Informatik 2

Volker Aurich

Heinrich-Heine-Universität Düsseldorf

Sommersemester 2000

Inhaltsverzeichnis

1 Grundlagen der Rechenzeitanalyse **4**

1.1 Einleitung 4

1.2 Benchmarks 6

1.3 Theoretische Rechenzeit von Algorithmen 7

1.4 Die Komplexität eines Problems 11

1.5 Asymptotische Abschätzungen 12

1.6 Speicherbedarfanalyse 16

1.7 Die Rolle der Faktoren bei Wachstumsabschätzungen 16

2 Die Strategie *Divide et Impera* **18**

2.1 Das Grundprinzip von *Divide et Impera* 18

2.2 Typische Gestalt eines *Divide-et-Impera*-Algorithmus 18

2.3 Vorteile der Strategie *Divide et Impera* 19

2.4 Typische *Divide-et-Impera*-Situationen 20

2.5 Asymptotische Rekursionsgleichungen 21

2.6 Beispiel: Binäre Suche 23

2.7 Beispiel: Berechnung von $\lceil \sqrt{n} \rceil$ für $n \in \mathbb{N}$ 26

2.8 Beispiel: Die schnelle Fouriertransformation 28

2.9 Beispiel: Schnelle Multiplikation von n-Bit-Zahlen 31

2.10 Beispiel: Schnelle Matrixmultiplikation 35

2.11 Beispiel: Minimalabstand 37

3 Sortieren **40**

3.1 Grundbegriffe 40

3.2 Überblick über Sortierverfahren 41

3.3 Einfache Sortierverfahren . . . 42
3.4 Merge Sort . . . 44
3.5 Quick Sort . . . 45
3.6 Heap Sort . . . 51
3.7 Zusammenfassender Überblick . . . 55
3.8 Sortieren durch Verteilen in Fächer, Bin Sort . . . 56
3.9 Eine untere Laufzeitschranke für vergleichsbasierte Sortierverfahren . . . 59

4 Berechenbarkeit **60**
4.1 Einige Entscheidungsproblem . . . 60
4.2 Gödelisierung . . . 61
4.3 Algorithmisch unentscheidbare Probleme . . . 62
4.4 Turingmaschinen . . . 62
4.5 Andere Maschinenmodelle . . . 64
4.6 Rekursive Funktionen . . . 64
4.7 Entscheidbarkeit . . . 66
4.8 Bemerkungen . . . 66

5 NP-Vollständigkeit **68**
5.1 Effiziente Algorithmen . . . 68
5.2 Nichtdeterministische Algorithmen . . . 69
5.3 Polynomialzeitbeschränkte Reduktionen . . . 70
5.4 NP-Vollständigkeit . . . 71

A Mathematische Ergänzungen **72**
A.1 Einige Grundlagen . . . 73
A.2 Der Kalkül mit den Symbolen O, Θ, Ω . . . 75

B Lineare Rekursionsgleichungen **82**
B.1 Einführung . . . 82
B.2 Homogene lineare Rekursionsgleichungen . . . 83
B.3 Beispiele . . . 86
B.4 Inhomogene lineare Rekursionsgleichungen . . . 89
B.5 Beispiele . . . 92

C *Divide-et-Impera*-Rekursionsgleichungen **95**

C.1 Lösungen auf $D := \{b^k n_0 : k \in \mathbb{N}\}$ 95
C.2 Lösungen auf ganz $\mathbb{N}$ 98

D Allgemeine Rekursionsgleichungen 102
D.1 Erraten einer Lösung 102
D.2 Iteration 102
D.3 Substitution 103
D.4 Rekursionsgleichungen mit voller Geschichte 103

Kapitel 1

Grundlagen der Rechenzeitanalyse von Algorithmen

1.1 Einleitung

Bei praktischen Anwendungen treten folgende Fragen auf:

- Wie lange benötigt ein Programm zur Lösung eines Problems? Wieviel Speicher benötigt es?
- Wenn es zur Lösung eines Problems mehrere Algorithmen gibt, welcher ist der günstigste (schnellste, speichersparsamste)?

Die Antworten hängen stark von den konkreten Umständen der jeweiligen Anwendung ab wie z.B. der typischen Größe des Problems, den verwendeten Rechnern oder Compilern. Trotzdem (oder gerade deshalb?) ist es sinnvoll, die Laufzeit von Algorithmen theoretisch und möglichst unabhängig von Maschinen und Programmiersprachen abzuschätzen. Allerdings macht es natürlich einen Unterschied, ob man Algorithmen für einen Parallelrechner oder für einen Rechner mit nur einem Prozessor entwirft. Wir werden zunächst nur serielle Algorithmen betrachten d.h. Algorithmen, deren Schritte strikt nacheinander (durch einen einzigen Prozessor) ausgeführt werden.

Um die Idee eines Algorithmus zu vermitteln, ist seine Realisierung in einer konkreten Programmiersprache meist nicht sehr günstig, weil so ein Programm zu unübersichtlich ist. Stattdessen verwendet man lieber einen Pseudocode, der die üblichen Schleifenkonstrukte und Fallunterscheidungen imperativer Sprachen, aber nicht viel mehr enthält. Es gibt viele Spielarten von Pseudocodes; wir orientieren uns an der Syntax von C. Allerdings werden wir abweichend von C auch Arrays verwenden, deren Indexbereich ein beliebiges Intervall $[m \dots n] := \{k \in \mathbb{Z} \colon m \leq k \leq n\}$ in $\mathbb{Z}$ ist. So ein Array sei mit $\mathit{arrayname}[m \dots n]$ bezeichnet.

Wir werden die grundlegenden Begriffe der Rechenzeitanalyse an den folgenden drei Beispielalgorithmen illustrieren.

1.1.1 Beispiel: Sortieren durch Auswahl, Selection Sort

Aufgabe: Sortiere jede vorgelegte Folge $a_1, \ldots, a_n$ ganzer Zahlen d.h. permutiere sie so, daß eine monoton wachsende Folge entsteht.

Dabei interessiert die Permutation nicht, sondern nur die entstehende monotone Folge. Ein einfacher Sortieralgorithmus ist folgender.

```
SelectionSort(A[1...n]) {                              | A enthält die Folge a_1,...,a_n
    for (i = 1; i ≤ n; i = i + 1) {            (1)   |
        j = i;                                  (2)   |
        for (k = i + 1; k ≤ n; k = k + 1)      (3)   | finde das kleinste j ∈ [i...n],
            if (A[k] < A[j])   j = k;           (4)   | so daß A[j] das kleinste
                                                      | Element von A[i...n] ist

        hilf = A[i];                            (5)   | vertausche A[i] und A[j]
        A[i] = A[j];                            (6)   |
        A[j] = hilf;                            (7)   |
    }                                                 |
}                                                     |
```

1.1.2 Beispiel: Sortieren durch Verschmelzen, Merge Sort

Aufgabe: wie oben

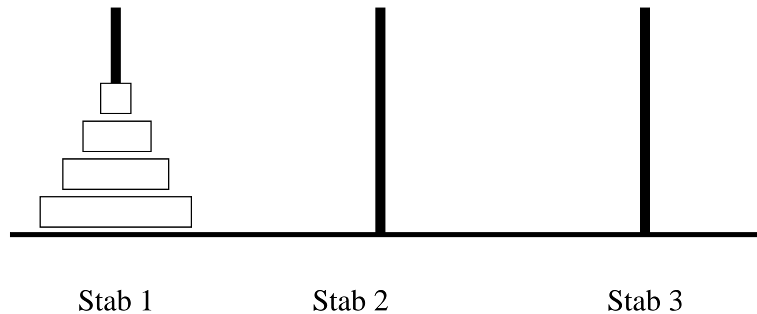
Lösungsidee: Spalte $a_1, \ldots, a_n$ in zwei Hälften $a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ und $a_{\lfloor \frac{n}{2} \rfloor + 1} \ldots a_n$ auf, sortiere diese beiden Teilfolgen und mische sie zu einer sortierten Folge zusammen. Mache daraus einen rekursiven Algorithmus, indem zum Sortieren jeder Teilfolge wieder dieselbe Strategie verwendet, außer wenn sie die Länge 1 hat und somit nicht sortiert werden muß.

```
MergeSort(A[m...k]) {
    if(m < k) {
        n = k − m + 1;
        l = ⌊n/2⌋ − 1;
        MergeSort(A[m...m + l]);
        MergeSort(A[m + l + 1...k]);
        verschmelze A[m...m + l] und A[m + l + 1...k]
        zu dem sortierten Array A[m...k];
    }
}
```

Bemerkung: Die Idee von MergeSort ist ein Beispiel für eine Strategie, die *divide et impera* oder *teile und herrsche* oder *divide and conquer* genannt wird. Sie besteht darin, zu zeigen, daß ein Problem in kleinere, gleichartige Probleme zerteilt werden kann, aus deren Lösungen man die Lösung des Ursprungsproblems gewinnen kann, und daß man durch wiederholte Zerteilung schließlich bei einfach zu lösenden Problemen landet. Auf diese Weise erhält man einen rekursiven Lösungsalgorithmus.

1.1.3 Beispiel: Die Türme von Hanoi (TvH)

Gegeben sind n Scheiben mit Mittelloch und paarweise verschiedenen Radien, die wie abgebildet längs des Stabes 1 zu einem Turm gestapelt sind.



Stab 1 Stab 2 Stab 3

Aufgabe: Lege alle Scheiben auf Stab 3 um unter Beachtung folgender Regeln:

(1) Eine Scheibe darf nur auf einen Stab gelegt werden.
(2) Es darf stets nur eine Scheibe auf einmal versetzt werden.
(3) Es darf keine Scheibe auf eine kleineren Durchmessers gelegt werden.

Lösungsstrategie: Der Fall $n = 1$ ist trivial. Für $n > 1$ gehe rekursiv vor d.h. benutze vollständige Induktion nach n:

Induktionsvoraussetzung: Man kann $n-1$ Scheiben unter Verwendung eines Hilfsstabes regelgerecht auf einen anderen transportieren.

Induktionsschritt: Transportiere die oberen $n-1$ Scheiben unter Verwendung von Stab 3 auf Stab 2; das ist nach Induktionsvoraussetzung möglich. Dann setze die auf Stab 1 verbliebene (größte) Scheibe auf Stab 3. Anschließend transportiere die $n-1$ Scheiben von Stab 2 nach Stab 3 unter Verwendung von Stab 1 als Hilfsstab; das ist wiederum nach Induktionsvoraussetzung möglich. Nun befindet sich der Turm auf Stab 3.

Wir beschreiben den Algorithmus in Pseudocode als Funktion, deren Argumente die Anzahl n der Scheiben und die Nummern des Start-, Ziel- und Hilfsstabes sind.

```
TvH(n, start, ziel, hilf) {
    if(n == 1) lege die eine Scheibe von Stab start auf den Stab ziel;
    else {
        TvH(n − 1, start, hilf, ziel);
        lege die Scheibe von Stab start auf den Stab ziel;
        TvH(n − 1, hilf, ziel, start);
    }
}
```

1.2 Benchmarks

Um zwei Programme, die das gleiche Problem lösen, bezüglich ihrer Rechenzeit zu vergleichen, wählt man oft einige typische Eingabedaten (benchmarks genannnt) und vergleicht die Rechenzeiten für diese Eingaben. (Das Wort *benchmark* bezeichnet ur-

sprünglich einen geodätischen Meßpunkt.)

Dabei treten folgende Probleme auf:

1. Welche Eingabedaten sind typisch? Das hängt stark von der jeweiligen Anwendung ab! So wird z.B. beim Sortieren die Anzahl der zu sortierenden Daten eine Rolle spielen, aber vielleicht auch, ob sie teilweise bereits vorsortiert sind.

2. Die Rechenzeiten sind abhängig von dem verwendeten Compiler. Und zwar manchmal sogar sehr stark!

3. Die Rechenzeiten sind von der Hardwarearchitektur und vom Betriebssystem abhängig. Deshalb werden Benchmark-Messungen ja auch zum Vergleich der Geschwindigkeit von Rechnern eingesetzt.

Folgerung: Benchmark-Vergleiche von Programmen sind sinnvoll, wenn Anwendung, Rechner und Compiler festliegen. Sie können dann auch dazu dienen, ein Programm durch gezielte Änderungen des Quellcodes zu beschleunigen. Es gibt Software-Werkzeuge (profiler wie z.B. gprof), mit denen man feststellen kann, welchen Anteil jeder Programmteil an der Gesamtrechenzeit hat. Damit kann man dann versuchen, den Code der sogenannten hot spots d.h. der Teile, die stark zur Gesamtrechenzeit beitragen, zu verbessern.

Um den prinzipiellen Rechenaufwand von Algorithmen zu vergleichen, sind Benchmark-Messungen nicht geeignet.

1.3 Theoretische Rechenzeit von Algorithmen

Man definiert für einen Algorithmus A als Rechenzeit

T_{A}= Anzahl der elementaren Rechenschritte, die der Algorithmus ausführt

Der Begriff des elementaren Rechenschritts kann unterschiedlich festgelegt werden; so könnte man z.B. Programm-Statements oder auch Assemblerbefehle wählen. Oft zählt man nur Schritte einer gewissen Art. Bei Sortieralgorithmen z.B. werden meist nur die Anweisungen gezählt, die zwei Daten vergleichen.

In den meisten Fällen hängt die Rechenzeit von den Werten einiger Parameter des Algorithmus ab, z.B. beim Sortieren von der zu sortierenden Datenfolge. Man faßt sie als Eingabeparameter des Algorithmus A auf und sagt, man habe für jede Wahl der Parameter ein Problem p (oder eine Fragestellung). $T_{\mathsf{A}}(p)$ bezeichne die Rechenzeit des Algorithmus A für das Problem p.

Oft ordnet man jeder solchen Fragestellung p eine natürliche Zahl $\gamma(p)$ (oder ein Tupel) zu, die man **Größe** von p nennt. Die Definition von γ ist willkürlich, ergibt sich aber meist auf natürliche Weise. So wählt man als Größe eines Sortierproblems die Anzahl der zu sortierenden Daten. Beim Problen der Türme von Hanoi wird man die Anzahl n der Scheiben wählen.

Probleme derselben Größe n können durchaus unterschiedliche Rechenzeiten haben. Deshalb definiert man

als Worst-Case-Rechenzeit	$T_{\mathsf{A}}^{worst}(n) = \max\{T_{\mathsf{A}}(p) : \gamma(p) = n\}$
als Best-Case-Rechenzeit	$T_{\mathsf{A}}^{best}(n) = \min\{T_{\mathsf{A}}(p) : \gamma(p) = n\}$
als Average-Case-Rechenzeit	$T_{\mathsf{A}}^{average}(n) =$ Mittelwert aller $T_{\mathsf{A}}(p)$ mit $\gamma(p) = n$

Als Mittelwert wird meist das arithmetische Mittel verwendet; falls jedoch bei einer Anwendung die einzelnen Probleme unterschiedlich häufig auftreten, ist es sinnvoll, einen gewichteten Mittelwert zu wählen.

1.3.1 Beispiel: Die Rechenzeit von Sortieralgorithmen hängt i.a. von der Länge n der Datenfolge $a_1, \ldots, a_n$ und der Anordnung der $a_1, \ldots, a_n$ ab, aber nicht von den Werten der a_i. Daher definiert man meist als Average-Case-Rechenzeit $T_{\mathsf{A}}^{average}(n)$ den arithmetischen Mittelwert der Rechenzeiten $T_{\mathsf{A}}(b_1, \ldots, b_n)$, wobei $(b_1, \ldots, b_n)$ alle möglichen $n!$ Anordnungen einer Folge $(a_1, \ldots, a_n)$ von Daten durchläuft. Oft werden überdies noch die a_i als paarweise verschieden vorausgesetzt.

1.3.2 Bemerkungen zur Definition der Eingabe- oder Problemgröße

Man definiert γ i.a. so, daß das Urbild $\gamma^{-1}(n)$ für jedes $n \in \mathbb{N}$ endlich ist. Dann gibt es keine prinzipiellen Schwierigkeiten, die Average-Case-Rechenzeit als arithmetischen Mittelwert zu definieren.

