	158
4.2	Definitionen und Graph-Darstellungen	159
4.2.1	Graph-Darstellungen	160
4.2.2	Programme zu den Darstellungen	162

4.3	Graph-Durchlauf	163
4.3.1	Programm für den Graph-Durchlauf	164
4.4	Kürzeste Wege	169
4.4.1	Dijkstra-Algorithmus (Single-Source Best Path)	169
4.4.2	Floyd-Algorithmus (All Pairs Best Path)	173
4.4.3	Warshall-Algorithmus	176
4.5	Minimaler Spannbaum	178
4.5.1	Definitionen	178
4.5.2	MST Property	178
4.5.3	Prim-Algorithmus	179
4.6	Zusammenfassung	181

Literaturverzeichnis

- [Cormen et al.] T.H. Cormen, C.E. Leiserson, R.L. Rivest: *Introduction to Algorithms*. MIT press / McGraw Hill, 10th printing, 1993.
- [Sedgewick 02] R. Sedgewick: *Algorithms in Java*, 3rd ed., Addison-Wesley, 2002.
- [Sedgewick 98] R. Sedgewick: *Algorithms*, 2nd ed., Addison-Wesley, 1998.
- [Sedgewick 88] R. Sedgewick: *Algorithms*. Addison-Wesley, 1988.
- [Sedgewick 93] R. Sedgewick: *Algorithms in Modula-3*. Addison-Wesley, 1993.
- [Schöning 97] U. Schöning: *Algorithmen – kurz gefasst*. Spektrum Akad. Verl., 1997, vergriffen.
- [Schöning 01] U. Schöning: *Algorithmik*. Spektrum Akad. Verl., 2001.
- [Güting] R.H. Güting: *Datenstrukturen und Algorithmen*. Teubner, 1992.
- [Aho et al. 83] A.V. Aho, J.E. Hopcroft, J.D. Ullman: *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [Aho et al. 74] A.V. Aho, J.E. Hopcroft, J.D. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [Mehlhorn] K. Mehlhorn: *Datenstrukturen und effiziente Algorithmen*. Bde 1,2,3; (primär Band 1: „Sortieren und Suchen“), Teubner 1988. (Bde 2+3 vergriffen)
- [Wirth] N. Wirth: *Algorithmen und Datenstrukturen mit Modula-2*. 4. Auflage, Teubner 1986.
- [Aigner] M. Aigner: *Diskrete Mathematik*. Vieweg Studium, 1993.
- [Ottmann] T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen*. 4. Auflage, Spektrum Akademischer Verlag, 2002

Hinweise zur Literatur

[Cormen et al.]	beste und modernste Darstellung
[Sedgewick 02, Sedgewick 98, Sedgewick 88]	konkrete Programme
[Schöning 97]	sehr kompakte und klare Darstellung
[Güting]	schöne Darstellung des Suchens in Mengen
[Mehlhorn]	Motivation und Analyse der Algorithmen
[Wirth]	viel Modula-2
allgemein:	DUDEN Verlag: Schülerduden INFORMATIK, bzw. Duden INFORMATIK (fast identisch)

1 Grundlagen

1.1 Einführung

1.1.1 Problemstellungen

Diese Vorlesung befaßt sich mit Algorithmen und den zugehörigen Datenstrukturen. Dabei werden sowohl konkrete Implementierungen erarbeitet als auch Analysen ihrer Komplexität bezüglich Rechenzeit und Speicherplatzbedarf angestellt. Wir werden folgende Problemstellungen der Informatik betrachten:

Sortieren

- Sortierung einer Folge von Zahlen
- Bestimmung des Median

Suchen in Mengen

- Symboltabellen für Compiler (enthalten vordefinierte Wörter, Variablen, etc.)
- Autorenverzeichnis einer Bibliothek
- Kontenverzeichnis einer Bank
- allgemein Suchen in Datenbanken

Pfade in einem Graphen

- kürzester Pfad von A nach B
- Minimaler Spannbaum (kürzester Pfad, der alle Knoten miteinander verbindet)
- kürzeste Rundreise (Traveling-Salesman-Problem (TSP))

String-Matching

- Suche nach Zeichenfolgen im Text
- DNA/RNA - Sequenzen in der Bioinformatik

1.1.2 Aktualität des Themas

Die Suche nach guten Algorithmen und ihrer effizienten Implementierung ist stets aktuell gewesen. Auch in Zeiten immer schneller wachsender Speicher- und Rechenkapazitäten ist es aus ökonomischen Gründen unerlässlich, diese möglichst effizient zu nutzen. Wie wir sehen werden, können ineffiziente Algorithmen auch noch so große Kapazitäten sehr schnell erschöpfen.

Auf der theoretischen Seite existieren immer noch zahlreiche ungeklärte Fragestellungen: So ist zum Beispiel bisher keine effiziente, exakte Lösung für das Traveling-Salesman-Problem bekannt. Daneben gibt es in der angewandten Informatik viele Gebiete, die nach extrem effizienten Algorithmen verlangen, z.B.: Compilerbau, Datenbanken, künstliche Intelligenz, Bild- und Sprachverarbeitung und -erkennung.