Diese Endlichkeit von γ ist in praktischen Situationen meist auf natürliche Weise erfüllt. Denn man geht davon aus, daß jeder Eingabeparameter einen (einfachen) Datentyp hat, der in praktischen Implementierungen einen endlichen Wertebereich hat (z.B. `int`). Wenn man als Problemgröße die Anzahl der Eingabeparameter wählt, so gibt es zu gegebener Problemgröße nur endlich viele Wertekombinationen für die Eingabe. In speziellen Fällen, in denen ein Eingabeparameter einen komplexen Datentyp hat, dessen Speicherbedarf nicht von vornherein beschränkt ist (z.B. beliebig große Dezimalzahlen), wird man die Länge des belegten Speichers (Anzahl der Bytes oder Bits) zur Problemgröße hinzunehmen. Es kann dann auch sinnvoll sein, die Problemgröße durch ein Tupel natürlicher Zahlen zu beschreiben (z.B. bei der Multiplikation beliebig großer Zahlen).

1.3.3 Die Rechenzeit von TvH

Es sei $T(n)$ die Anzahl der Scheibenbewegungen, die der Algorithmus zur Lösung des Problems mit n Scheiben durchführt. Man sieht unmittelbar, daß gilt

$$T(1) = 1 \quad \text{und} \quad T(n) = 2 \cdot T(n-1) + 1$$

Diese Rekursionsgleichung (mit Anfangsbedingung) hat die Lösung $T(n) = 2^n - 1$.

Beweis: durch vollständige Induktion nach n.
Induktionsanfang: $T(1) = 2^1 - 1 = 1$.
Induktionsschluß von $n-1$ nach n: $T(n) = 2^n - 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2 \cdot T(n-1) + 1$.
q.e.d.

Zu jeder Problemgröße n gibt es nur ein Problem dieser Größe. Daher fallen Worst-Case, Best-Case und Average-Case zusammen.

1.3.4 Die Rechenzeit von SelectionSort

Wir bestimmen zunächst nur, wie oft zwei der Eingabedaten $a_1, \ldots, a_n$ verglichen wer-

den oder ein Datum zugewiesen wird. In der inneren **for**-Schleife (Zeile (4)) kommen $n-i$ Vergleiche und keine Zuweisung vor. Die äußere **for**-Schleife wird n-mal durchlaufen; jedesmal werden 3 Zuweisungen durchgeführt. Also insgesamt $\sum_{i=1}^{n}(n-i) = \sum_{j=0}^{n-1} j = \frac{n(n-1)}{2}$ Vergleiche und $3n$ Zuweisungen.

Etwas aufwendiger ist es, alle Operationen zu zählen.

		Summe
Zeile (1):	1 Zuweisung, $n+1$ Vergleiche,	
	n Inkrementoperationen	$2n+2$
Zeile (2):	n Zuweisungen	n
Zeile (3):	n Zuweisungen,	
	$\sum_{i=1}^{n-1}(n-i+1) = \frac{n(n-1)}{2} + n - 1$ Vergleiche,	
	$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$ Inkrementoperationen	$n(n-1)+2n-1$
Zeile (4):	$\sum_{i=1}^{n-1}(n-i)$ **if**-Anweisungen,	$n(n-1)$
	$\sum_{i=1}^{n-1}(n-i)$ Vergleiche,	bis
	0 bis $\sum_{i=1}^{n-1}(n-i)$ Zuweisungen	$\frac{3}{2}n(n-1)$
Zeilen (5)-(7):	3n Zuweisungen	$3n$

Insgesamt sind dies $2n(n-1)+8n+1$ bis $\frac{5}{2}n(n-1)+8n+1$ Operationen. Der Worst-Case-Aufwand $T^{worst}(n)$ wächst also quadratisch.

Wir machen hier die typische Erfahrung, daß die exakte Bestimmung des Aufwandes $T^{worst}(n)$ meist schwieriger als eine Abschätzung des asymptotischen Wachstums ist.

1.3.5 Die Rechenzeit von MergeSort

Aufgrund der rekursiven Struktur des Algorithmus ergibt sich für die Rechenzeit folgende Rekursionsgleichung

$$\begin{aligned}\text{Zeit zum Sortieren von } A[m \ldots k] = \; & \text{Zeit zum Sortieren von } A[m \ldots m+l] \\ & + \text{Zeit zum Sortieren von } A[m+l+1 \ldots k] \\ & + \text{Zeit zum Verschmelzen der beiden Teilarrays}\end{aligned}$$

Wir wollen im Folgenden nur die Anzahl der Vergleichsoperationen zwischen Eingabedaten bestimmen. Sei also $T(n)$ die Anzahl der Vergleiche, die MergeSort im schlechtesten Fall benötigt, um eine Folge der Länge n zu sortieren.

Mit $n = k - m + 1$ gilt dann

1.3.5.1 $\quad T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + M(n), \quad T(1) = 0,$

wobei $M(n)$ die Anzahl der Vergleiche ist, die zum Mischen der beiden bereits sortierten Arrays $A[m \ldots m+l]$ und $A[m+l+1 \ldots k]$ benötigt werden.

Wir geben zunächst einen Verschmelzungsalgorithmus an.

```
Merge( A[1...n], A1[1...n1], A2[1...n2] ) {
    i = j = k = 1;
    while ( (i ≤ n1) und (j ≤ n2) ) {
        if (A1[i] < A2[j]) { A[k] = A1[i]; i = i + 1; k = k + 1; }
```

```
        else { A[k] = A2[i]; j = j + 1; k = k + 1; }
    }
    while(i ≤ n1) { A[k] = A1[i]; i = i + 1; k = k + 1; }
    while(j ≤ n2) { A[k] = A2[i]; j = j + 1; k = k + 1; }
}
```

Dieser Algorithmus benötigt im schlechtesten Fall $n_1 + n_2 - 1$ Vergleiche zwischen den Elementen von A_1 und A_2. Innerhalb von MergeSort ist $n_1 = \lfloor \frac{n}{2} \rfloor$ und $n_2 = \lceil \frac{n}{2} \rceil$, also werden im schlechtesten Fall $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil - 1 = n - 1$ Vergleiche ausgeführt. Wir zeigen, daß es nicht besser geht.

1.3.5.2 Lemma Jeder Algorithmus, der zwei beliebige sortierte Folgen $x_1 \leq x_2 \leq \ldots \leq x_{\lfloor \frac{n}{2} \rfloor}$ und $y_1 \leq y_2 \leq \ldots \leq y_{\lceil \frac{n}{2} \rceil}$ zu einer sortierten Folge $z_1 \leq z_2 \leq \ldots \leq z_n$ mischt, benötigt dazu im schlechtesten Fall $n - 1$ Vergleichsoperationen.

Beweis: Wähle die x_i und y_j so, daß $y_1 < x_1 < y_2 < x_2 < \ldots < x_{\lfloor \frac{n}{2} \rfloor}$ ($< y_{\lceil \frac{n}{2} \rceil}$, falls n ungerade). Angenommen, der Algorithmus führt weniger als $n - 1$ Vergleiche durch. Dann kann er nicht alle Paare (y_i, x_i) und (x_i, y_{i+1}) vergleichen; denn es gibt $n - 1$ solcher Paare. OBdA habe er y_i und x_i nicht verglichen. Wir tauschen x_i und y_i in den beiden Anfangsfolgen aus. Beide bleiben sortiert! Der Algorithmus führt dieselben Vergleichsoperationen wie bei den ursprünglichen Folgen aus und liefert deshalb dieselbe z-Folge; sie ist aber nicht sortiert, weil jetzt $y_i > x_i$ ist. Dies ist ein Widerspruch! q.e.d.

1.3.5.3 Folgerung Es gilt $M(n) = n - 1$ und folglich

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + n - 1, \quad T(1) = 0$$

1.3.5.4 Lemma Die Rekursionsgleichung

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + n - 1, \quad T(1) = 0$$

hat die Lösung $T(n) = n \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$.

Beweis: Es gilt $T(1) = 0$. Für $n > 1$ schreibe man $n = 2^k + a$ mit $0 \leq a < 2^k$ und $k \in \mathbb{N}$. Es ist $\frac{n}{2} = 2^{k-1} + \frac{a}{2}$, also $\lfloor \frac{n}{2} \rfloor = 2^{k-1} + \lfloor \frac{a}{2} \rfloor$ und $\lceil \frac{n}{2} \rceil = 2^{k-1} + \lceil \frac{a}{2} \rceil$.

Für $a = 1$ gilt:

$$\left\lfloor \frac{n}{2} \right\rfloor = 2^{k-1} \text{ und } \left\lceil \frac{n}{2} \right\rceil = 2^{k-1} + 1$$

$$\log_2 \left\lfloor \frac{n}{2} \right\rfloor = k - 1 \text{ und } k - 1 < \log_2 \left\lceil \frac{n}{2} \right\rceil \leq k$$

$$\left\lceil \log_2 \left\lfloor \frac{n}{2} \right\rfloor \right\rceil = k - 1 \text{ und } \left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil = k$$

also

$$\left\lceil \log_2 \left\lfloor \frac{n}{2} \right\rfloor \right\rceil = \left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil - 1$$

Für $a \neq 1$ gilt: Ist $a = 0$, so ist $\lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil = 2^{k-1}$, also $\lceil \log_2 \lfloor \frac{n}{2} \rfloor \rceil = \lceil \log_2 \lceil \frac{n}{2} \rceil \rceil = k - 1$. Ist $a \geq 2$, so gilt:

$$2^{k-1} < \left\lfloor \frac{n}{2} \right\rfloor < 2^k \text{ und } 2^{k-1} < \left\lceil \frac{n}{2} \right\rceil \leq 2^k$$

$$k - 1 < \log_2 \left\lfloor \frac{n}{2} \right\rfloor < k \text{ und } k - 1 < \log_2 \left\lceil \frac{n}{2} \right\rceil \leq k$$

$$\left\lceil \log_2 \left\lfloor \frac{n}{2} \right\rfloor \right\rceil = k \text{ und } \left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil = k$$

Außerdem folgt für jedes $n \in \mathbb{N}, n > 1$,

$$\left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil = \lceil \log_2 n \rceil - 1$$

denn für $a \neq 0$ gilt $\lceil \log_2 \lceil \frac{n}{2} \rceil \rceil = k = k + 1 - 1 = \lceil \log_2 n \rceil - 1$, und für $a = 0$ gilt $\lceil \log_2 \lceil \frac{n}{2} \rceil \rceil = k - 1 = \lceil \log_2 n \rceil - 1$.

1.Fall: $a \neq 1$. Es gilt $\lceil \log_2 \lfloor \frac{n}{2} \rfloor \rceil = \lceil \log_2 \lceil \frac{n}{2} \rceil \rceil = \lceil \log_2 n \rceil - 1$.

$$\begin{aligned} T(n) &= n \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1 \\ &= (\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil)(\lceil \log_2 n \rceil - 1) - 2 \cdot 2^{(\lceil \log_2 n \rceil - 1)} + n + 1 \\ &= \left\lfloor \frac{n}{2} \right\rfloor \left\lceil \log_2 \left\lfloor \frac{n}{2} \right\rfloor \right\rceil - 2^{\lceil \log_2 \lfloor \frac{n}{2} \rfloor \rceil} + 1 + \left\lceil \frac{n}{2} \right\rceil \left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil - 2^{\lceil \log_2 \lceil \frac{n}{2} \rceil \rceil} + 1 + n - 1 \\ &= T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + n - 1 \end{aligned}$$

2. Fall: $a = 1$. Es gilt $\lceil \log_2 \lfloor \frac{n}{2} \rfloor \rceil = \lceil \log_2 \lceil \frac{n}{2} \rceil \rceil - 1 = \lceil \log_2 n \rceil - 2$.

$$\begin{aligned} T(n) &= n \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1 \\ &= n \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 2^{(\lceil \log_2 n \rceil - 2)} - \left\lfloor \frac{n}{2} \right\rfloor + 1 \\ &= \left\lfloor \frac{n}{2} \right\rfloor (\lceil \log_2 n \rceil - 1) + \left\lceil \frac{n}{2} \right\rceil \lceil \log_2 n \rceil - \frac{3}{4} \cdot 2^{\lceil \log_2 n \rceil} + 1 \\ &= \left\lfloor \frac{n}{2} \right\rfloor (\lceil \log_2 n \rceil - 2) + \left\lceil \frac{n}{2} \right\rceil (\lceil \log_2 n \rceil - 1) - 2^{(\lceil \log_2 n \rceil - 2)} - 2^{(\lceil \log_2 n \rceil - 1)} + n + 1 \\ &= \left\lfloor \frac{n}{2} \right\rfloor \left\lceil \log_2 \left\lfloor \frac{n}{2} \right\rfloor \right\rceil - 2^{\lceil \log_2 \lfloor \frac{n}{2} \rfloor \rceil} + 1 + \left\lceil \frac{n}{2} \right\rceil \left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil - 2^{\lceil \log_2 \lceil \frac{n}{2} \rceil \rceil} + 1 + n - 1 \\ &= T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + n - 1 \end{aligned}$$

q.e.d.

1.4 Die Komplexität eines Problems

Als Maß für die Schwierigkeit eines Problems der Größe n kann man

$$K(n) = \min\{T_{\mathsf{A}}^{worst}(n) \colon \mathsf{A} \text{ Algorithmus, der das Problem löst } \}$$

verwenden. $K(n)$ ist der Mindestaufwand, den ein Algorithmus im ungünstigsten Fall zur Lösung des Problems der Größe n braucht, also die beste untere Schranke für das Worst-Case-Verhalten aller Lösungalgorithmen.

Es gilt stets $K(n) \leq T_{\mathsf{A}}^{worst}(n)$ für jeden Lösungsalgorithmus A .

In manchen Fällen kennt man K; z.B. haben wir in Lemma 1.3.5.2 gezeigt, daß für das Mischen zweier sortierter Listen $K(n) = n - 1$ gilt (wobei nur die Vergleiche gezählt werden). Oft kennt man K jedoch nicht genau, so z.B. für die Multiplikation langer ganzer Zahlen oder für die Multiplikation von Matrizen.

1.5 Asymptotische Abschätzungen des Wachstums der Rechenzeit

Wie wir in 1.3.5.4 gesehen haben, kann es schon für einfache Algorithmen sehr mühsam sein, ihre Rechenzeit exakt zu bestimmen. Dazu kommt, daß nicht von vorn herein klar ist, welche Rechenschritte tatsächlich gezählt werden sollen oder mit welchem Gewicht unterschiedliche Rechenschritte in die gesamte Rechenzeit eingehen sollen; denn in praktischen Anwendungen hängt eine realistische Gewichtung stark von der verwendeten Hardware ab.

Folgende Beobachtungen weisen einen Ausweg:
1. Für kleine Problemgrößen n unterscheiden sich die Rechenzeiten unterschiedlicher Algorithmen oft nicht allzu sehr, und es ist oft einfach zu sehen, welcher der schnellere ist.
2. Für größe Problemgrößen n ist letztendlich derjenige Algorithmus schneller, dessen Rechenzeit langsamer mit n wächst.
3. Die Gewichtung der unterschiedlichen Rechenschritte geht nur unwesentlich in das asymptotische Wachstum der Rechenzeit für $n \to \infty$ ein (das wird im Folgenden präzisiert).

Deshalb untersucht man oft nur, welcher asymptotischen Wachstumsklasse die Rechenzeit eines Algorithmus angehört. Zur Definition dieser Klassen verwendet man den Kalkül mit den Symbolen O, Θ, Ω (siehe Anhang).

Unsere bisherigen Überlegungen zeigen, daß die Worst-Case-Rechenzeiten von TvH in $\Theta(2^n)$, die von SelectionSort in $\Theta(n^2)$ und die von MergeSort in $\Theta(n \log n)$ liegt (beachte $O(2^{\lceil \log_2 n \rceil}) = O(2^{\log_2 n}) = O(n) \subset O(n \log n)$). Um diese Aussagen zu erhalten, wäre es nicht nötig gewesen, die Rechenzeit auf mühsame Weise exakt zu bestimmen; wir hätten sie nur bis auf Konstanten und Terme niedrigerer Ordnung abschätzen müssen. Solche Abschätzungen können von Algorithmus zu Algorithmus recht unterschiedlich verlaufen; andererseits gibt es aber typische Strukturen, die in vielen Algorithmen auftreten und ähnliche Abschätzungen erlauben, insbesondere Schleifenkonstrukte und Rekursionen.

Wir werden daher jetzt typische Wachstumsabschätzungen für solche Strukturen untersuchen. Rekursionen führen auf natürliche Weise auf Rekursionsgleichungen. Leider gibt es für deren Lösung keine geschlossene Theorie. Wir werden aber einige Arten von Rekursionsgleichungen im Anhang lösen.

Im Folgenden verstehen wir unter der Rechenzeit $T(n)$ in Abhängigkeit von der Problemgröße n stets die *Worst-Case*-Rechenzeit, sofern nicht explizit etwas anderes erwähnt wird.

1.5.1 Einfache Anweisungen

Gemeint sind Anweisungen, die keine Funktionsaufrufe oder Schleifenkonstrukte enthalten, insbesondere Wertzuweisungen von Variablen (in C: `=`), Ausdrücke mit Ungleichungs-, Gleichheits- oder arithmetischen Operatoren (in C: `==, !=, <, >, +, ++, %` usw.). Bis auf wenige Ausnahmen wird jede solche Anweisung durch den Compiler in eine Folge von Maschinensprachebefehlen übersetzt, deren Worst-Case-Ausführungszeit durch eine von der Problemgröße n unabhängige Konstante nach

oben abgeschätzt werden kann. **Man nimmt also an, daß die Worst-Case-Rechenzeit einer solchen Anweisung in $O(1)$ liegt.** Ausnahmen sind z.B. Zuweisungen von Structs, deren Länge von der Problemgröße abhängen kann.

1.5.2 Blöcke

Gemeint ist eine endliche Folge $P_1, \ldots, P_k$ von Programmteilen, die nacheinander ausgeführt werden (keine verschachtelten Schleifen oder `goto`-Anweisungen). Es sei $T_j(n)$ die Worst-Case-Rechenzeit von P_j für die Problemgröße n. Wenn $T_j \in O(f_j)$ für $j = 1, \ldots, k$ gilt, so ergibt sich für die Rechenzeit T des gesamten Blocks aus den P_j, daß $T \leq \sum_{j=1}^{k} T_j \in \sum_{j=1}^{k} O(f_j) = O(\sum_{j=1}^{k} f_j) = O(\max\{f_1, \ldots, f_k\})$, also $T \in O(\max\{f_1, \ldots, f_k\})$.

1.5.3 Blöcke aus einfachen Anweisungen

Ist jedes P_j eine einfache Anweisung, so gilt $T_j \in O(f_j)$ mit $f_j = 1$, also $T \in O(1)$.

1.5.4 Bedingte Anweisungen

Für prinzipielle Betrachtungen sind **if-then-else**-Anweisungen folgender Gestalt ausreichend:

if *Bedingung* **then** *if-Block* **else** *else-Block*

Für die Rechenzeit T gilt entweder $T \leq T_{\textit{Bedingung}} + T_{\textit{if-Block}}$ oder $T \leq T_{\textit{Bedingung}} + T_{\textit{else-Block}}$.

Kennt man Wachstumsabschätzungen

$$T_{\textit{if-Block}} \in O(f_{\textit{if-Block}})$$

und

$$T_{\textit{else-Block}} \in O(f_{\textit{else-Block}}),$$

so folgt

$$T \in O(T_{\textit{Bedingung}}) + O(\max\{f_{\textit{if-Block}}, f_{\textit{else-Block}}\}).$$

In vielen Fällen ist die Bedingung nicht von der Problemgröße n abhängig, also $T_{\textit{Bedingung}} \in O(1)$. Außer in dem trivialen Fall, daß sowohl der **if**-Teil als auch der **else**-Teil leer sind, ist $O(1) \subset O(\max\{T_{\textit{if-Block}}, T_{\textit{else-Block}}\})$, also insgesamt

$$T \in O(\max\{f_{\textit{if-Block}}, f_{\textit{else-Block}}\}).$$

1.5.5 Schleifen

Voraussetzungen:

1. Der Schleifenkörper ist nicht leer; seine Ausführung benötigt mindestens die Zeit 1.

2. Die Initialisierung und die Abbruchsbedingung (und bei **for**-Schleifen die Iterationsanweisung) enthalten nur einfache Anweisungen und keinen Funktionsaufruf, so daß zu ihrer Ausführung bei jedem Schleifendurchlauf (bzw. vor oder nach jedem Durchlauf) höchstens eine konstante, von der Problemgröße unabhängige Zeit $T_{\textit{iteration}}$ nötig ist, also $T_{\textit{iteration}} \in O(1)$.

Die Ausführungszeit für den Schleifenkörper sei in $O(f_{Körper})$. Die Anzahl der Schleifendurchläufe in Abhängigkeit von der Problemgröße n sei in $O(f_{Anzahl})$.

Dann gilt für die gesamte Rechenzeit T der Schleife

$$\begin{aligned} T &\in O(1) + O(f_{Anzahl}) \cdot O(1) + O(f_{Anzahl}) \cdot O(f_{Körper}) \\ &= O(1) + O(f_{Anzahl}) + O(f_{Anzahl} \cdot f_{Körper}). \end{aligned}$$

Wegen 1 gibt es ein $n_1 \in \mathbb{N}$ und ein $a_1 > 0$, so daß $f_{Körper}(n) \geq a_1$ für $n > n_1$. Setzt man voraus, daß es ein $n_2 \in \mathbb{N}$ und ein $a_2 > 0$ gibt, so daß $f_{Anzahl}(n) > a_2$ für $n > n_2$ gilt (was notwendigerweise der Fall ist, wenn die Schleife stets mindestens einmal durchlaufen wird), so gilt $O(1) \subset O(f_{Anzahl} \cdot f_{Körper})$ und $O(f_{Körper}) \subset O(f_{Anzahl} \cdot f_{Körper})$ und folglich

$$T \in O(f_{Anzahl} \cdot f_{Körper})$$

1.5.6 Beispiel

Mit den bisherigen Betrachtungen wollen wir das asymptotische Worst-Case-Verhalten von SelectionSort abschätzen.

Zeile (1): Nach 1.5.4 gilt $T \in O(1)$.
Innere **for**-Schleife: Nach 1.5.5 gilt $T \in O(n)$, weil die Anzahl der Schleifendurchläufe nicht größer als n ist.
Körper der äußeren **for**-Schleife: Nach 1.5.1 und 1.5.2 ergibt sich $T \in O(1) + O(n) + O(1) = O(n)$.
Äußere **for**-Schleife: Anzahl der Schleifendurchläufe $= n$, also gilt nach 1.5.5

$$T \in O(n \cdot n) = O(n^2)$$

1.5.7 Funktionen oder Prozeduren

Enthält ein Algorithmus Aufrufe von Prozeduren (in C: Funktionen), so erstellt man zunächst den Aufrufgraphen: Seine Knoten sind die in dem Algorithmus vorkommenden Prozeduren, und eine gerichtete Kante von P_1 nach P_2 gibt es genau dann, wenn die Prozedur P_2 innerhalb der Prozedur P_1 aufgerufen wird.

1. Fall: Der Aufrufgraph ist kreisfrei (rekursionsfreier Algorithmus).
Dann ruft eine Prozedur sich nie selbst auf, weder direkt noch indirekt d.h. durch Zwischenaufrufe anderer Prozeduren; es liegt also keine Rekursion vor. Man zerlegt die Menge der Prozeduren (die Knotenmenge des Aufrufgraphen) auf folgende Weise:
$K_0 =$ Menge der Prozeduren, die keine andere aufrufen
und für $l > 0$
$K_l =$ Menge der Prozeduren, die nur Prozeduren aus $\bigcup_{j=0}^{l-1} K_j$ aufrufen.

Weil es nur endlich viele Prozeduren gibt und der Aufrufgraph kreisfrei ist, gibt es mindestens eine Prozedur, die keine andere aufruft; also $K_0 \neq \emptyset$. Ist $l \geq$ Anzahl der Prozeduren, so muß K_l alle Prozeduren enthalten. Und es gilt $K_{l-1} \subset K_l$.
Setze $M_0 = K_0$ und $M_l = K_l \setminus K_{l-1}$ für $l > o$.
Dann ist $(M_l)_l$ eine Zerlegung der Menge aller Prozeduren in paarweise disjunkte Mengen.

Bestimmung der Rechenzeit T_P einer Prozedur P:
Es gibt genau ein $l \in \mathbb{N}$ mit $P \in M_l$. Zunächst bestimmt man auf die oben beschriebene Weise die Rechenzeit jeder Prozedur in M_0. dann bestimmt man induktiv für

$j = 1, \ldots, l$ für jede Prozedur P aus M_j die Rechenzeit, indem man wieder die obigen Regeln anwendet und die bereits bestimmten Rechenzeiten der Prozeduren aus K_{j-1} verwendet, sofern sie in P aufgerufen werden.

2. Fall: Der Aufrufgraph ist nicht kreisfrei.
Dann gibt es mindestens eine Prozedur P, die direkt oder indirekt sich selbst aufruft. Wir nennen solche Prozeduren rekursiv.

Man geht nun von der folgenden *Hypothese* aus: Man kann den Eingabeparametern jeder rekursiven Prozedur P eine natürliche Zahl n als Größe so zuordnen, daß gilt:

1. P ruft sich selbst nur mit Eingabeparametern auf, deren Größe echt kleiner ist.
2. Die Rechenzeit von P hängt (asymptotisch) nur von n ab. Bezeichnung: $T_P(n)$.
3. Es gibt ein $n_0 \in \mathbb{N}$, so daß für $n < n_0$ die Rekursion abbricht d.h. P sich nicht mehr selbst aufruft. Wir nennen n_0 Abbruchschwelle.

Für jede rekursive Prozedur P bestimmt man unterhalb ihrer Abbruchschwelle die Rechenzeit nach obigen Regeln. Oberhalb ihrer Abbruchschwelle erhält man für die Rechenzeit $T_P(n)$ eine Rekursionsgleichung

$$T_P(n) = \text{ Summe von einigen } T_P(n') \text{ mit } n' < n \quad + \text{ eine Funktion von } n$$

1.5.7.1 Beispiele

1.5.7.1.1 Türme von Hanoi

Wir betrachten den rekursiven Lösungsalgorithmus TvH aus 1.1.3.
Größe der Eingabeparameter = Anzahl der Scheiben.
Abbruchschwelle = 2.
$T_{\mathsf{TvH}}(1) = 1$.
Für $n \geq 2$ ruft sich TvH zweimal selbst auf mit $n-1$ als Größe und führt außerdem eine zusätzliche Scheibenbewegung aus. Daher lautet die Rekursionsgleichung für $n \geq 2$

$$T_{\mathsf{TvH}}(n) = 2 \cdot T_{\mathsf{TvH}}(n-1) + 1$$

1.5.7.1.2 MergeSort

Größe der Eingabe = Länge des zu sortierenden Arrays.
Abbruchschwelle = 2.
$T(1) = 0$.

$$T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + n - 1 \text{ für } n \geq 2$$

Folgerung: Man versucht, die Rechenzeit einer rekursiven Prozedur als Lösung einer Rekursionsgleichung zu bestimmen, die sich aus der Art der rekursiven Aufrufe ergibt.

Bemerkungen:

1. Für Rekursionsgleichungen gibt es keine einheitliche Lösungstheorie, für spezielle Arten jedoch schon (analog zu Differentialgleichungen).
2. Oft erhält man für die Worst-Case-Rechenzeiten nur Rekursionsungleichungen. So könnte man z.B. bei MergeSort abschätzen

$$T^{worst}(n) \leq 2 \cdot T^{worst}(\left\lceil \frac{n}{2} \right\rceil) + n - 1 \text{ für } n \geq 2$$

Diese Ungleichungen werden häufig mit der O, Θ, Ω-Schreibweise geschrieben, z.B.

$$T(n) = 2 \cdot T(\left\lceil \frac{n}{2} \right\rceil) + O(n-1)$$

oder

$$T_{\mathsf{TvH}}(n) = 2 \cdot T_{\mathsf{TvH}}(n-1) + \Theta(1)$$

Man spricht dann von einer *asymptotischen Rekursionsgleichung* und will dann auch nur die O- oder Θ-Klasse von T bestimmen.

1.6 Speicherbedarfanalyse

Für praktische Anwendungen ist nicht nur die Rechenzeit eines Algorithmus, sondern auch sein Speicherbedarf von Interesse. Zur Abschätzung des Speicherbedarfs verwendet man dieselben Methoden wie zur Abschätzung der Rechenzeit. Daher spricht man auch neutral von Aufwandsanalyse von Algorithmus.

1.7 Die Rolle der Faktoren bei Wachstumsabschätzungen

A_1 und A_2 seien Algorithmen, die dasselbe Problem lösen, deren Rechnezeiten T_{A_1} und T_{A_2} aber unterschiedlich schnell wachsen; sei z.B. $T_{A_1} \in O(n)$ und $T_{A_2} \in \Omega(n^2)$. Dann gibt es $c_1 > 0$ und $c_2 > 0$ und $n_0 \in \mathbb{N}$, so daß $T_{A_1}(n) \leq c_1 n$ und $T_{A_2}(n) \geq c_2 n^2$. Für $n > \frac{c_1}{c_2}$ folgt $T_{A_1}(n) \leq c_1 n < c_2 n^2 \leq T_{A_2}(n)$; d.h. A_1 ist für große n schneller als A_2, auch wenn c_1 sehr viel größer als c_2 ist.
Größenordnung setzt sich gegen Faktoren schließlich durch!

Die Tabelle in der Übungsaufgabe auf Blatt 1 zeigt, wie stark sich die Problemgrößen n unterscheiden können, die in einer vorgegebenen Zeit gelöst werden können.

Außerdem führe man sich vor Augen, daß der Übergang zu einem Algorithmus, dessen Rechenzeit asymptotisch langsamer wächst, für große n viel effizienter ist als der Übergang zu einem schnelleren Rechner.
Beispiel: A_1 benötige zur Lösung eines Problems der Größe n die Zeit $T_{A_1}(n) = n^2$ ms, A_2 dagegen nur $T_{A_2}(n) = n \log n$ ms.
dann kann man mit A_1 in einer Stunde ein Problem der Größe $n_1 = 1897$ lösen. Auf einem 100-mal schnelleren Rechner benötigt der Algorithmus A_1 die Zeit $T'_{A_1}(n) = \frac{1}{100}\, n^2$ ms. Damit kann man in einer Stunde ein Problem der Größe $n = \sqrt{100}\, n_1 = 10\, n_1 = 18970$ lösen. Durch Verwendung von Algorithmus A_2 dagegen kann man in einer Stunde ein Problem der Größe 286 498 lösen. Mit Algorithmus A_1 bräuchte man dafür einen Rechner, der $\left(\frac{286\,498}{1897}\right)^2 \approx 22809$-mal schneller ist.

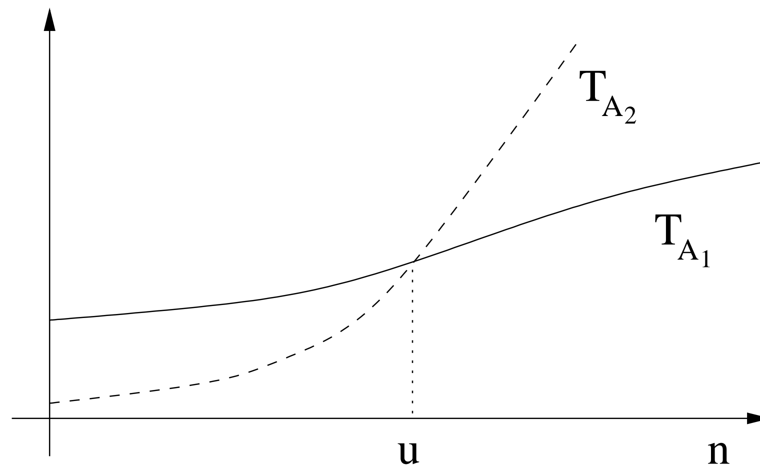
Langsame Algorithmen bleiben auch auf schnellen Rechnern langsam!

Nochmal anders ausgedrückt:
Löse ein Problem der Größe 10000, wobei $T_{A_1}(n) = n^2, T_{A_2}(n) = n \log_2 n$.

A_1 auf dem langsamen Rechner benötigt $(10000)^2$ ms $= 10^8$ ms $= 10^5$ s $\approx 27,77$ h .
A_1 auf dem 100-mal schnelleren Rechner $\frac{1}{100}(10000)^2 = 10^6$ ms $= 10^3$ s $\approx 16,66$ min .
A_2 auf dem langsamen Rechner $1000 \cdot \log_{10} 10000 = 10^4 \cdot 4$ ms $= 40$ s .

1.7.1 Der Übernahmepunkt

Das asymptotische Wachstum einer Funktion $T(n)$ für $n \to \infty$ sagt nichts über das Verhalten für kleine Argumente n aus. Deshalb kann es passieren, daß ein Algorithmus A_1 zwar asymptotisch schneller als ein anderer Algorithmus A_2 ist, aber für kleine n deutlich langsamer.