1.1.3 Ziele der Vorlesung

Sie sollten am Ende der Vorlesung Kenntnisse und Fähigkeiten in den folgenden Bereichen erlangt haben:

Theoretische Kenntnisse: Was zeichnet effiziente Verfahren aus, und welche Algorithmen und Datenstrukturen stecken dahinter? Wie ist der Aufwand (Rechenzeit, Speicherplatz) eines Verfahrens definiert? Wie kann man diesen Aufwand in Komplexitätsmaße fassen, und wie wird ein Algorithmus analysiert?

Praktische Fähigkeiten: Wie analysiert man ein Problem und bildet es auf Datenstrukturen und Algorithmen ab? Wie implementiert man den Algorithmus als lauffähiges Programm und testet dieses dann?

1.1.4 Hinweis auf das didaktische Problem der Vorlesung

Es ist wesentlich schwieriger zu beschreiben, wie man von der Problemstellung zum fertigen Programm kommt, als im nachhinein das fertige Programm zu erklären und nachzuvollziehen. Daher konzentrieren sich die meisten Bücher mehr auf die fertigen Programme als auf die Motivation und den Weg zum Programm.

Gelegentlich werden wir auch in der Vorlesung diesen Weg gehen. Versuchen Sie dann selbst, die Herleitung und Implementierung dieser Standard-Algorithmen nachzuvollziehen. Implementieren Sie diese vielleicht nach einigen Tagen noch einmal aus freier Hand. Verifizieren Sie insbesondere, daß Sie auch Details wie z.B. Index-Grenzen richtig hinbekommen, denn der Teufel steckt im Detail.

Aus den folgenden Gebieten der *Mathematik* werden immer wieder Anleihen gemacht, achten Sie deshalb darauf, daß Ihnen deren Grundlagen vertraut sind:

- Kombinatorik (Fakultät, Binomialkoeffizienten)
- Wahrscheinlichkeitsrechnung
- elementare Analysis (hauptsächlich Grenzwerte)

1.1.5 Vorgehensweise

Bei der Behandlung von Algorithmen und Datenstrukturen werden zwei Ebenen unterschieden:

Die algorithmische Ebene umfaßt eine möglichst allgemeine und maschinen-unabhängige Beschreibung der Objekte (zu manipulierende Daten) und des Algorithmus. Dies hat den Vorteil, daß man sich auf das Wesentliche konzentrieren kann, und sich nicht mit maschinen- und programmiersprachen-spezifischen Details aufhalten muß.

Die programmiersprachliche Ebene umfaßt die konkrete Implementierung. Der Übersichtlichkeit wegen werden in der Vorlesung nicht immer vollständig lauffähige Programme vorgestellt. Zum Verständnis der Programmtexte ist die grundlegende Kenntnis einer imperativen Programmiersprache Voraussetzung. Programme und Algorithmen sind hier (fast immer) in Modula-3 oder Modula-3-ähnlichem Pseudocode formuliert.

1.1.6 Datenstrukturen

Datenstrukturen und Algorithmen sind unmittelbar miteinander verknüpft und können nicht getrennt voneinander betrachtet werden, da ein Algorithmus mit den Methoden arbeiten muß, die auf einer Datenstruktur definiert (und implementiert) sind. In den folgenden Kapiteln werden Datenstrukturen aus den folgenden Kategorien vorgestellt:

- Sequenzen (Folgen, Listen) (Abschnitt 1.3.2)
- Mengen (speziell Wörterbücher (dictionaries)) (Abschnitt 3)
- Graphen (speziell Bäume) (Abschnitt 1.3.5)

1.2 Algorithmen und Komplexität

Die Komplexitätsordnung eines Algorithmus bestimmt maßgeblich die Laufzeit und Platzbedarf der konkreten Implementierung. Diese hängen aber, außer vom Algorithmus selbst, u.a. noch von den folgenden Größen ab:

- Eingabedaten
- Rechner-Hardware
- Qualität des Compilers
- Betriebssystem

Die letzten drei Punkte sollen uns hier nicht interessieren. Wir betrachten nur die *abstrakte Laufzeit* $T(n)$ in Abhängigkeit von dem Parameter n der Eingabedaten. Dabei drückt der Parameter n die “Größe” des Problems aus, die von Fall zu Fall unterschiedlich sein kann:

- Beim *Sortieren* ist n die Anzahl der zu sortierenden Werte a_1, \dots, a_n .
- Bei der *Suche nach Primzahlen* gibt n z.B. die obere Grenze des abzusuchenden Zahlenbereichs an.

Beispiel: Sortieren durch Auswählen

Zur Veranschaulichung betrachten wir ein einfaches Sortierverfahren, *Sortieren durch Auswählen*. Es besteht aus zwei verschachtelten Schleifen und kann wie folgt implementiert werden:

```
PROCEDURE SelectionSort(VAR a: ARRAY [1..N] OF ItemType) =
VAR min : INTEGER ;
    t   : ItemType ;
BEGIN
    FOR i := 1 TO N - 1 DO
        min := i ;
        FOR j := i + 1 TO N DO
            IF a[j] < a[min] THEN
                min := j ;
            END ;
        END ;
        t := a[min] ; a[min] := a[i] ; a[i] := t ;
    END ;
END SelectionSort ;
```

Die Rechenzeit $T(n)$ wird hier bestimmt durch den Aufwand für die elementaren Operationen *vergleichen* und *vertauschen*. Deren Anzahl können wir in Abhängigkeit von der Anzahl n der zu sortierenden Elemente wie folgt berechnen:

- Anzahl der Vergleiche (compares):

$$C(n) = \sum_{i=1}^{n-1} (n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- Anzahl der Vertauschungen (exchanges, swaps):

$$E(n) = n - 1$$

Dies gilt, weil Vertauschungen nur in der äußeren Schleife ausgeführt werden, und diese genau $(n-1)$ -mal durchlaufen wird.

- Bezeichnet man den relativen Aufwand für Vergleiche und Vertauschungen mit α_1 bzw. α_2 , dann ergibt sich der folgende Gesamtaufwand:

$$\begin{aligned} T(n) &= \alpha_1 C(n) + \alpha_2 E(n) \\ &= \alpha_1 \frac{n(n-1)}{2} + \alpha_2 (n-1) \\ &\cong \alpha_1 \frac{n^2}{2} \quad \text{für große } n \quad (n \gg \frac{\alpha_2}{\alpha_1}) \end{aligned}$$

Diese Aussage gilt für große n . Wir sagen, daß das *asymptotische Verhalten* von $T(n)$ dem von n^2 entspricht oder daß $T(n)$ quadratische Komplexität hat.

1.2.1 Random-Access-Machine (RAM)

Die abstrakte Laufzeit ergibt sich aus der Zahl der elementaren Operationen. Um genaue, vergleichbare Aussagen über die Komplexität eines Programms machen zu können, muß definiert werden, was unter einer elementaren Operation zu verstehen ist. Dazu eignet sich ein axiomatisch definiertes Rechnermodell. Dieser Modell-Rechner dient dann als Vergleichsmaßstab.

Ein solches Rechnermodell stellt die *verallgemeinerte Registermaschine* (random access machine, RAM) dar. Eine RAM wird durch folgende Elemente definiert (vgl. Abb. 1.1):

- adressierbare Speicherzellen (random access registers), deren Inhalte ganzzahlige Werte (integers) sind
- Ein- und Ausgabebänder, die nur von links nach rechts bewegt werden können
- zentrale Recheneinheit mit:
 - Akkumulator
 - Befehlszähler (program counter)
- Programm (Der Programmcode soll sich selbst nicht verändern können und wird daher außerhalb des adressierbaren Speichers abgelegt.)

Das Programm wird in einem sehr rudimentären Befehlssatz (abstrakter Assembler) verfaßt (vgl. Tabelle 1.1). Einzelheiten können variieren; so ist der Akkumulator oft kein eigener Speicher, sondern einfach die Speicherzelle mit der Nummer 0. Variationen gibt es auch beim Befehlssatz, den möglichen Datentypen usw.

Anmerkung: Die (moderneren) RISC-Prozessoren (reduced instruction set computer) sind in ihrer Architektur einer RAM sehr ähnlich. Die konkurrierenden CISC-Architekturen (complex instruction set computer) haben einen wesentlich umfangreicheren Befehlssatz, und zerlegen die komplexen Befehle bei der Ausführung in einfachere Arbeitsschritte.

Kostenabschätzung

Je nach Zielsetzung kann man zwischen zwei Kostenmaßen wählen:

Einheitskostenmaß: Jeder Instruktion wird ein konstanter Aufwand zugeordnet, unabhängig von den jeweiligen Operanden. Dies entspricht dem Fall, daß die zu verarbeitenden Werte die Wortlänge des Rechners nicht überschreiten.