In praktischen Anwendungen ist es interessant, den sogenannten Übernahmepunkt zweier Algorithmen zu bestimmen; das ist ein $u \in \mathbb{N}$, so daß

$$T_{A_2}(n) \leq T_{A_1}(n) \text{ für } n \leq u$$

und

$$T_{A_2}(n) > T_{A_1}(n) \text{ für } n > u$$

Für $n \leq u$ verwendet man dann den Algorithmus A_2, für $n > u$ den Algorithmus A_1. Ist A_1 ein rekursiver Algorithmus, so wird man die Rekursion abbrechen, wenn die Eingabegröße $\leq u$ wird, und statt A_1 den Algorithmus A_2 zur Lösung dieses kleinen Problems verwenden.

Kapitel 2

Die Strategie *Divide et Impera*

2.1 Das Grundprinzip von *Divide et Impera*

Die Strategie *Divide et Impera* zur Lösung eines Problems, dessen Eingabe die Größe n hat, besteht in folgender Idee:

a) Zerlege das Problem in mehrere Probleme der gleichen Art wie das ursprüngliche, so daß

1. die Eingabe jedes dieser Teilprobleme kleiner als n ist,
2. die Lösung des ursprünglichen Problems aus den Lösungen der Teilprobleme berechnet werden kann.

b) Löse jedes der Teilprobleme

- entweder durch weitere Zerlegung wie in a)
- oder direkt
  (damit meint man, daß das Problem so einfach ist, daß es trivial ist oder durch einen sehr einfachen (aber nicht unbedingt asymptotisch schnellen) Algorithmus lösbar ist).

Diese Strategie führt also zu einem rekursiven Algorithmus. Seine Rechenzeit wird davon beeinflußt, bei welcher Eingabegröße man ein Teilproblem nicht mehr zerlegt, sondern direkt löst (Festlegung des Übernahmepunktes, Abbruch der Rekursion).

2.2 Typische Gestalt eines *Divide-et-Impera*-Algorithmus

Wir geben den Algorithmus in Form einer Funktion an, die bei Eingabe von x die Lösung y des Problems zurückgibt.

```
AUSGABETYP DivideEtImpera(EINGABETYP x) {
    if( x ist genügend klein )
        y = direkteLösung(x);
    else{
        zerlege x in kleinere Datensätze x_1, ..., x_a;
        for(j=1;j<=a;j=j+1) y_j = DivideEtImpera(x_j);
        setze die y_j zu einer Gesamtlösung y zusammen;
    }
    return y;
}
```

2.3 Vorteile der Strategie *Divide et Impera*

2.3.1 Oft bietet sich die benötigte Zerlegung des Problems auf natürliche Weise an; dann liefert die Strategie *Divide et Impera* auf einfache Weise einen Lösungsalgorithmus. Man denke an MergeSort oder an den rekursiven Algorithmus zur Lösung des Problems der Türme von Hanoi, bei dem die Strategie *Divide et Impera* allerdings etwas ausgeartet ist, weil die Größe der Teilprobleme sehr unterschiedlich ist.

2.3.2 Oft erhält man einen Algorithmus, der effizienter ist als der, der zur direkten Lösung der kleineren Teilprobleme eingesetzt wird. Die wollen wir an einem Beispiel verdeutlichen.

2.3.3 Beispiel für Effizienzsteigerung

A sei ein Algorithmus mit Rechenzeit $T_1(n) = c\,n^2$ mit $c > 0$.
Nehmen wir an, ein Problem der Größe n lasse sich in 3 Probleme der Größe $\lceil \frac{n}{2} \rceil$ aufteilen. Für das Zusammenfügen ihrer Lösungen zu einer Gesamtlösung werde $f(n) = b\,n$ Zeit benötigt. Löst man die 3 Teilprobleme wieder mit dem Algorithmus A , so benötigt man die Zeit

$$\begin{aligned} T_2(n) &= 3\,T_1(\left\lceil \frac{n}{2} \right\rceil) + bn \\ &\leq 3c\,(\frac{n}{2}+1)^2 + bn \\ &= \frac{3}{4}c\,n^2 + (3c+b)n + 3c \\ &= \frac{3}{4}T_1(n) + (3c+b)n + 3c \\ &\in O(T_1) \end{aligned}$$

1. Folgerung: Für große n ist $T_2(n)$ fast 25% kleiner als $T_1(n)$. Das asymptotische Wachstum wird jedoch nicht langsamer, denn $T_2(n) \geq 3\,c\,(\frac{n}{2})^2 + b\,n \geq \frac{3}{4}\,T_1(n)$, also $T_2 \in \Omega(T_1)$.

Was ändert sich, wenn wir die Teilprobleme rekursiv mit derselben Strategie lösen?

Nehmen wir an, daß Teilprobleme, deren Größe $\leq n_0$ ist, mit dem Algorithmus A gelöst werden. Dann ergibt sich für die Rechenzeit dieses *Divide-et-Impera*-Algorithmus

$$T_3(n) = \begin{cases} c\,n^2 & \text{für } n \leq n_0 \\ 3T_3(\lceil \frac{n}{2} \rceil) + bn & \text{für } n > n_0 \end{cases}$$

Für $n = 2^m n_0, m > 0$, gilt

$$\begin{aligned}
T_3(n) &= T_3(2^m n_0) = 3\,T_3(2^{m-1}n_0) + b\,n \\
&= 3 \cdot 3 \cdot T_3(2^{m-2}n_0) + 3\,b\,2^{m-1}n_0 + b\,2^m n_0 \\
&= \ldots \\
&= 3^m\,T_3(n_0) + 3^{m-1}\,b\,2\,n_0 + \ldots + 3^0\,b\,2^m\,n_0 \\
&= 3^m c\,n_0^2 + b\,n_0 \sum_{j=1}^{m} 3^{m-j}2^j \\
&= n_0 3^m \left(c\,n_0 + b \sum_{j=1}^{m} \left(\frac{2}{3}\right)^j \right) \\
&= n_0 2^{m \log_2 3} \left(c\,n_0 + b \sum_{j=1}^{m} \left(\frac{2}{3}\right)^j \right) \\
&\leq n_0 \left(\frac{n}{n_0}\right)^{\log_2 3} \left(c\,n_0 + \sum_{j=1}^{\infty} \left(\frac{2}{3}\right)^j \right)
\end{aligned}$$

2. Folgerung: Auf $D = \{2^m n_0 \colon m \in \mathbb{N}\}$ gilt $T_3 \in O(n^{\log_2 3}) \subsetneq O(n^2)$ (wegen $\log_2 3 < 2$). Wenn man nur Probleme einer Größe $2^m n_0$ betrachtet, wächst die Rechenzeit des *Divide-et-Impera*-Algorithmus echt langsamer als die von A .

Ungeklärt ist, ob $T_3 \in O(n^{\log_2 3})$ auch gilt, wenn man als Definitionsbereich ganz $\mathbb{N}$ wählt. Diese Frage untersuchen wir im Anhang genauer. In obigem Beispiel gilt tatsächlich $T_3 \in O(n^{\log_2 3})$ auf ganz $\mathbb{N}$.

2.4 Typische *Divide-et-Impera*-Situationen

Oft ist die Anzahl a der Teilprobleme, in die ein Problem zerlegt wird, recht klein und unabhängig von der Größe n der Eingabe. Sei $\gamma_i(n)$ die Größe des i-ten Teilproblems, $\gamma_i(n) < n$. Dann erfüllt die Rechenzeit eine Rekursionsgleichung der Form

$$T(n) = T(\gamma_1(n)) + \ldots T(\gamma_a(n)) + f(n),\ n \geq n_0, \tag{2.1}$$

wobei $f(n)$ der Aufwand ist, der nötig ist, um aus den Lösungen der Teilprobleme eine Lösung des Gesamtproblems zu machen.

Meist versucht man (oder es ergibt sich), daß die Teilprobleme ähnliche Größen haben. Häufig gibt es ein $b \in \mathbb{N}$, so daß $\gamma_i(n) = \lfloor \frac{n}{b} \rfloor$ oder $\gamma_i(n) = \lceil \frac{n}{b} \rceil$ gilt. Bei MergeSort gilt z.B. $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n - 1$.
Für Problemgrößen der Form $n = b^m n_0$ (n_0 Abbruchschwelle, Übernahmepunkt) gilt dann

$$T(n) = aT(\tfrac{n}{b}) + f(n), \quad T(n_0) = \alpha. \tag{2.2}$$

Um die Rechenzeit so eines *Divide-et-Impera*-Algorithmus asymptotisch abzuschätzen, geht man meist folgendermaßen vor:

1. Man betrachtet erst die Gleichung 2.2 auf dem Definitionsbereich $D = \{b^m n_0 \colon m \in \mathbb{N} \cup \{0\}\}$ und bestimmt für ihre Lösung $T_{2.2}$ eine Wachstumsabschätzung $T_{2.2} \in O(g)$ (die nur auf D gilt!), wobei $g \colon \mathbb{N} \to \mathbb{R}$ eine asymptotisch nichtnegative Funktion ist.

2. Dann betrachtet man die Gleichung 2.1. Ihre Lösung $T_{2.1}$ erfüllt ebenfalls $T_{2.1} \in O(g)$ (aber auf $\mathbb{N}$!), wenn $T_{2.1}$ und g gewisse asymptotische Eigenschaften haben.

   Die genauen Einzelheiten sind im Anhang nachzulesen.

2.5 Asymptotische Rekursionsgleichungen

Manchmal kann oder will man den Aufwand, der für das Zusammenfügen der Teillösungen erforderlich ist, nicht genau bestimmen und begnügt sich mit der Abschätzungen der folgenden Form:

Es gibt $n_0 \in \mathbb{N}$, $c_1 > 0$, $c_2 > 0$ und zwei Funktionen $T_1 \colon \{1, \ldots, n_0\} \to [0, \infty[$ und $T_2 \colon \{1, \ldots, n_0\} \to [0, \infty[$ sowie eine Funktion $f \colon \mathbb{N} \to]0, \infty[$, so daß

$$T(n) \quad \geq \quad \begin{cases} T_1(n) & \text{für } n \leq n_0 \\ T(\gamma_1(n)) + \ldots + T(\gamma_a(n)) + c_1 f(n) & \text{für } n > n_0 \end{cases} \tag{2.3}$$

und/oder

$$T(n) \quad \leq \quad \begin{cases} T_2(n) & \text{für } n \leq n_0 \\ T(\gamma_1(n)) + \ldots + T(\gamma_a(n)) + c_2 f(n) & \text{für } n > n_0 \end{cases} \tag{2.4}$$

Diese Aussagen werden zusammengefaßt zu der asymptotischen Rekursionsgleichung

$$T \in T \circ \gamma_1 + \ldots + T \circ \gamma_a + \Theta(f)\,, \text{ wenn 2.3 und 2.4 gelten,}$$

$$T \in T \circ \gamma_1 + \ldots + T \circ \gamma_a + O(f)\,, \text{ wenn 2.4 gilt,}$$

$$T \in T \circ \gamma_1 + \ldots + T \circ \gamma_a + \Omega(f)\,, \text{ wenn 2.3 gilt.}$$

Die Wachstumsklasse von T läßt sich mit derjenigen der Lösung der Rekursionsgleichung

$$S(n) = S(\gamma_1(n)) + \ldots + S(\gamma_a(n)) + f(n),\ S(1) = 1$$

bestimmen; denn es gilt

2.5.1 Satz Ist S die Lösung der Rekursionsgleichung

$$S(n) = S(\gamma_1(n)) + \ldots + S(\gamma_a(n)) + f(n),\ S(1) = 1,$$

so gilt $T \in \Theta(S)$, wenn 2.3 und 2.4 gelten,

$T \in O(S)$, wenn 2.4 gilt,

$T \in \Omega(S)$, wenn 2.3 gilt.

Beweis:
Seien $d_1 = \min\{c_1, \frac{T_1(1)}{S(1)}, \ldots, \frac{T_1(n_0)}{S(n_0)}\}$ und $d_2 = \max\{c_2, \frac{T_2(1)}{S(1)}, \ldots, \frac{T_2(n_0)}{S(n_0)}\}$.
Wir zeigen $d_1 S(n) \leq T(n) \leq d_2 S(n)$ durch vollständige Induktion nach n.

Induktionsanfang: $n \leq n_0$

$$d_1S(n) \leq T_1(n) \leq T(n) \leq T_2(n) \leq d_2S(n)$$

Induktionsvoraussetzung: Die Aussage gilt für $m \leq n$.
Induktionsschritt von n nach $n+1$: Wegen $\gamma_j(n+1) \leq n$ folgt

$$\begin{aligned}
d_1S(n+1) &= d_1S(\gamma_1(n+1)) + \ldots + d_1S(\gamma_a(n+1)) + d_1f(n+1) \\
&\leq T(\gamma_1(n+1)) + \ldots + T(\gamma_a(n+1)) + d_1f(n+1) \\
&\leq T(n+1) \\
&\leq T(\gamma_1(n+1)) + \ldots + T(\gamma_a(n+1)) + d_2f(n+1) \\
&\leq d_2S(\gamma_1(n+1)) + \ldots + d_2S(\gamma_a(n+1)) + d_2f(n+1) \\
&= d_2S(n+1)
\end{aligned}$$

q.e.d.

2.6 Beispiel: Binäre Suche

2.6.1 Aufgabe 1 Gib einen Algorithmus an, der zu jeder Folge $a_1, \ldots, a_n \in \mathbb{R}$ und jedem $x \in \mathbb{R}$ feststellt, ob es ein $j \in \{1, \ldots, n\}$ mit $a_j = x$ gibt und gegebenenfalls so ein j ausgibt. Versuche ihn möglichst effizient zu wählen.

Bemerkung Die a_j müssen keine reellen Zahlen sein, sondern können Elemente einer beliebigen geordneten Menge sein wie z.B. Wörter (aus den üblichen Buchstaben) mit der lexikografischen Ordnung. Die Elemente der geordneten Menge nennt man in dieser Situation auch Schlüssel. In praktischen Anwendungen sind diese Schlüssel oft alphanumerische Zeichenketten (z.B. Namen, Matrikelnummern, Bestellnummern), denen jeweils ein Datensatz (z.B. in Gestalt eines `struct`) zugeordnet ist. Um einen Datensatz zu finden, sucht man nach dessem Schlüssel.

Natürlich gibt es einen Algorithmus, der Aufgabe 1 löst, nämlich die sogenannte lineare Suche. Wie effizient er ist, wollen wir genauer untersuchen.

2.6.2 Der naive Ansatz: Lineare Suche

```
LinSuche(x, a_1, ..., a_n) {
        j = 1;
        while(j ≤ n) {
                if(x == a_j) gib j aus und stoppe;
                j = j + 1;
        }
        gib aus, daß es kein j gibt;
}
```

Dadurch daß dieser Algorithmus auf die a_j rein sequentiell zugreift, können sich die a_j in einer nur sequentiell lesbaren Datei eines Massenspeichers befinden und müssen nicht erst in einen frei adressierbaren Speicher (RAM) transferiert werden.

Als Maß für den Rechenaufwand nehmen wir die Anzahl $T(n)$ der Vergleiche zwischen x und den a_j.

Die lineare Suche braucht **im besten Fall** 1 Vergleich (wenn nämlich $x = a_1$ ist) und **im schlechtesten Fall** n Vergleiche (wenn $x \notin \{a_1, \ldots, a_n\}$).

Aufwand im Mittel, average case: Wir nehmen an, daß die a_j paarweise verschieden sind und daß die Ereignisse $\{x = a_1\}, \ldots, \{x = a_n\}$, $\{x \notin \{a_1, \ldots, a_n\}\}$ die gleiche Wahrscheinlichkeit $\frac{1}{n+1}$ haben. Im Fall $x = a_j$ werden j Vergleiche benötigt, im Fall $x \notin \{a_1, \ldots, a_n\}$ genau n Vergleiche. Im Mittel werden somit $\frac{1}{n+1}\left(\sum_{j=1}^{n} j + n\right) = \frac{1}{n+1}\left(\frac{n}{2}(n+1) + n\right) = \frac{n}{2} + \frac{n}{n+1}$ Vergleiche ausgeführt.

Wir wollen noch überlegen, daß eine *Divide-et-Impera*-Strategie hier keine Vorteile bringt. Sie würde z.B. so aussehen:

(1) Teile $a_1, \ldots, a_n$ in $a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ und $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n$ auf.

(2) Suche x in jeder Teilfolge.

Im schlechtesten Fall gilt $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$. Beschränken wir uns auf $n = 2^k$, so folgt $T(2^k) = 2\,T(2^{k-1})$. Mit der Anfangsbedingung $T(2^0) = T(1) = 1$ ergibt sich $T(2^k) = 2^k$, also $T(n) = n$.