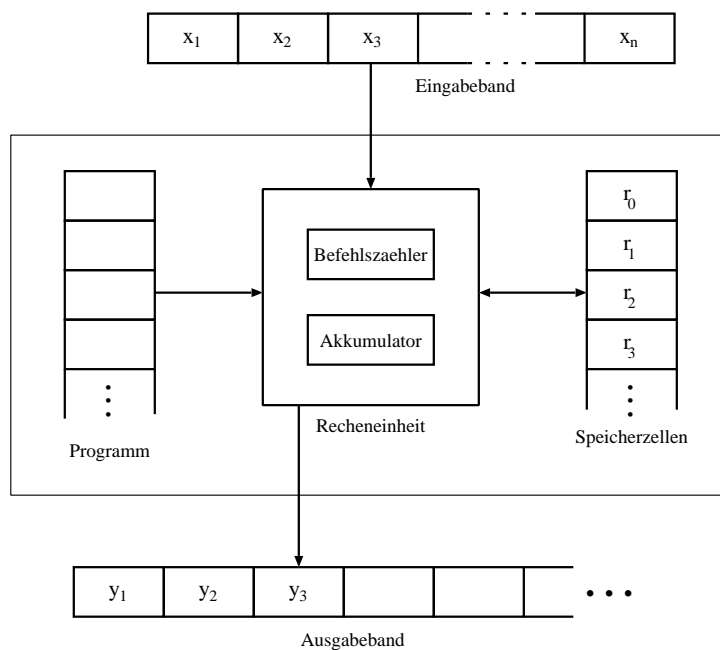


Abbildung 1.1: Schematische Darstellung der Random-Access-Machine (RAM)

Tabelle 1.1: Befehlssatz einer Random-Access-Machine (RAM): Es steht $c(\cdot)$ für „Inhalt von“, A für „Akkumulator“. Der Operand a ist entweder i , $c(i)$, oder $c(c(i))$ mit $i \in \mathbb{N}$, entsprechend den Adressierungsarten immediate, direkt und indirekt.

LOAD a	$c(A) \leftarrow a$
STORE a	$a \leftarrow c(A)$
ADD a	$c(A) \leftarrow c(A) + a$
SUB a	$c(A) \leftarrow c(A) - a$
MUL a	$c(A) \leftarrow c(A) * a$
DIV a	$c(A) \leftarrow \lfloor c(A)/a \rfloor$
READ	$c(A) \leftarrow$ aktuelles Eingabesymbol
WRITE	aktuelles Ausgabesymbol $\leftarrow c(a)$
JUMP b	Programmzähler wird auf Anweisung Nummer b gesetzt.
JZERO b	Programmzähler wird auf Anweisung Nummer b gesetzt, falls $c(A) = 0$, sonst auf die nächste Anweisung.
JGTZ b	Programmzähler wird auf Anweisung Nummer b gesetzt, falls $c(A) > 0$, sonst auf die nächste Anweisung.
HALT	Verarbeitung wird beendet.

Logarithmisches Kostenmaß: (auch Bit-Kostenmaß) Die Kosten sind von der Länge des Operanden (Anzahl der Bits) abhängig. Dies ist angebracht, wenn die Operanden mehr als je eine Speicherzelle (Wort) belegen.

Das logarithmische Kostenmaß entspricht dem Komplexitätsbegriff auf einer Turing-Maschine.

Die Kosten, die einem Befehl zugeordnet werden, sind wenig standardisiert. Man versucht in der Regel eine möglichst realistische Schätzung zu finden. Folgende Faktoren spielen dabei eine Rolle:

- Art des Speicherzugriffs: keiner, direkt, indirekt
- Art des Befehls
 - Arithmetische Operationen: MUL und DIV können wesentlich teurer sein als ADD und SUB.
 - Sprungbefehle: Bedingte Sprünge (JZERO) können teurer sein als unbedingte (JUMP)

Die RAM erfasst (ziemlich) genau die im realen Rechner anfallenden Speicher-Operationen. Externe Speicher wie z.B. Festplatten werden nicht betrachtet.

Prozeduraufrufe und Rekursion

Der Befehlssatz der RAM enthält kein Konstrukt zur Definition von Unterprogrammen (Funktionen, Prozeduren), die insbesondere für rekursive Algorithmen benötigt werden. Prozeduren, wie sie in Modula-3 und vielen anderen Sprachen zur Verfügung stehen, werden vom Compiler nach folgender Methode in elementare Anweisungen der RAM umgewandelt:

1. Bei jedem Aufruf (Inkarnation, Aktivierung) wird ein Speicherbereich, der sogenannte Aktivierungsblock (Versorgungsblock), (auf dem sog. call-stack) reserviert. Dieser enthält Platz für
 - die aktuellen Parameter,
 - die Rücksprungadresse und
 - die lokalen Variablen der Prozedur.
2. Das aufrufende Programm legt die aktuellen Parameter und die Rücksprungadresse im Aktivierungsblock ab und setzt den Programmzähler auf den Anfang der aufzurufenden Prozedur.

3. Nach Beendigung der Prozedur wird der zugehörige Aktivierungsblock entfernt und die Kontrolle an das aufrufende Programm zurückgegeben.

Der Speicherbedarf für die Aktivierungsblöcke muß natürlich bei der Analyse der Speicherkomplexität berücksichtigt werden. Bei rekursiven Algorithmen wächst der Speicherbedarf proportional zur Rekursionstiefe, da für jeden individuellen Aufruf ein eigener Aktivierungsblock angelegt wird.

1.2.2 Komplexitätsklassen

Um Algorithmen bezüglich ihrer Komplexität vergleichen zu können, teilt man diese in Komplexitätsklassen ein. In der folgenden Tabelle sind typische Zeitkomplexitätsklassen aufgeführt.