2.6.3 Aufgabe 2 Gib einen Algorithmus an, der zu jeder *sortierten* Folge von Zahlen $a_1, \ldots, a_n \in \mathbb{R}$ (d.h. $a_1 \leq \ldots \leq a_n$ und jedem $x \in \mathbb{R}$ feststellt, ob es ein $j \in \{1, \ldots, n\}$ mit $a_j = x$ gibt und gegebenenfalls so ein j ausgibt.

Die lineare Suche löst natürlich auch diese Aufgabe. Es gibt jedoch einen anderen Algorithmus, der effizienter ist. Und zwar bringt für sortierte Eingabefolgen ein *Divide-et-Impera*-Ansatz Vorteile: denn man braucht nur festzustellen, ob $x \leq a_{\lfloor \frac{n}{2} \rfloor}$ gilt; wenn ja, durchsucht man nur die erste Teilfolge, wenn nein, nur die zweite. Die *Divide-et-Impera*-Strategie artet also zu einer bloßen Vereinfachung aus; ein Zusammenfügen der Teillösungen ist nicht nötig. Übrigens verfährt man ähnlich, wenn man manuell in einer Kartei sucht.

2.6.4 Binäre Suche

Wir stellen den Algorithmus als Funktion dar, die mit BinSuche$(x, a_1, \ldots, a_n)$ aufgerufen wird.

BinSuche$(x, a_l, \ldots, a_r)$ {
    $n = r - l + 1$;
    if$(n == 1)$ {
        if$(x == a_l)$ gib l aus und stoppe;
        else return;
    }
    else if$(x \leq a_{l+\lfloor \frac{n}{2} \rfloor - 1})$ BinSuche$(x, a_l, \ldots, a_{l+\lfloor \frac{n}{2} \rfloor - 1})$;
        else BinSuche$(x, a_{l+\lfloor \frac{n}{2} \rfloor}, \ldots, a_r)$;
}

Der Aufruf von BinSuche$(x, a_1, \ldots, a_n)$ endet ohne Ausgabe, wenn x in der Folge nicht vorkommt, oder gibt ein j mit $x = a_j$ aus.

2.6.5 Aufwandsabschätzung

$T(n)$ sei die Anzahl der Vergleiche von a_j, die bei Eingabe einer Folge der Länge n im schlechtesten Fall ausgeführt wird.

Es gilt $T(n) \leq T(\lceil \frac{n}{2} \rceil) + 1$.

Um das Wachstum von T abzuschätzen, betrachten wir die Lösung S der Rekursionsgleichung

$$S(n) = S(\left\lceil \frac{n}{2} \right\rceil) + 1,\ n > 1,\ S(1) = 1$$

Nach 2.5.1 gilt $T \in O(S)$. Und für S wissen wir nach C.2.4, daß $S \in \Theta(\log n)$ gilt (setze $a = 1$, $b = 2$, $l = 0$, Fall $a = b^l$). Mit A.2.6 folgt $S \in \Theta(\log n) \subset O(\log n)$ und $O(S) \subset O(\log)$. Insgesamt ergibt sich

$$T \in O(\log)$$

2.6.6 Genauere Aufwandsanalyse

Sei $2^k \leq n < 2^{k+1}$, $k \in \mathbb{N}$. Die Dualdarstellung von n hat dann $k + 1 = \lfloor \log_2 n \rfloor + 1$ Stellen d.h. $n = \sum_{j=0}^{k} \nu_j\, 2^j$ mit $\nu_j \in \{0, 1\}$ und $\nu_k = 1$ (vgl. die Übungen). Es gilt

$$2^{k-1} \leq \lfloor \tfrac{n}{2} \rfloor \leq \lceil \tfrac{n}{2} \rceil < 2^k, \text{ falls } k \geq 2$$

und

$$2^{k-1} \leq \left\lfloor \frac{n}{2} \right\rfloor \leq \left\lceil \frac{n}{2} \right\rceil \leq 2^k, \text{ falls } k = 1$$

Wendet man diese Überlegungen auf den Algorithmus BinSuche an, so sieht man, daß die Länge der durchsuchten Teilarrays nach $k-1$ Rekursionsschritten (für $k \geq 2$) einen Wert l mit $2 = 2^1 \leq l < 2^2 = 4$, also $l = 2$ oder $l = 3$, erreicht. Somit wird nach k oder $k+1$ Rekursionsschritten die Länge 1 erreicht und die Rekursion abgebrochen. Folglich gilt

$$\lfloor \log_2 n \rfloor + 1 = k + 1 \leq \text{ Anzahl der Vergleiche } \leq k + 2 = \lfloor \log_2 n \rfloor + 2$$

Daraus folgt

$$T \in \Theta(\log_2)$$

2.6.7 Mittlerer Aufwand:

Meist setzt man voraus, daß die a_j paarweise verschieden sind und setzt $T_{mittel}(n) = \frac{1}{n+1}(V_1 + \ldots + V_0)$, wobei V_j = Anzahl der Vergleiche, falls $x = a_j$ und $j \in \{1, \ldots, n\}$, und V_0 = Anzahl der Vergleiche, falls $x \neq a_j$ für jedes j. Aus obiger Ungleichung ergibt sich

$$T_{mittel} \in \Theta(\log_2)$$

2.6.8 Iterative Realisierung der binären Suche

Die binäre Suche läßt sich auch ohne Rekursion als iterativer Algorithmus formulieren.

BinSuche_iterativ$(x, a_1, \ldots, a_n)$ {
  links = 1; rechts = n;
  do {
    $m = \lfloor \frac{1}{2}(\text{links} + \text{rechts}) \rfloor$;
    if$(x < a_m)$ rechts = $m - 1$;
    else links = $m + 1$;
  } **while**$((x \neq a_m)$ und (links $\leq$ rechts));
  if$(x == a_m)$ gib m aus;
}

2.6.9 Bemerkung Es lohnt sich eine Datenfolge zu sortieren, wenn man anschließend sehr oft Elemente darin suchen will; denn bereits für recht kleine n ist $\log_2 n$ viel kleiner als n; so gilt z.B. $\frac{1000}{\log_2 1000} \approx 100$.

2.7 Beispiel: Berechnung von $\lceil\sqrt{n}\,\rceil$ für $n \in \mathbb{N}$

Wir wollen untersuchen, wieviele Multiplikationen benötigt werden, um $\lceil\sqrt{n}\,\rceil$ zu berechnen.

2.7.1 Naiver Algorithmus

Setze $m = 1$.
while$(m^2 < n)$ $m = m + 1$;
return m.

Dieser Algorithmus überprüft alle $m \in \mathbb{N}$ mit $m \leq \lceil\sqrt{n}\rceil$; er braucht also $T(n) = \lceil\sqrt{n}\,\rceil$ Multiplikationen.

Wir wollen zeigen, daß ein *Divide-et-Impera*-Ansatz zu einem schnelleren Algorithmus führt.

2.7.2 *Divide-et-Impera*-Ansatz Die Idee besteht darin, das ursprüngliche Suchintervall $[1, n]$ in zwei Hälften aufzuteilen und zu entscheiden, in welcher Hälfte $\lceil\sqrt{n}\,\rceil$ liegen muß, und in dieser mit derselben Strategie weiterzusuchen.

1. Setze $l = 1$ und $r = n$.
2. Gilt $\left(\left\lfloor\frac{l+r}{2}\right\rfloor\right)^2 \geq n$, so setze $r = \left\lfloor\frac{l+r}{2}\right\rfloor$, ansonsten setze $l = \left\lfloor\frac{l+r}{2}\right\rfloor + 1$.
3. Wenn $l < r$, gehe zu 2. Ansonsten gib l aus und stoppe.

Als rekursiver Algorithmus formuliert:

Wurzel(n, l, r) {
if$(l == r)$ gib l aus und stoppe;
else if$(\left\lfloor\frac{l+r}{2}\right\rfloor * \left\lfloor\frac{l+r}{2}\right\rfloor \geq n)$ Wurzel$(n, l, \left\lfloor\frac{l+r}{2}\right\rfloor)$;
else Wurzel$(n, \left\lfloor\frac{l+r}{2}\right\rfloor + 1, r)$;
}

2.7.3 Lemma Beim Aufruf von Wurzel$(n, 1, n)$ wird $\lceil\sqrt{n}\rceil$ ausgegeben.

Beweis: Zu zeigen ist, daß der Algorithmus terminiert und korrekt ist.
Der Algorithmus terminiert, weil in 2. die Länge des Intervalls $\{l, \ldots, r\}$ echt abnimmt.
Wir zeigen nun, daß während der Ausführung des Algorithmus stets

$$l \leq \lceil\sqrt{n}\rceil \leq r$$

gilt. Dazu führen wir eine vollständige Induktion über die Anzahl k der rekursiven Aufrufe des Algorithmus durch.
$k = 1$: Dann sind $l = 1$ und $r = n$, also $1 \leq \sqrt{n} \leq n$ und somit $1 \leq \lceil\sqrt{n}\rceil \leq n$.
Induktionsvoraussetzung: Nach k Aufrufen gelte $l \leq \lceil\sqrt{n}\rceil \leq r$.
Induktionsschritt: Rufe Wurzel(n, l, r) auf. Wenn $l = r$ gilt, ist nichts zu zeigen.
Sei $\left(\left\lfloor\frac{l+r}{2}\right\rfloor\right)^2 \geq n$. Dann gilt $\sqrt{n} \leq \left\lfloor\frac{l+r}{2}\right\rfloor$, also mit der Induktionsvoraussetzung

$$l \leq \lceil\sqrt{n}\rceil \leq \left\lfloor\frac{l+r}{2}\right\rfloor$$

und somit nach Ersetzung von r durch $\left\lfloor \frac{l+r}{2} \right\rfloor$ die Induktionsbehauptung

$$l \leq \sqrt{n} \leq r$$

Sei $\left(\left\lfloor \frac{l+r}{2} \right\rfloor\right)^2 < n$. Dann gilt

$$\left\lfloor \frac{l+r}{2} \right\rfloor < \sqrt{n} \leq \left\lceil \sqrt{n} \right\rceil$$

also mit der Induktionsvoraussetzung

$$\left\lfloor \frac{l+r}{2} \right\rfloor + 1 \leq \left\lceil \sqrt{n} \right\rceil \leq r$$

und somit nach Ersetzung von l durch $\left\lfloor \frac{l+r}{2} \right\rfloor + 1$ die Induktionsbehauptung

$$l \leq \left\lceil \sqrt{n} \right\rceil \leq r$$

Wenn der Algorithmus terminiert, gilt $l = r$ und somit $l = \lceil \sqrt{n} \rceil$. q.e.d.

2.7.4 Aufwandsanalyse

Bei jedem Rekursionsschritt wird eine Multiplikation durchgeführt und die Anzahl m der Zahlen in $\{l, \ldots, r\}$ wird auf $\left\lfloor \frac{m}{2} \right\rfloor$ oder $\left\lceil \frac{m}{2} \right\rceil$ erniedrigt. Wie bei der binären Suche folgt, daß nach höchstens $\lfloor \log_2 n \rfloor + 1$ Rekursionsschritten $m = 1$ ist. Somit ist die Anzahl der Multiplikationen in $O(\log)$, und es gilt $O(\log_2 n) \subsetneq O(\sqrt{n})$.

2.8 Beispiel: Die schnelle Fouriertransformation

Die Fouriertransformation spielt eine eminente Rolle in der linearen Signalverarbeitung, und in technischen Anwendungen (und naturwissenschaftlichen Experimenten) wiederum spielt die lineare Verarbeitung von Signalen eine wichtige Rolle. Deswegen ist es von besonderem Interesse, sie schnell berechnen zu können. Wir wollen hier die Grundidee der sogenannten schnellen Fouriertransformation (FFT = Fast Fourier Transform) skizzieren, weil sie auf einem *Divide-et-Impera*-Ansatz beruht. i sei in diesem Abschnitt stets $\sqrt{-1}$.

Ist $f\colon \mathbb{R} \to \mathbb{C}$ Lebesgue-integrierbar, so definiert man die **Fouriertransformierte** $\hat{f}\colon \mathbb{R} \to \mathbb{C}$ von f als

$$\hat{f}(\omega) = \int_{\mathbb{R}} f(t)\, e^{-2\pi i\omega t} dt$$

Die Abbildung $f \mapsto \hat{f}$ heißt **Fouriertransformation**. Falls $\hat{f}$ wiederum Lebesgue-integrierbar ist, gilt die sogenannte Umkehrformel

$$f(t) = \int_{\mathbb{R}} \hat{f}(\omega)\, e^{2\pi i\omega t} d\omega$$

Diese Darstellung von f läßt sich anschaulich deuten. Man erinnere sich, daß $e^{i\omega t} = \cos \omega t + i \sin \omega t$ gilt. Folglich gibt die Umkehrformel eine Zerlegung von f in Cosinus- und Sinus-Schwingungen an. Dabei beschreibt der Wert $\hat{f}(\omega)$ den Beitrag der Schwingungen mit Frequenz ω, und zwar gibt der Betrag $|\hat{f}(\omega)|$ an, wie stark die Schwingungen der Frequenz ω beteiligt sind, und der Winkel $\arg \hat{f}(\omega)$ gibt die Phasenlage an. Sowohl die Fouriertransformation als auch die Umkehrformel lassen sich von dem Raum der Lebesgue-integrierbaren Funktionen auf größere Räume (von Funktionen oder Distributionen) ausdehnen. Allerdings gelten dort obige Formeln nicht mehr uneingeschränkt.

In Anwendungen hat man meist von einem Signal $\mathbb{R} \to \mathbb{C}$ nur die Werte an endlich vielen äquidistanten Stellen $t_0, \ldots, t_{n-1}$ gemessen. Es liegt also nur eine Funktion $f\colon \{0, \ldots, n-1\} \to \mathbb{C}$ vor; das ist nichts anderes als ein Vektor in $\mathbb{C}^n$. Dementsprechend verwendet man eine diskretes Analogon der Fouriertransformation.

Als **diskrete Fouriertransformierte** $\hat{f}$ von $f\colon \{0, \ldots, n-1\} \to \mathbb{C}$ bezeichnet man die Funktion $\hat{f}\colon \{0, \ldots, n-1\} \to \mathbb{C}$, die folgendermaßen definiert ist:

$$\hat{f}(j) = \sum_{k=0}^{n-1} f(k)\, e^{-\frac{2\pi i}{n} jk} \tag{2.5}$$

Die Abbildung $\mathbb{C}^n \to \mathbb{C}^n$, $f \mapsto \hat{f}$ heißt die **diskrete Fouriertransformation (DFT)**. Sie beschreibt eigentlich einfach einen Koordinatenwechsel im $\mathbb{C}^n$. Für jedes f gilt die Umkehrformel

$$f(k) = \frac{1}{n} \sum_{j=0}^{n-1} \hat{f}(j)\, e^{\frac{2\pi i}{n} jk}$$

Achtung! Der Zusammenhang zwischen der Fouriertransformierten eines kontinuierlichen Signals, das auf ganz $\mathbb{R}$ definiert ist, und der diskreten Fouriertransformierten eines daraus durch Abtastung entstandenen diskreten Signals ist nicht trivial! Das Aussehen der diskreten Fouriertransformierten kann leicht zu unüberlegten Fehlschlüssen verleiten.

Die rechte Seite der Definition von $\hat{f}$ ist für alle $j \in \mathbb{Z}$ definiert. Und wegen $e^{2\pi i} = 1$ ist sie periodisch mit Periode n. Deshalb kann man die diskrete Fouriertransformierte auch als periodische Funktion auf $\mathbb{Z}$ ansehen. Sie ist vollständig bestimmt durch ihre Werte auf einem Intervall der Länge n. Häufig wählt man als Definitionsbereich das fast symmetrische Intervall $\{-\frac{n}{2}, \dots, -1, 0, 1, \dots, \frac{n}{2} - 1\}$, um die Argumente in Analogie zum kontinuierlichen Fall als positive und negative Frequenzen deuten zu können.

Die Berechnung von $\hat{f}$ gemäß der Definition 2.5 benötigt n^2 Multiplikationen und $n(n-1)$ Additionen. (Die Berechnung der $e^{-\frac{2\pi i}{n} j\, k}$ wird nicht gezählt, weil sie in Tabellen abgelegt werden können.)

Es gibt schnellere Algorithmen zur Berechnung der DFT, die sogenannten **FFT-Algorithmen (Fast Fourier Transform)**. Sie beruhen alle auf einer *Divide-et-Impera*-Strategie. Wir skizzieren die Grundidee.

n sei ab jetzt eine Zweierpotenz, $n = 2^m$, $m = \log_2 n \in \mathbb{N}$.