Klasse	Bezeichnung	Beispiel
1	konstant	elementarer Befehl
$\log(\log n)$	doppelt logarithmisch	Interpolationssuche
$\log n$	logarithmisch	binäre Suche
n	linear	lineare Suche, Minimum einer Folge
$n \log n$	überlinear	Divide-and-Conquer-Strategien, effiziente Sortierverfahren, schnelle Fourier- Transformation (FFT)
n^2	quadratisch	einfache Sortierverfahren
n^3	kubisch	Matrizen-Inversion, CYK-Parsing
n^k	polynomiell vom Grad k	lineare Programmierung
2^n	exponentiell	erschöpfende Suche (exhaustive search), Backtracking
$n!$	Fakultät	Zahl der Permutationen (Traveling-Salesman-Problem)
n^n		

Bei den logarithmischen Komplexitätsklassen spielt die Basis in der Regel keine Rolle, da ein Basiswechsel einer Multiplikation mit einem konstanten Faktor entspricht. Wenn eine Basis nicht explizit angegeben wird, dann geht diese für $\log(n)$ meist aus dem Zusammenhang hervor. Außerdem sind folgende Schreibweisen üblich:

$\text{ld} := \log_2$	dualer Logarithmus
$\text{ln} := \log_e$	natürlicher Logarithmus
$\text{lg} := \log_{10}$	dekadischer Logarithmus

Betrachtet man die Laufzeit einiger hypothetischer Algorithmen (siehe Abb. 1.2), so wird schnell klar, daß in den meisten Fällen nur Algorithmen mit maximal polynomieller

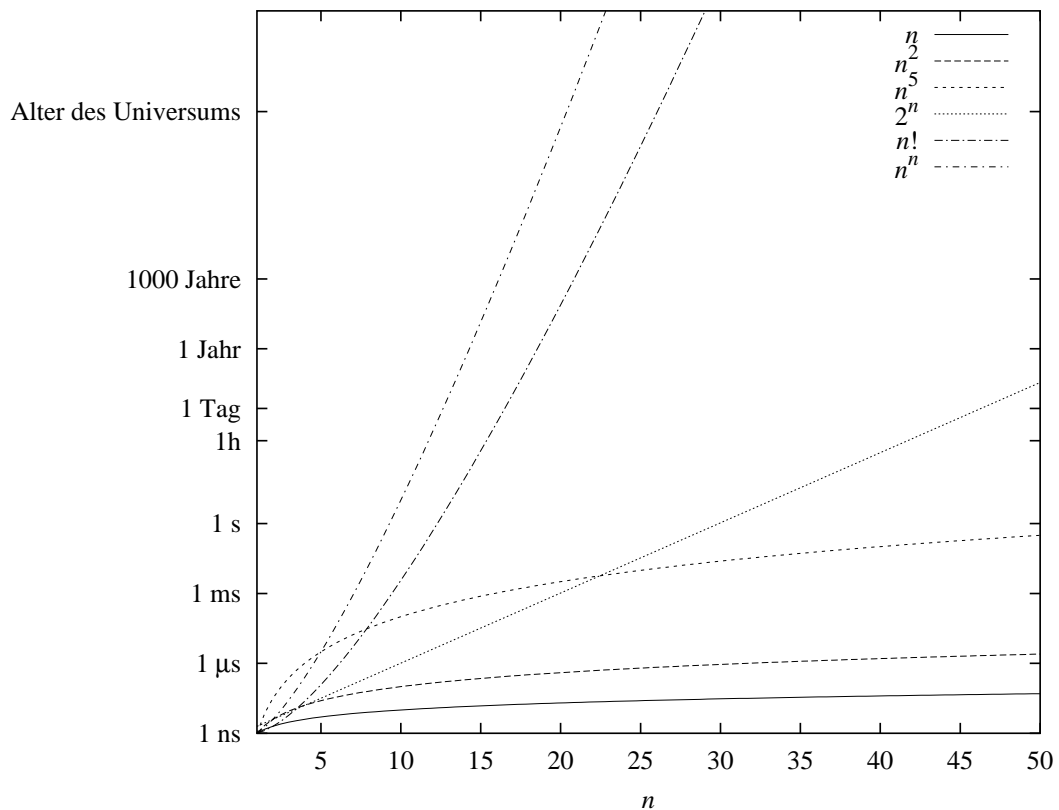


Abbildung 1.2: Wachstumsverhalten einiger wichtiger Komplexitätsklassen. Dargestellt ist der Laufzeitbedarf hypothetischer Algorithmen mit den angegebenen Komplexitäten, wobei angenommen wird, daß der Rechner für jede elementare Operation 1 ns (1 ns = 10^{-9} Sekunden) benötigt. (Die Zeitachse ist logarithmisch.)

Komplexität praktikabel sind. Laufzeiten proportional 2^n oder $n!$ nehmen schon bei moderaten Problemgrößen astronomische Werte an.