2.8.1 Lemma (Danielson, Lanczos 1942)

Gegeben sei $f\colon \{0, \dots, n-1\} \to \mathbb{C}$.
Definiere f_0, $f_1\colon \{0, \dots, \frac{n}{2} - 1\} \to \mathbb{C}$ durch $f_0(j) = f(2j)$ und $f_1(j) = f(2j+1)$ für $j \in \{0, \dots, \frac{n}{2} - 1\}$.
Dann gilt für $k \in \{0, \dots, \frac{n}{2} - 1\}$

$$\begin{aligned} \hat{f}(k) &= \hat{f}_0(k) + e^{-\frac{2\pi i k}{n}} \hat{f}_1(k) \\ \hat{f}(\tfrac{n}{2} + k) &= \hat{f}_0(k) - e^{-\frac{2\pi i k}{n}} \hat{f}_1(k) \end{aligned}$$

Beweis: Nachrechnen

$\widehat{f_0}$ und $\widehat{f_1}$ berechnet man wiederum nach obigem Schema aus $\widehat{f_{00}}$ und $\widehat{f_{01}}$ bzw. aus $\widehat{f_{10}}$ und $\widehat{f_{11}}$ usw., bis nach $m = \log_2 n$ Schritten Funktionen $f_{\alpha_1 \dots \alpha_m}$ mit $(\alpha_1, \dots, \alpha_m) \in \{0,1\}^m$ entstehen, deren Definitionsbereich $\{0, \dots, \frac{n}{2^m} - 1\} = \{0\}$ auf einen Punkt zusammengeschrumpft ist. Von solchen Funktionen läßt sich die DFT einfach berechnen; es gilt nämlich $\widehat{f_{\alpha_1 \dots \alpha_m}}(0) = f_{\alpha_1 \dots \alpha_m}(0)$.

Für jedes $(\alpha_1, \dots, \alpha_m) \in \{0,1\}^m$ ist $f_{\alpha_1 \dots \alpha_m}(0)$ gleich $f(j)$ für ein $j \in \{0, \dots, n-1\}$, aber für welches j?

Genaueres Nachdenken zeigt $f_{\alpha_1 \dots \alpha_m}(0) = f(\sum_{l=0}^{m-1} \alpha_{l+1}\, 2^l)$. Man beachte, daß die Dualdarstellung von $\sum_{l=0}^{m-1} \alpha_{l+1}\, 2^l$ gerade $\alpha_m \dots \alpha_1$ ist, also gleich dem Index in umgekehrter Reihenfolge (bit reversal).

Damit hat man einen rekursiven Algorithmus zur Berechnung von $\hat{f}$ gefunden. Die Anzahl $T_{\text{mult}}(n)$ der benötigten Multiplikationen und die Anzahl $T_{\text{add}}(n)$ der benötig-

ten Additionen erfüllen die folgenden Rekursionsgleichungen.

$$\begin{aligned} T_{\text{mult}}(n) &= 2\,T_{\text{mult}}(\tfrac{n}{2}) + \frac{n}{2}, \quad n \in D = \{2^k : k \in \mathbb{N}\}, \quad T_{\text{mult}}(1) = 0 \\ T_{\text{add}}(n) &= 2\,T_{\text{add}}(\tfrac{n}{2}) + n, \quad n \in D, \quad T_{\text{add}}(1) = 0 \end{aligned}$$

Nach C.1.2 ($a = 2$, $b = 2$, $l = 1$, $a = b^l$) gilt auf D

$$\begin{aligned} T_{\text{mult}}(n) &\in \Theta(n \log n) \\ T_{\text{add}}(n) &\in \Theta(n \log n) \end{aligned}$$

Dies ist eine wesentliche asymptotische Beschleunigung gegenüber der direkten Berechnung.

2.8.2 Bemerkungen **1.** In der Praxis werden FFT-Algorithmen iterativ ohne Rekursion implementiert; man fängt bei den $f_{\alpha_1 \ldots \alpha_m}$ zu rechnen an. Es gibt raffinierte Implementierungen, siehe z.B. das Buch *Numerical Recipes in C*.

2. Die multiplikativen Wachstumskonstanten sind recht klein; es gilt $T_{\text{mult}}(n) \leq \frac{n}{2} \log_2 n$ und $T_{\text{add}}(n) \leq n \log_2 n$. Deshalb sind die FFT-Algorithmen bereits für kleine n schneller als die direkte Berechnung.

3. Ist n keine Zweierpotenz, verlängert man meist f durch Nullen bis zur nächsten Zweierpotenz. Es gibt aber auch FFT-Algorithmen für n, die keine Zweierpotenzen sind. Zur weiteren Orientierung siehe z.B.
R.E. Blahut: *Fast Algorithms for Digital Signal Processing.* Addison-Wesley 1985.

2.8.3 Vergleich des Rechenaufwands von DFT und FFT

n	n^2 (DFT)	$\frac{n}{2} \log_2 n$ (FFT)	Speed Up
4	16	4	4
32	1024	80	12,8
256	65536	1024	64
1024	1048576	5120	204,8

2.9 Beispiel: Schnelle Multiplikation von n-Bit-Zahlen

Seien $n \in \mathbb{N}$, $a, b \in \mathbb{N}$, $0 < a, b < 2^n$. Dann haben a und b Dualentwicklungen $a = \sum_{j=0}^{n-1} \alpha_j 2^j$, $b = \sum_{j=0}^{n-1} \beta_j 2^j$, $\alpha_j, \beta_j \in \{0, 1\}$. a und b haben also Dualdarstellungen $\alpha_{n-1} \dots \alpha_0$ und $\beta_{n-1} \dots \beta_0$ der Länge n. Solche Zahlen nennt man deshalb n-Bit-Zahlen.

Problem: Finde schnelle Algorithmen zur Multiplikation zweier n-Bit-Zahlen.

2.9.1 Schulalgorithmus

```
p = 0;
for (j = 0; j < n; j = j + 1) {
        p = p + α_j b;
        b = 2 b;
}
```

Zum Schluß ist $p = a\,b$.

Welche Operationen zählen wir bei der Aufwandsberechnung? Und mit welcher Gewichtung?

- Additionen: $T_{\text{add}}(n)$ sei der Aufwand zur Addition zweier n-Bit-Zahlen.
  Annahme: $T_{\text{add}}(n) \in \Theta(n)$
- Schiebeoperationen: Ist b eine n-Bit-Zahl mit Dualdarstellung $\beta_{n-1} \dots \beta_0$, so hat $2b$ die Dualdarstellung $\beta_{n-1} \dots \beta_0\, 0$, d.h. die Multiplikation mit 2 läßt sich durch eine Schiebeoperation realisieren. $T_{\text{shift}}(n)$ sei der Aufwand zur Verschiebung einer n-Bit-Zahl um eine oder mehrere Stellen nach links.
  Annahme: $T_{\text{shift}}(n) \in \Theta(n)$

2.9.2 Aufwand des Schulalgorithmus

Pro Durchlauf durch die **for**-Schleife wird 1 Schiebeoperation mit Aufwand $T_{\text{shift}}(n+j)$ und 1 Addition mit Aufwand $T_{\text{add}}(n+j)$ ausgeführt. Die Multiplikation mit α_j kostet nur konstante Zeit τ, weil sie durch eine **if**-Abfrage ($\alpha_j == 0$?) erledigt werden kann. Insgesamt beträgt der Aufwand also

$$\sum_{j=0}^{n-1} (T_{\text{shift}}(n+j) + T_{\text{add}}(n+j) + \tau)$$

Es gibt $n_0 \in \mathbb{N}$, $c_1 > 0$, $c_2 > 0$, so daß für $n \geq n_0$, $0 \leq j < n$ gilt

$$c_1(n+j) \leq T_{\text{shift}}(n+j) + T_{\text{add}}(n+j) \leq c_2(n+j)$$

und somit

$$
\begin{aligned}
c_1(n^2 + \frac{n}{2}(n-1)) &= c_1 \sum_{j=0}^{n-1}(n+j) \\
&\leq \sum_{j=0}^{n-1}(T_{\text{shift}}(n+j) + T_{\text{add}}(n+j)) \\
&\leq c_2 \sum_{j=0}^{n-1}(n+j) \\
&= c_2(n^2 + \frac{n}{2}(n-1))
\end{aligned}
$$

Daraus folgt

$$T_{\text{Schulalgorithmus}} \in \Theta(n^2)$$

2.9.3 Ansatz zu einem schnelleren Algorithmus

Sei $n = 2^k$ mit $k \in \mathbb{N}$.
Schreibe $a = a_1 \cdot 2^{\frac{n}{2}} + a_2$, $b = b_1 \cdot 2^{\frac{n}{2}} + b_2$. Die Zahlen a_1, a_2, b_1, b_2 sind $\frac{n}{2}$-Bit-Zahlen mit den Dualdarstellungen $a_1 = \alpha_{n-1} \dots \alpha_{\frac{n}{2}}$, $a_2 = \alpha_{\frac{n}{2}-1} \dots \alpha_0$, $b_1 = \beta_{n-1} \dots \beta_{\frac{n}{2}}$, $b_2 = \beta_{\frac{n}{2}-1} \dots \beta_0$.

Es gilt $a \cdot b = a_1 b_1 \cdot 2^n + (a_1 b_2 + a_2 b_1) \cdot 2^{\frac{n}{2}} + a_2 b_2$.

Das sind

4 Multiplikationen von $\frac{n}{2}$-Bit-Zahlen	$4\, T_{\text{mult}}(\frac{n}{2})$
3 Additionen von Zahlen mit höchstens n Bit	$3\, T_{\text{add}}(n)$
2 Schiebeoperationen von n-Bit-Zahlen	$2\, T_{\text{shift}}(n)$

Nun gilt $a \cdot b = a_1 b_1 \cdot 2^n + ((a_1 + a_2)(b_1 + b_2) - a_2 b_2 - a_1 b_1) \cdot 2^{\frac{n}{2}} + a_2 b_2$. Wären $a_1 + a_2$ und $b_1 + b_2$ $\frac{n}{2}$-Bit-Zahlen, so bräuchte man

- nur 3 Multiplikationen,
- aber 6 Additionen
- und 2 Schiebeoperatioen.

$a_1 + a_2$ und $b_1 + b_2$ können aber $\frac{n}{2} + 1$ Bit lang werden. Setze

$$
\begin{aligned}
a_1 + a_2 &= \alpha \cdot 2^{\frac{n}{2}} + A \\
b_1 + b_2 &= \beta \cdot 2^{\frac{n}{2}} + B
\end{aligned}
$$

wobei α, $\beta \in \{0, 1\}$ und A und B $\frac{n}{2}$-Bit-Zahlen sind.
Es gilt $(a_1 + a_2)(b_1 + b_2) = \alpha\, \beta \cdot 2^n + (\alpha\, B + \beta\, A) \cdot 2^{\frac{n}{2}} + A\, B$.
Das sind

1 Multiplikation von $\frac{n}{2}$-Bit-Zahlen	$T_{\text{mult}}(\frac{n}{2})$
3 Additionen von Zahlen mit höchstens $n+1$ Bit	$T_{\text{add}}(n+1)$
2 Multiplikationen mit 0 oder 1	$\in \Theta(1)$, da durch **if** realisierbar
1 Schiebeoperation von n-Bit-Zahlen	$T_{\text{shift}}(n)$

2.9.4 Skizze eines *Divide-et-Impera*-Algorithmus (Karatsuba-Ofman 1962)

(1) Zerlege a in a_1 und a_2, b in b_1 und b_2.

(2) Berechne α, β, A, B.

(3) Berechne $z_1 = a_1 b_1$, $z_2 = a_2 b_2$, $z_3 = AB$.
Das sind Produkte von $\frac{n}{2}$-Bit-Zahlen. Falls $\frac{n}{2} > 1$, werden sie rekursiv nach diesem Algorithmus berechnet. Falls $\frac{n}{2} = 1$, ist nur eine Multiplikation von zwei Bits auszuführen; dazu kann man eine Tabelle (oder einen Maschinensprachebefehl) benutzen.

(4) Berechne
if $(\alpha == 1$ und $\beta == 1)$ $y = 2^n + (A + B) \cdot 2^{\frac{n}{2}} + z_3$;
else if $(\ \alpha == 1$ und $\beta == 0)$ $y = B \cdot 2^{\frac{n}{2}} + z_3$;
else if $(\ \alpha == 0$ und $\beta == 1)$ $y = A \cdot 2^{\frac{n}{2}} + z_3$;
else $y = z_3$;
$y = y - z_1 - z_2$;

(5) Produkt $= z_1 \cdot 2^n + y \cdot 2^{\frac{n}{2}} + z_2$;

2.9.5 Aufwandsanalyse

(1) 4 Kopieroperationen in $O(n)$.

(2) 2 Additionen von $\frac{n}{2}$-Bit-Zahlen, Bestimmung des Übertrags, Kopieren von zwei $\frac{n}{2}$-Bit-Zahlen. Insgesamt ein Aufwand in $O(n)$.

(3) 3 Multiplikationen von $\frac{n}{2}$-Bit-Zahlen, also $3\,T_{\text{mult}}(\frac{n}{2})$.

(4) Höchstens 1 Addition von $\frac{n}{2}$-Bit-Zahlen, also $\leq T_{\text{add}}(\frac{n}{2})$,
höchstens 2 Additionen von n-Bit-Zahlen, also $\leq 2\,T_{\text{add}}(n)$,
2 Subtraktionen von n-Bit-Zahlen, also $2\,T_{\text{add}}(n)$,
Vergleiche und Addition von 2^n mit einem Aufwand in $O(1)$,
insgesamt ist dies ein Aufwand in $O(n)$.

(5) 2 Schiebeoperationen, also $2\,T_{\text{shift}}(n)$,
2 Additionen, also $2\,T_{\text{add}}(n)$,
also insgesamt ein Aufwand in $O(n)$.

Zusammengefaßt ergibt sich

$$T_{\text{mult}}(n) \in 3\,T_{\text{mult}}(\frac{n}{2}) + O(n)$$

Daher gibt es ein $c > 0$ mit

$$T_{\text{mult}}(n) \leq 3\,T_{\text{mult}}(\frac{n}{2}) + c\,n \quad \text{für } n \in D := \{2^k : k \in \mathbb{N}\},\ T(1) = 1$$

Für die Lösung von $S(n) = 3\,S(\frac{n}{2}) + c\,n,\ n \in D, n > 1,\ S(1) = 1$, gilt nach C.1.2 $S \in O(n^{\log_2 a})$. Mit 2.5.1 folgt daraus

$$T_{\text{mult}} \in O(n^{\log_2 3}) \text{ auf } D$$

Beachte: $O(n^{\log_2 3}) \underset{\neq}{\subset} O(n^2)$ weil $\log_2 3 \approx 1,59 < 2$.
Dieser Multiplikationsalgorithmus ist also asymptotisch schneller als der Schulalgorithmus. Es gibt aber noch bessere (von Schönhage und Strassen 1971), die aber nur theoretische Bedeutung haben.

2.10 Beispiel: Schnelle Matrixmultiplikation

R sei ein Ring (z.B. $\mathbb{R}$), $n \in \mathbb{N}$, $R^{n\times n}$ der Ring der $n \times n$-Matrizen über R.
A, $B \in R^{n\times n}$ seien zwei $n \times n$-Matrizen, $A = (a_{ij})_{i,j}$, $B = (b_{ij})_{i,j}$.
$C = (c_{kl})_{k,l} = A \cdot B \in R^{n\times n}$ ist definiert durch

$$c_k l = \sum_{j=1}^{n} a_{kj} b_{jl} \quad \text{für } k, l \in \{1, \ldots, n\}$$

Zur Berechnung von C nach der Definition benötigt man $n^2 \cdot n = n^3$ skalare Multiplikation (d.h. Multiplikationen in R) und $n^2(n-1)$ skalare Additionen.

Überraschenderweise gibt es Algorithmen zur Berechnung von $A \cdot B$, deren Aufwand asymptotisch langsamer wächst als n^3. Der erste stammt von V. Strassen 1969. Er beruht auf einer *Divide-et-Impera*-Strategie und setzt voraus, daß n eine Zweierpotenz ist.

Sei also $n = 2^k$ mit $k \in \mathbb{N}$.

Man teilt A, B und C jeweils in vier $\frac{n}{2} \times \frac{n}{2}$-Matrizen auf:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \; B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \; C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Man faßt also A, B, C als 2×2-Matrizen über dem Ring $R^{\frac{n}{2}\times\frac{n}{2}}$ auf.

Es gilt

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

Die Multiplikation von A und B wird also auf 8 Multiplikationen von $\frac{n}{2} \times \frac{n}{2}$-Matrizen und 4 Additionen von $\frac{n}{2} \times \frac{n}{2}$-Matrizen, also $4 \cdot (\frac{n}{2})^2$ skalare Additionen zurückgeführt. Sei $T(n)$ die Summe der Anzahl der skalaren Multiplikationen und der Anzahl der skalaren Additionen, die zur Berechnung der Produkts zweier $n \times n$-Matrizen ausgeführt werden, so gilt $T(n) = 8\,T(\frac{n}{2}) + 4 \cdot (\frac{n}{2})^2$.

Berechnet man $A \cdot B$ rekursiv nach obigem Muster mit Abbruchschwelle $n_0 = 1$, also $T(1) = 1$, so wird der Rechenaufwand durch obige Rekursionsgleichung gegeben. Nach C.1.2 (mit $a = 8$, $b = 2$, $l = 2$, $a > b^l$) folgt $T \in O(n^2)$ auf $D = \{2^k : k \in \mathbb{N}\}$ (denn $\log_b a = \log_2 8 = 3$). Also keine Verbesserung gegenüber der direkten Berechnung!

Was muß man ändern, um einen asymptotisch langsamer wachsenden Aufwand zu erhalten?

Betrachte die Rekursionsgleichung

$$T(n) = a\,T(\tfrac{n}{2}) + c\,(\tfrac{n}{2})^2$$

Wir gehen davon aus, daß $a > 2^2 = 4$. Dann ist nach C.1.2 $T \in \Theta(n^{\log_2 a})$. Und es gilt $\log_2 a < 3 \iff a < 8$. Man müßte also den Rekursionsschritt so modifizieren, daß weniger als acht $\frac{n}{2} \times \frac{n}{2}$-Matrizen zu multiplizieren sind.

2.10.1 Satz (Strassen) Das Produkt von zwei 2×2-Matrizen über einem Ring M kann durch 7 Multiplikationen und 18 Additionen (einschließlich Subtraktionen) berechnet werden.

Beweis: Sei

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$\begin{aligned}
X_1 &:= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
X_2 &:= (A_{21} + A_{22}) \cdot B_{11} \\
X_3 &:= A_{11} \cdot (B_{21} - B_{22}) \\
X_4 &:= A_{22} \cdot (B_{21} - B_{11}) \\
X_5 &:= (A_{11} + A_{12}) \cdot B_{22} \\
X_6 &:= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
X_7 &:= (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
\end{aligned}$$

Damit gilt

$$\begin{aligned}
C_{11} &= X_1 + X_4 - X_5 + X_7 \\
C_{12} &= X_3 + X_5 \\
C_{21} &= X_2 + X_4 \\
C_{22} &= X_1 + X_3 - X_2 + X_6
\end{aligned}$$

q.e.d.

Wendet man diesen Satz mit $M = R^{\frac{n}{2} \times \frac{n}{2}}$ an, so hat man die Multiplikation von $A, B \in R^{n \times n}$ auf 7 Multiplikationen und 18 Additionen in $R^{\frac{n}{2} \times \frac{n}{2}}$ zurückgeführt. Dies verwendet man als Rekursionsschritt eines rekursiven Algorithmus mit Abbruchschwelle $n_0 = 1$. Sein Aufwand erfüllt die Rekursionsgleichung

$$T(n) = 7\,T(\tfrac{n}{2}) + 18 \cdot (\tfrac{n}{2})^2,\ n \in D = \{2^k : k \in \mathbb{N}\},\ T(1) = 1$$

Nach C.1.2 (mit $a = 7$, $b = 2$, $l = 2$, $a > b^l$) gilt

$$T \in \Theta(n^{\log_2 7}) \subsetneq O(n^3)$$

weil $\log_2 7 \approx 2{,}81 < 3$.

2.10.2 Bemerkungen

1. Der Algorithmus ist für die Praxis nicht von Bedeutung, weil die multiplikative Konstante zu groß ist; erst ab $n = 128$ ist Strassens Algorithmus schneller als die direkte Berechnung.
2. Mit anderen Methoden kann man Algorithmen konstruieren, die asymptotisch noch schneller sind: $T \in O(n^{2{,}376})$ (Coppersmith, Winograd 1987).

2.11 Beispiel: Minimalabstand einer endlichen Punktmenge in der Ebene

2.11.1 Aufgabe: Gegeben sei eine endliche Punktmenge $P \subset \mathbb{R}^2$. Bestimme ihren Minimalabstand $d(P) = \min\{||\, p - q\, ||\colon p, q \in P, p \neq q\}$. Dabei sei $||\;||$ die euklidische Norm.

Sei $P = \{p_1, \ldots, p_n\}$ und $p_j = (x_j, y_j)$, $p_i \neq p_j$ für $i \neq j$, also $\#P = n$, $n \geq 2$.

2.11.2 Naiver Algorithmus

(1) Berechne für jedes Paar $(i, j) \in \{1, \ldots, n\}^2$ mit $i < j$ den Abstand $d_{ij} = ||\, p_i - p_j\, || = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.
(2) Bestimme $d = \min\{d_{ij}\colon i < j\}$.

Aufwand: Wir zählen nur Abstandsberechnungen zwischen Punkten im $\mathbb{R}^2$. Es werden $\frac{1}{2}n(n-1)$ Abstandsberechnungen ausgeführt. Somit ist der Aufwand in $\Theta(n^2)$.

2.11.3 *Divide-et-Impera*-Ansatz

Zerlege P in zwei möglichst gleiche Teile P_1 und P_2. Kennt man die Minimalabstände $d(P_1)$ und $d(P_2)$, so kann man $d(P)$ folgendermaßen berechnen:

$$d(P) = \min\{d(P_1),\, d(P_2),\, d(P_1, P_2)\}$$

wobei $d(P_1, P_2) = \min\{||\, p - q\, ||\colon p \in P_1,\, q \in P_2\}$.

Zur Berechnung von $d(P_1, P_2)$ muß man $(\frac{n}{2})^2$ Abstandsberechnungen ausführen. Die Idee ist nun, aus den vielen möglichen Zerlegungen von P in zwei Hälften eine auszuwählen, welche die Berechnung von $d(P_1, P_2)$ mit geringerem Aufwand ermöglicht.

Wir nehmen an, daß die $p_1, \ldots, p_n$ so numeriert sind, daß die Folge $x_1, \ldots, x_n$ ihrer Abszissen monoton wächst. Das läßt sich durch ein Sortierverfahren wie z.B. Merge-Sort mit einem Aufwand in $O(n \log n)$ erreichen.

Wähle ein $x_0 \in \mathbb{R}$, so daß $\#\{x_j\colon x_j < x_0\} \leq \lceil \frac{n}{2} \rceil$ und $\#\{x_j\colon x_j > x_0\} \leq \lceil \frac{n}{2} \rceil$ (z.B. $x_0 = x_{\lfloor \frac{n}{2} \rfloor}$).

P_1 enthalte alle p_j mit $x_j < x_0$, P_2 alle p_j mit $x_j > x_0$. Die p_j mit $x_j = x_0$ werden so auf P_1 und P_2 verteilt, daß $|\#P_1 - \#P_2\,| \leq 1$ gilt.

Sei $\delta = \min\{d(P_1),\, d(P_2)\}$.

Um $d(P)$ zu berechnen, braucht man nur noch Punktepaare $(p, q) \in P_1 \times P_2$ mit $||\, p - q\, || < \delta$ zu betrachten, falls es welche gibt. Solche Punkte liegen in dem Streifen

$$S = \{(x, y) \in \mathbb{R}^2\colon |\, x - x_0\, | < \delta\}$$

Da aber $P_1 \times P_2$ ganz in diesem Streifen liegen kann, erhält man nicht unbedingt eine Reduktion des Aufwands. Man muß noch eine weitere geometrische Betrachtung anschließen.

2.11.4 Lemma Für jedes $y_0 \in \mathbb{R}$ enthalten sowohl $P_1 \cap (]x_0 - \delta,\, x_0] \times]y_0 - \delta,\, y_0 + \delta[)$ als auch $P_2 \cap (]x_0,\, x_0 + \delta] \times]y_0 - \delta,\, y_0 + \delta[)$ höchstens 10 Punkte.

Beweis: Wegen $d(P_1) \geq \delta$ gilt $||\, p - q\, || \geq \delta$ für alle $p, q \in P_1$, $p \neq q$. Daher gibt es zu jedem y höchstens einen Punkt in $P_1 \cap S$ mit Ordinate y.

Seien $R =]x_0 - \delta, x_0] \times]y_0 - \delta, y_0 + \delta[$ und $P_1 \cap R = \{q_1, \ldots, q_k\}$. Die offene Kreisscheibe im $\mathbb{R}^2$ mit Mittelpunkt q_j und Radius $\frac{\delta}{2}$ sei mit $K(q_j, \frac{\delta}{2})$ bezeichnet. Diese Kreisscheiben sind disjunkt, weil ihre Mittelpunkte ja mindestens den Abstand δ voneinander haben. Außerdem hat der Durchschnitt so einer Kreisscheibe mit dem Rechteck R einen Flächeninhalt, der mindestens ein Viertel der Kreisscheibenfläche beträgt. Damit folgt

$$\begin{aligned} 2\,\delta^2 &= \text{Fläche von } R \\ &\geq \text{Fläche von } \bigcup_{j=1}^{k} K(q_j, \frac{\delta}{2}) \cap R \\ &= \sum_{j=1}^{k} \text{Fläche von } K(q_j, \tfrac{n}{2}) \cap R \\ &\geq \sum_{j=1}^{k} \tfrac{1}{4}\pi\tfrac{\delta^2}{4} \\ &= k\pi\tfrac{\delta^2}{16} \end{aligned}$$

also $k \leq \lfloor \frac{32}{\pi} \rfloor \leq 10$.
Entsprechend folgt die Behauptung für P_2. q.e.d.

2.11.5 Folgerung Ist $p = (x_p, y_q) \in P_1 \cap S$, so gibt es höchstens 10 Punkte $q = (x_q, y_q) \in P_2 \cap S$ mit $|\,y_p - y_q| < \delta$. Bei der Berechnung von $d(P)$ braucht man $||\,p - q\,||$ nur für diese höchstens zehn Punkte q zu bestimmen.

Wie findet man diese Punkte q schnell?
Indem man die Punkte in $P \cap S$ so umsortiert, daß ihre Ordinaten monoton wachsen. Sei nämlich $P \cap S = \{q_1, \ldots, q_l\}$, $q_j = (u_j, v_j)$, $v_1 < \ldots < v_l$. Zu q_j können dann höchstens die Punkte $q_{j-19}, \ldots, q_{j-1}, q_{j+1}, \ldots, q_{j+19}$ einen Abstand kleiner als δ haben.

Somit erhält man folgenden rekursiven Algorithmus.

2.11.6 Erster Algorithmus

1. Schritt: Sortiere die Punkte von P als Folge $p_1, \ldots, p_n$ mit wachsender Abszisse.

2. Schritt: Berechne $d(P)$ rekursiv auf folgende Weise:
Falls $\#P = 1$, setze $d(P) = 0$; ansonsten

- teile P wie oben beschrieben durch eine senkrechte Trennlinie in zwei möglichst gleiche Teile P_1 und P_2;
- berechne (rekursiv) $d(P_1)$ und $d(P_2$;
- setze $\delta = \min\{d(P-1), d(P_2)\}$ und $S =$ Streifen der Breite 2δ um die Trennlinie;
- sortiere die Punkte in $P \cap S$ nach wachsender Ordinate;
- gehe diese Punkte der Reihe nach durch und berechne jeweils ihren Abstand zu den 9 nächsten Nachfolgern; falls dabei ein Abstand $\alpha < \delta$ auftritt, ersetze den Wert von δ durch den von α;

- gib den Wert δ zurück.

Aufwandanalyse

Sortieren nach wachsenden Abszissen	$O(n \log n)$
Sortieren nach wachsenden Ordinaten	$O(n \log n)$
Durchlauf durch $P \cap S$	$O(n)$

Also ergibt sich für den Aufwand T die asymptotische Rekursionsgleichung

$$T(n) \in 2\,T(\lceil \tfrac{n}{2} \rceil) + O(n \log n), \quad T(1) = 0$$

Mit 2.5.1 und C.2.5 folgt

$$T \in O(n\,(\log n)^2)$$

2.11.7 Zweiter Algorithmus

Man kann den obigen Algorithmus beschleunigen, indem man bei jedem rekursiven Aufruf nicht nur den Minimalabstand zurückgibt, sondern auch die nach wachsender Ordinate sortierten Punkte. Dazu muß man lediglich die derart sortierten Punkte von P_1 und P_2 zusammenmischen wie bei MergeSort, was mit einem Aufwand in $O(n)$ geht (statt $O(n \log n)$ für das komplette Sortieren wie in obigem Algorithmus). Für den Aufwand erhält man dann die Rekursionsgleichung

$$T(n) \in 2\,T(\lceil \tfrac{n}{2} \rceil) + O(n), \quad T(1) = 0$$

Nach 2.5.1 und C.2.4 gilt

$$T \in O(n \log n)$$

Kapitel 3

Sortieren

3.1 Grundbegriffe

3.1.1 Aufgabenstellung beim Sortieren

Gegeben sei eine Folge von Datensätzen $D_1, \ldots, D_n$ (z.B. als Structs oder Records) und eine geordnete Menge S mit der Ordnungsrelation $\leq$. Zu jedem Datensatz D_j gehöre ein sogenannter Schlüssel $a_j \in S$ (der meist in einer Komponente des Datensatzes D_j abgespeichert ist).
Gesucht ist eine monoton wachsende Anordnung $a_{\pi(1)} \leq \ldots \leq a_{\pi(n)}$ der Schlüssel $a_1, \ldots, a_n$, wobei π eine Permutation von $\{1, \ldots, n\}$ ist.
Man nennt dann $D_{\pi(1)}, \ldots, D_{\pi(n)}$ eine nach den Schlüsseln a_j sortierte Folge.

3.1.2 Bemerkungen

1. Typische Beispiele für die Schlüsselmenge S sind $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ oder A^* d.h die Menege der Wörter über einem Alphabet A; dabei ist auf A irgendeine Ordnung gegeben, und auf S wird die induzierte lexikografische Ordnung genommen.

2. Für die prinzipielle Lösung der Sortierverfahren spielt die spezielle Gestalt der Datensätze und Schlüssel keine Rolle. Man kann daher (wie wir es bisher auch getan haben) o.B.d.A $S = \mathbb{N}$ annehmen und den sonstigen Inhalt der Datensätze vernachlässigen. In der Praxis muß man jedoch auf die Größe der Datensätze und die Art der Schlüssel Rücksicht nehmen, weil sie die Zeit der Kopier- und Vergleichsoperationen beeinflussen. Man wird oft mit Zeigern auf die Datensätze und nicht mit den Datensätzen selbst arbeiten.

3. Nach IBM-Angaben wird 25% der Rechenzeit in komerziellen Systemen für Sortieren verwendet. Das liegt daran, daß Sortieren als Basisoperation in vielen Algorithmen vorkommt; insbesondere kann die Suche von Datensätzen durch vorheriges Sortieren wesentlich beschleunigt werden.

3.1.3 Eigenschaften von Sortierverfahren

Interne Sortierverfahren sind darauf ausgelegt, daß sämtliche Datensätze im Arbeitsspeicher (RAM) des Rechners liegen, also insbesondere schnell in beliebiger Reihenfolge auf die Datensätze zugegriffen werden kann.

Externe Sortierverfahren sind darauf ausgelegt, daß die Datensätze auf externen Datenträgern (Magnetbändern, Festplatten, Disketten) liegen, die nur sequentiell oder blockweise gelesen werden können.

Ein Sortierverfahren heißt **am Ort** (in place, in situ), wenn neben dem Speicherbedarf für die Eingabe nur noch ein konstanter d.h. von der Größe der Eingabe unabhängiger Speicherplatz benötigt wird.

Ein Sortierverfahren heißt **stabil**, wenn die relative Position der Datensätze mit gleichem Schlüssel erhalten bleibt d.h. wenn gilt: $a_i = a_j$ und $i < j \Longrightarrow \pi(i) < \pi(j)$.

3.2 Überblick über Sortierverfahren

Den **Rechenaufwand** von Sortierverfahren mißt man meist mit Hilfe der Anzahl von Schlüsselvergleichen und/oder der Anzahl von Kopieroperationen von Datensätzen (bzw. von Zeigern auf Datensätze). In der Praxis kann der wirkliche Zeitaufwand stark schwanken, z.B. ist der Vergleich von Strings viel aufwendiger als der von Integers.