1.2.3 O-Notation für asymptotische Aussagen

Beschreibt eine Funktion die Laufzeit eines Programms, so ist es oft ausreichend, nur ihr asymptotisches Verhalten zu untersuchen. Konstante Faktoren werden vernachlässigt, und die Betrachtung beschränkt sich auf eine „einfache“ Funktion. Zugleich sollen aber möglichst enge Schranken gefunden werden.

Um asymptotische Aussagen mathematisch konkret zu fassen, definiert man die folgenden Mengen (sog. *O-Notation*):

Definition: Sei $f : \mathbb{N} \rightarrow \mathbb{R}^+$ eine Funktion, dann ist

$$\begin{aligned} O(f) &:= \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \\ \Omega(f) &:= \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : c \cdot f(n) \leq g(n)\} \\ \Theta(f) &:= \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : \frac{1}{c} \cdot f(n) \leq g(n) \leq c \cdot f(n)\} \\ o(f) &:= \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : g(n) < cf(n)\} \\ \omega(f) &:= \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : cf(n) < g(n)\} \end{aligned}$$

Übliche Sprechweise:

- $g \in O(f)$: f ist *obere Schranke* von g .
 g wächst höchstens so schnell wie f .
 $g \in \Omega(f)$: f ist *untere Schranke* von g .
 g wächst mindestens so schnell wie f .
 $g \in \Theta(f)$: f ist die *Wachstumsrate* von g .
 g wächst wie f

Statt $g \in O(f)$ schreibt man oft (formal nicht ganz korrekt) $g(n) \in O(f(n))$, um die Abhängigkeit von n deutlich zu machen und die Definition zusätzlicher „Hilfsfunktionen“ zu vermeiden. So steht beispielsweise „ $O(n^2)$ “ für „ $O(f)$ mit $f(n) = n^2$ “. Auch die Schreibweise $g = O(f)$ ist üblich.

Allgemeine Aussagen und Rechenregeln

Für das Rechnen mit den oben definierten Mengen gelten einige einfache Regeln. Seien f und g Funktionen $\mathbb{N} \rightarrow \mathbb{R}^+$, dann gilt:

1. Linearität:
Falls $g(n) = \alpha f(n) + \beta$ mit $\alpha, \beta \in \mathbb{R}^+$ und $f \in \Omega(1)$, dann gilt: $g \in O(f)$

2. Addition:

$$f + g \in O(\max\{f, g\}) = \begin{cases} O(g) & \text{falls } f \in O(g) \\ O(f) & \text{falls } g \in O(f) \end{cases}$$

3. Multiplikation:

$$a \in O(f) \wedge b \in O(g) \Rightarrow a \cdot b \in O(f \cdot g)$$

4. Falls der Grenzwert $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)}$ existiert, so ist $g \in O(f)$. (Der Umkehrschluß gilt nicht!)

5. Es gilt:

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

Dabei sind Addition, Multiplikation, Maximumbildung von zwei Funktionen *bildweise* zu verstehen, also z.B. $(f + g)(n) = f(n) + g(n)$.

Beweis zu 1: Wegen $f \in \Omega(1)$ gibt es $c' > 0$ und $n' > 0$, mit $f(n) \geq c' \cdot 1 \quad \forall n \geq n'$.
Wähle $c = \alpha + \frac{\beta}{c'}$ und $n_0 = n'$, dann gilt:

$$g(n) := \alpha \cdot f(n) + \beta \leq c \cdot f(n) \quad \forall n \geq n_0$$

D.h. $g \in O(f)$.

Beispiele:

a) Sei $f(n) = 7n + 3$. Dann gilt:

$$7n + 3 \leq c \cdot n \quad \forall n \geq n_0 \text{ mit } c = 8 \text{ und } n_0 = 3$$

und damit $f(n) \in O(n)$

b) Sei $f(n) = \sum_{k=0}^K a_k n^k$ mit $a_k > 0$. Dann ist $f \in O(n^K)$, denn

$$f(n) \leq cn^K \quad \forall n \geq n_0 \text{ mit } c = \sum_{k=0}^K a_k \text{ und } n_0 = 1$$

c) Sei $T(n) = 3^n$. Dann gilt: $T(n) \notin O(2^n)$, denn es gibt kein Paar $c > 0$, $n_0 > 0$, so daß

$$3^n \leq c \cdot 2^n \quad \forall n \geq n_0$$

d) Seien $T_1(n) \in O(n^2)$, $T_2(n) \in O(n^3)$ und $T_3(n) \in O(n^2 \log n)$, dann gilt:

$$T_1(n) + T_2(n) + T_3(n) \in O(n^3)$$

e) Seien $f(n) = n^2$ und $g(n) = 5n^2 + 100 \log n$. Dann gilt:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{5n^2 + 100 \log n}{n^2} = 5$$

und damit $g \in O(f)$.

1.2.4 Regeln für die Laufzeitanalyse

Elementare Anweisungen sind in $O(1)$.

Es ist wichtig zu beachten, was im gewählten Rechnermodell (z.B. RAM) als elementare Anweisung gilt. Höhere Programmiersprachen stellen oft (scheinbar elementare) Anweisungen zur Verfügung, die vom Compiler in komplexere Anweisungsfolgen umgesetzt werden.

Folgen von Anweisungen: Setzt sich ein Programm aus mehreren Teilen zusammen, die hintereinander ausgeführt werden, so wird das asymptotische Verhalten der Laufzeit allein von dem Teil bestimmt, der den größten Aufwand hat (vgl. *Addition*).

Seien A und B zwei Anweisungen mit Laufzeiten $T_A \in O(f)$ und $T_B \in O(g)$, dann hat die Hintereinanderausführung

$A ; B ;$

die Laufzeit $T = T_A + T_B \in O(\max\{f, g\})$.

Schleifen: Die Gesamtlaufzeit ergibt sich als Summe über die Laufzeiten der einzelnen Durchläufe.

Oft ist die Laufzeit T_K des Schleifenkörpers K unabhängig davon, um den wievielten Durchlauf es sich handelt. In diesem Fall kann man einfach die Laufzeit des Schleifenkörpers T_K mit der Anzahl der Durchläufe $d(n)$ multiplizieren.

Im allgemeinen berechnet man entweder die Summe über die einzelnen Durchläufe explizit oder sucht obere Schranken für T_K und $d(n)$: Wenn $T_K \in O(f)$ und $d \in O(g)$ so ist $T \in O(f \cdot g)$ (vgl. *Multiplikation*).

Bedingte Anweisungen: A und B seien zwei Prozeduren mit Laufzeiten $T_A \in O(f)$ und $T_B \in O(g)$, dann hat die Anweisung

IF *Bedingung* THEN A() ELSE B() END ;

eine Laufzeit in $O(1) + O(g + f)$, falls die Bedingung in konstanter Zeit ausgewertet wird (vgl. *Addition*).

Prozeduraufrufe müssen nach nicht-rekursiven und rekursiven unterschieden werden.

nicht-rekursiv: Jede Prozedur kann separat analysiert werden. Zusätzlich zu den Anweisungen der Prozedur tritt nur noch ein (in der Regel kleiner) konstanter Overhead für Auf- und Abbau des Aktivierungsblocks auf.

rekursiv: Eine einfache Analyse ist nicht möglich. Man muß eine Rekursionsgleichung aufstellen und diese detailliert analysieren (siehe Abschnitt 1.5).

1.2.5 Beispiel: Fibonacci-Zahlen

Wir wollen nun Laufzeitanalysen am Beispiel verschiedener Programme zur Berechnung der Fibonacci-Zahlen durchführen.

Definition der Fibonacci-Zahlen

$$f(n) := \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ f(n-1) + f(n-2) & \text{für } n > 1 \end{cases}$$

Damit ergibt sich die Folge: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Behauptung

1. $f(n) \in O(2^n)$
2. $f(n) \in \Omega(2^{n/2})$

Beweis

- a) $f(n)$ ist positiv: $f(n) > 0 \quad \forall n > 0$
- b) $f(n)$ ist für $n > 2$ streng monoton wachsend: $f(n) < f(n+1)$
- c) Abschätzung nach oben: Für alle $n > 3$ gilt

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ &< 2 \cdot f(n-1) \\ &< 4 \cdot f(n-2) \\ &< \dots \\ &< 2^{n-1} \cdot f(1) = \frac{1}{2} 2^n \end{aligned}$$

Also gilt $f(n) \in O(2^n)$.

d) Abschätzung nach unten: Für alle $n > 3$ gilt

$$\begin{aligned} f(n) &> 2 \cdot f(n-2) \\ &> 4 \cdot f(n-4) \\ &\vdots \\ &> \begin{cases} 2^{\frac{n-1}{2}} \cdot f(1) & , \text{ für } n \text{ ungerade} \\ 2^{\frac{n}{2}-1} \cdot f(2) & , \text{ für } n \text{ gerade} \end{cases} \end{aligned}$$

Also gilt $f(n) \in \Omega(2^{n/2})$.

In Abschnitt 1.5.3 werden wir zeigen, daß $f(n) \in \Theta(c^n)$ mit $c = \frac{1+\sqrt{5}}{2}$.

Zwei Algorithmen zur Berechnung der Fibonacci-Zahlen

Zur Berechnung der Fibonacci-Zahlen betrachten wir einen *rekursiven* und einen *iterativen* Algorithmus.

Naiver, rekursiver Algorithmus

Hier wurde die Definition der Fibonacci-Zahlen einfach übernommen und in ein Programm umgesetzt:

Pseudocode (Modula):

```
PROCEDURE Fibonacci(n : CARDINAL) : CARDINAL =
BEGIN
  IF n <= 1 THEN
    RETURN n ;
  ELSE
    RETURN Fibonacci(n-1) + Fibonacci(n-2) ;
  END ;
END Fibonacci ;
```

Java-Code:

```
int Fibonacci(int n) {
  if(n<=1) {
    return n;
  } else {
    return Fibonacci(n-1) + Fibonacci(n-2);
  }
}
```

Sei $a > 0$ die Dauer für Funktionsaufruf, Verzweigung (IF) und die RETURN-Anweisung. Dann gilt für die Laufzeit $T(n)$ dieses Algorithmus:

$$T(n) = \begin{cases} a & \text{für } n \leq 1 \\ a + T(n-1) + T(n-2) & \text{für } n > 1 \end{cases}$$

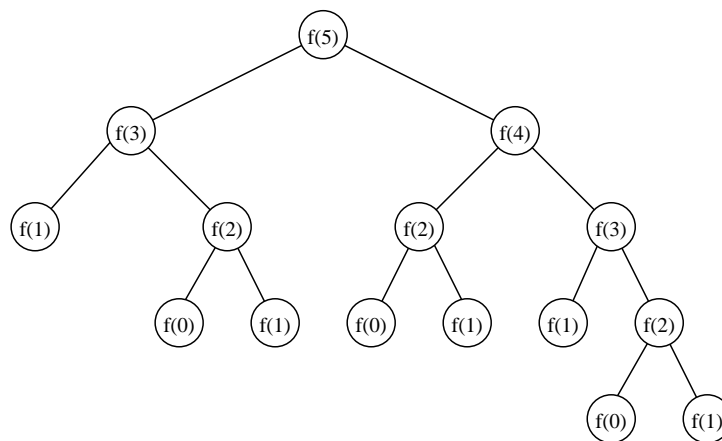


Abbildung 1.3: Baumdarstellung der Aufrufe des rekursiven Fibonacci-Programms für $n=5$

$$\begin{aligned} \text{Offensichtlich ist: } T(n) &= a, a, 3a, 5a, 9a, \dots \\ &= \dots \\ &\geq af(n+1). \end{aligned}$$

$$\text{Daher ist: } T(n) \in \Omega(2^{n/2})$$

Stellt man die rekursiven Aufrufe dieser Prozedur für $f(5)$ als Baum dar (siehe Abb. 1.3), wird klar, warum dieser Algorithmus sehr aufwendig ist: Man betrachte nur die Wiederholungen der Aufrufe von $f(0)$ und $f(1)$.

Iterativer Algorithmus

Der iterative Algorithmus listet die Fibonacci-Zahlen in aufsteigender Reihenfolge auf, indem er jede neue Zahl als Summe ihrer beiden Vorgänger berechnet. Das spart die redundanten Aufrufe zur Berechnung von Fibonacci-Zahlen, die schon einmal berechnet wurden.

```

PROCEDURE Fibonacci(n : CARDINAL) : CARDINAL =
VAR fib_new, fib_old, t : CARDINAL ;
BEGIN
  fib_new := 0 ; fib_old := 1 ;
  FOR i := 1 TO n DO
    t := fib_new ;
    fib_new := fib_new + fib_old ;
    fib_old := t ;
  END ;
  RETURN fib_new ;
END Fibonacci ;

```

Die Laufzeit dieses Algorithmus ist offensichtlich $\Theta(n)$. Das bedeutet eine erhebliche Verbesserung gegenüber der naiven Implementierung.

1.2.6 Beispiel: Fakultät

Wir wollen nun noch am Beispiel der Fakultät zeigen, daß in manchen Fällen auch ein rekursiver Algorithmus, abgesehen vom Overhead für die Funktionsaufrufe und dem Speicherbedarf für die Versorgungsblöcke, keine Nachteile bzgl. Laufzeit-Komplexität hat.

Definition der Fakultät

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n - 1)! & \text{für } n > 0 \end{cases}$$

Für die Laufzeit $T(n)$ des entsprechenden rekursiven Programms ergibt sich mit geeigneten Konstanten $a, b > 0$:

$$\begin{aligned} T(n) &= \begin{cases} a & n = 0 \\ b + T(n - 1) & n > 0 \end{cases} \\ T(n) &= b + T(n - 1) \\ &= 2 \cdot b + T(n - 2) \\ &= \dots \\ &= n \cdot b + T(0) \\ &= n \cdot b + a \\ &\in \Theta(n) \end{aligned}$$

Der Algorithmus hat eine Laufzeit $T(n)$, die linear in n ist, und hat damit (bis auf einen konstanten Faktor) das gleiche Laufzeitverhalten wie eine entsprechende iterative Implementierung.

1.3 Datenstrukturen

1.3.1 Datentypen

Die in Modula-3-ähnlichen Programmiersprachen verfügbaren Datentypen lassen sich in drei Gruppen unterteilen: elementare und zusammengesetzte Typen sowie Zeiger.

Elementare (atomare) Datentypen

- INTEGER : Ganze Zahlen
- CARDINAL : Natürliche Zahlen
- CHARACTER : Zeichen (Buchstaben etc.)
- REAL : Dezimalzahlen
- BOOLEAN : Wahrheitswerte (TRUE oder FALSE)