Da meist eine große Zahl von Datensätzen sortiert werden soll, ist der Speicherbedarf des Verfahrens wichtig, insbesondere ob es am Ort arbeitet.

Der Berechnung des **mittleren Aufwandes** legt man meist folgende Voraussetzungen zugrunde:
1. Die Schlüssel $a_1, \ldots, a_n$ der zu sortierenden Datensätze sind paarweise verschieden.
2. Alle Anordnungen $a_{\pi(1)}, \ldots, a_{\pi(n)}$ mit $\pi \in \mathfrak{S}_n$ = Gruppe der Permutationen von $\{1, \ldots, n\}$, sind gleichwahrscheinlich; man berechnet also den arithmetischen Mittelwert $\frac{1}{n!} \sum_{\pi \in \mathfrak{S}_n} T(a_{\pi(1)}, \ldots, a_{\pi(n)})$.

Einfache interne Sortierverfahren

Sortieren durch Auswahl (Selection Sort)
Sortieren durch Einfügen (Insertion Sort)
Sortieren durch Austausch, speziell Bubble Sort

Alle diese Verfahren arbeiten am Ort und sind für kleine n recht schnell. Asymptotisch ist ihr Aufwand aber in $\Theta(n^2)$.

Höhere interne Sortierverfahren

	Bester Fall	Schlechtester Fall	im Mittel	
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
Quick Sort	$\Omega(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$	
Heap Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	am Ort

Die obigen Sortierverfahren werden allgemeine Sortierverfahren genannt, weil sie über die Schlüsselmenge keine besondere Voraussetzungen machen und Vergleiche gemäß der Ordnungsrelation benutzen. Für kleinere Schlüsselmengen kann man Sortierverfahren verwenden, die auf der Verteilung der Schlüssel auf vorgegebene Fächer beruhen. Dazu gehören *Bucket Sort* und *Radix Sort*. Ihr Rechenaufwand wächst nur linear mit n. Hat die Menge aller möglichen Schlüssel höchstens m Elemente, so ist der Aufwand von Bucket Sort asymptotisch durch $c \cdot (n + m)$ nach oben beschränkt, der von Radix Sort durch $c \cdot n \log m$ für ein $c > 0$.

Für **externes Sortieren** eignen sich Varianten von Merge Sort.

Fazit: Es gibt kein schlechthin bestes Sortierverfahren. Die jeweilige Anwendungssituation ist ausschlaggebend für die Wahl.

Die in Θ verborgenen multiplikativen Konstanten sind bei Merge Sort, Quick Sort und Heap Sort recht klein, so daß die Verfahren schon für kleine n (d.h. etwa $>$ 10 schneller als die einfachen Sortierverfahren sind. Im Mittel ist Quick Sort am schnellsten. Sind jedoch in einer Anwendung die Eingaben nicht gleichwahrscheinlich, so kann Quick Sort langsam werden; so mag Quick Sort in seiner Urform z.B. keine teilweise vorsortierten Eingaben.

In der Praxis will man meist eine Folge von Datensätzen $D_1, \ldots, D_n$ sortieren, die durch `structs` dargestellt werden, in denen eine Komponente den Schlüssel a_j enthält, also $a_j = D_j$.`Schluessel`. Weil der sonstige Inhalt der D_j beim Sortieren keine Rolle spielt, werden wir lediglich die Folge der Schlüssel $a_1, \ldots, a_n$ sortieren. Wir gehen davon aus, daß $a_1, \ldots, a_n$ in einem Array `A[1...n]` abgelegt ist und verwenden Arrays mit Indexbereichen, die nicht notwendig bei `1` anfangen. In praktischen Implementierungen können auch andere Datenstrukturen benutzt werden.

3.3 Einfache Sortierverfahren

3.3.1 Sortieren durch Auswahl

```
SelectionSort(A[1...n]) {
    for (i = 1; i < n; i = i + 1) {
        for (j = i + 1; j ≤ n; j = j + 1) {
            if (A[j] < A[i]) {vertausche A[i] und A[j]; }
        }
    }
}
```

3.3.2 Sortieren durch Einfügen

```
InsertionSort( A[1...n] ) {
    for (i = 2; i < n; i = i + 1) {
        j = i;
        a = A[i];
        while (A[j − 1] > a und j ≥ 2) {
            A[j] = A[j − 1];
            j = j − 1;
        }
        A[j] = a;
    }
}
```

In der **while**-Schleife ist $A[1 \ldots i-1]$ bereits sortiert, und $A[i]$ wird so in $A[1 \ldots i]$ eingefügt, daß $A[1 \ldots i]$ sortiert ist. Die rechts von der neuen Position von $A[i]$ stehenden

Elemente werden um eine Position nach rechts verschoben.

3.3.3 Sortieren durch Austausch

Idee: Durchlaufe die Schlüsselfolge $a_1, \ldots, a_n$ und vertausche dabei a_{j-1} und a_j, wenn $a_{j-1} > a_j$. Wiederhole dies, bis kein Austausch mehr möglich ist.

```
BubbleSort1( A[1...n] ) {
    do {
        ausgetauscht = nein;
        for (i = 2; i ≤ n; i = i + 1) {
            if ( A[i − 1] > A[i] ) {
                vertausche A[i − 1] und A[i];
                ausgetauscht = ja;
            }
        }
    } while (ausgetauscht == ja);
}
```

Beweis der Korrektheit: Wenn der Algorithmus terminiert, wurde kein Austausch mehr vorgenommen, und das bedeutet gerade, daß das Array sortiert ist. Es ist also nur zu zeigen, daß der Algorithmus terminiert.
Beachte:
Nach dem 1-ten Durchlauf durch die **while**-Schleife steht in $A[n]$ das Maximum von $A[1 \ldots n]$.
Nach dem 2-ten Durchlauf durch die **while**-Schleife steht in $A[n-1]$ das Maximum von $A[1 \ldots n-1]$.
usw. Nach dem $(n-1)$-ten Durchlauf durch die **while**-Schleife steht in $A[2]$ das Maximum von $A[1 \ldots 2]$.
Somit wird spätestens beim n-ten Durchlauf kein Austausch mehr vorgenommen, und der Algorithmus terminiert. q.e.d.

Aufgrund obiger Beobachtung, daß nach dem j-ten Durchlauf das Array $A[n-j+1 \ldots n]$ bereits die j größten Elemente in sortierter Reihenfolge enthält, kann man den Algorithmus ein wenig effizienter machen.

```
BubbleSort2( A[1...n] ) {
    for (j = n; j ≥ 2; j = j − 1) {
        for (i = 2; i ≤ j; i = i + 1)
            if ( A[i − 1] > A[i] ) vertausche A[i − 1] und A[i];
        }
    }
}
```

Durch Verwendung einer Variablen *ausgetauscht* wie in BubbleSort1 kann das Verhalten im besten Fall verbessert werden.

3.3.4 Rechenaufwand

Bei den obigen einfachen Sortierverfahren liegt die Anzahl der benötigten Vergleiche im schlechtesten und im mittleren Fall stets in $\Theta(n^2)$. Im besten Fall brauchen manche weniger Vergleiche.

3.3.5 Bemerkung Die obigen einfachen Sortierverfahren arbeiten alle **am Ort**. Für kleine n (etwa < 20) sind sie den im Folgenden aufgeführten komplexeren Sortierverfahren vorzuziehen.

3.4 Merge Sort

Sortieren durch Verschmelzen (oder Sortieren durch Mischen) haben wir schon im ersten Kapitel untersucht. Die prinzipielle Gestalt ist folgende:

MergeSort($A[m \dots k]$) {
  if($m < k$) {
    $n = k - m + 1$;
    $l = \lfloor \frac{n}{2} - 1 \rfloor$;
    MergeSort($A[m \dots m + l]$);
    MergeSort($A[m + l + 1 \dots k]$);
    verschmelze $A[m \dots m + l]$ und $A[m + l + 1 \dots k]$
    zu dem sortierten Array $A[m \dots k]$;
  }
}

3.4.1 Rechenaufwand

In 1.3.5 haben wir schon gesehen, daß das Verschmelzen mit $n - 1$ Vergleichen geht, aber nicht mit weniger. Deshalb gilt für die Anzahl der Vergleiche

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + n - 1, \quad T(1) = 0$$

und zwar unabhängig von den Werten der Glieder der zu sortierenden Folge $a_1, \dots, a_n$. Lediglich ihre Länge spielt eine Rolle. Diese Rekursionsgleichung beschreibt also die Rechenzeit sowohl im besten als auch im schlechtesten Fall.

In 1.3.5.4 haben wir die Rekursionsgleichung bereits explizit gelöst. Danach gilt

$$T_{\text{best}} = T_{\text{worst}} = T_{\text{average}} \in O(n \log n)$$

3.4.2 Speicherbedarf

Beim Verschmelzen wird ein Hilfsarray benötigt, dessen Länge die Summe der Längen der beiden Teilarrays ist; im schlechtesten Fall also n. Außerdem wird für die Rekursion ein Stapel der Tiefe $O(\log n)$ benötigt.

Die obige Form von Merge Sort arbeitet also nicht am Ort, sondern benötigt $\geq 2n + \log n$ Speicher.

3.4.3 Bemerkungen

1. Es gibt Modifikationen von Merge Sort, die mit $n + const$ viel Speicher auskommen (siehe Zitate in dem Buch von *Ottmann, Widmayer*).

2. Man kann Merge Sort auch rein iterativ formulieren. Dazu fängt man "unten an (bottom up) d.h. man betrachtet die a_i als 1-gliedrige Folgen und verschmilzt jeweils a_{2i-1} und a_{2i} zu einer 2-gliedrigen Folge. Diese 2-gliedrigen Folgen werden dann wiederum zu 4-gliedrigen Folgen verschmolzen usw.

3. Der Verschmelzungsschritt in Merge Sort kann leicht so entworfen werden, daß Merge Sort **stabil** ist.

3.5 Quick Sort

Quick Sort wurde 1962 von C.A.R. Hoare entwickelt. Es beruht wie Merge Sort auf einer *Divide-et-Impera*-Strategie. Der Unterschied besteht darin, daß die Zerlegung in zwei Teile aufwendiger gestaltet wird und dadurch das Verschmelzen der sortierten Teilfolgen trivial wird.

3.5.1 Idee

(1) Zerlege die Schlüsselfolge $a_1, \ldots, a_n$ in zwei Folgen $b_1, \ldots, b_{n_1}$ und $c_1, \ldots, c_{n_2}$, so daß $b_i \leq c_j$ für alle $i \in \{1, \ldots, n_1\}$ und $j \in \{1, \ldots, n_2\}$.

(2) Sortiere $b_1, \ldots, b_{n_1}$ und $c_1, \ldots, c_{n_2}$.

(3) $b_1, \ldots, b_{n_1}, c_1, \ldots, c_{n_2}$ ist dann eine sortierte Permutation von $a_1, \ldots, a_n$ d.h. das Verschmelzen besteht nur aus einem Aneinanderhängen.

Wesentlich ist der Zerlegungsschritt (1). Er wird bei Quick Sort typischerweise folgendermaßen durchgeführt:

- Wähle ein $p \in \{1, \ldots, n\}$.
- Teile $a_1, \ldots, a_n$ in zwei Teilfolgen auf; die erste bestehe aus allen a_i mit $a_i \leq a_p$, die zweite aus allen a_i mit $a_i > a_p$.

a_p wird **Pivotelement** genannt. Die verschiedenen Varianten von Quick Sort unterscheiden sich vor allem durch unterschiedliche Wahlen des Pivotelements. Wir werden im Folgenden a_1 als Pivatelement wählen. Es gibt aber andere heuristische Wahlen, die in manchen Situationen vorteilhafter sind. Ideal wäre der Median; leider muß für eine effiziente Berechnung des Medians die Folge bereits vorher sortiert werden.

Ist die Schlüsselfolge $a_1, \ldots, a_n$ in einem Array $A[1 \ldots n]$ gespeichert, so kann man dieses Array so umordnen, daß die erste Teilfolge in $A[1 \ldots k-1]$ und die zweite in $A[k+1 \ldots n]$ und a_p in $A[k]$ stehen.

3.5.2 Zerlegungschritt

```
Zerlege( A[1...n] ) {
    i = 1;
    j = n + 1;
    pivot = A[1];
    while ( i < j ) {
        do i = i + 1 while ( A[i] ≤ pivot und i < j );
        do j = j − 1 while ( A[j] > pivot und i < j );
        if ( i < j ) vertausche A[i] und A[j];
```

```
        }
        vertausche A[1] und A[j];
}
```

3.5.3 Bemerkungen

1. Bei Beendigung der **while**-Schleife ist $i = j$ oder $i = j + 1$, also $j - i = 0$ oder $j - i = -1$. Da am Anfang $j - i = n$ war, werden insgesamt n oder $n + 1$ Vergleiche ausgeführt.

2. Außer dem Array wird kein zusätzlicher Speicher benötigt.

3. Obiger Zerlegungsschritt erhält nicht die relative Position gleicher Schlüssel.

4. Die Abfragen, ob $i < j$ ist, in den beiden **do**-Schleifen, dienen nur dazu, daß der Indexbereich des Arrays nicht überschritten wird. Wenn z.B. $A[k] < pivot$ für alle $k \in \{2, \ldots, n\}$, so würde die erste **do**-Schleife nicht enden. Statt $i < j$ kann man in der ersten **do**-Schleife auch $i < n$ und in der zweiten $j > 1$ prüfen. Man kann die Abfragen sogar ganz weglassen, wenn man das Array um ein Element $A[0]$ und ein Element $A[n + 1]$ so erweitert, daß $A[0] < A[k] < A[n + 1]$ für alle $k \in \{1, \ldots, n\}$.

3.5.4 Quick Sort

Der gesamte Algorithmus lautet nun folgendermaßen.

```
        QuickSort( A[l...r] ) {
(1)         i = l;
(2)         j = r + 1;
(3)         pivot = A[l];
(4)         while ( i < j ) {
(5)             do i = i + 1 while ( A[i] ≤ pivot und i < j );
(6)             do j = j − 1 while ( A[j] > pivot und i < j );
(7)             if ( i < j ) vertausche A[i] und A[j];
            }
(8)         vertausche A[l] und A[j];
(9)         if ( l < j − 1 ) QuickSort( A[l...j − 1] );
(10)        if ( j + 1 < r ) QuickSort( A[j + 1...r] );
        }
```

3.5.5 Aufwandsanalyse

Die Anzahl der Operationen in den Zeilen (1)-(3) ist in $O(1)$. Sei $n = r - l + 1$. In den Zeilen (5) und (6) werden n oder $n+1$ Vergleiche zwischen Schlüsseln ausgeführt; die Anzahl der anderen Operationen liegt in O(Anzahl der Vergleiche). Die Anzahl der Operationen in Zeile (7) liegt in $O(1)$. Die **while**-Schleife wird höchstens so oft durchlaufen, wie Vergleiche zwischen Schlüsseln ausgeführt werden.

Die Anzahl aller Operationen liegt also in O(Anzahl der Vergleiche zwischen Schlüsseln). Sei $T(A[l \ldots r])$ die Anzahl der Vergleiche von Schlüsseln. Dann gilt

$$T(A[l \ldots r]) \leq n + 1 + T(A[l \ldots j - 1]) + T(A[j + 1 \ldots r])$$

3.5.5.1 Schlechtester Fall

Sei $T_{\text{worst}}(n)$ die Anzahl von Vergleichen zwischen Schlüsseln, die QuickSort bei Eingabe eines Arrays $A[1 \dots n]$ maximal ausführt. Dann gilt

$$T_{\text{worst}}(n) \leq n + 1 + \max\{T_{\text{worst}}(j-1) + T_{\text{worst}}(n-j) \colon 1 \leq j \leq n\} \text{ für } n \geq 2$$
$$T_{\text{worst}}(0) = T_{\text{worst}}(1) = 0$$

Lemma Es gilt $T_{\text{worst}}(n) \leq \frac{1}{2}(n+1)(n+2) - 3$ für $n \geq 2$.

Beweis: Vollständige Induktion nach n.

$n = 2$: $T_{\text{worst}}(2) \leq 3 + \max\{0, 0\} = 3 = \frac{1}{2} \cdot 3 \cdot 4 - 3$.

Induktionsschluß: $n - 1 \mapsto n$

Induktionsvoraussetzung: Für $1 \leq j \leq n$ gilt

$$\begin{aligned}
T_{\text{worst}}(j-1) &\leq \tfrac{1}{2}j(j+1) - 3 = \tfrac{1}{2}(j^2 + j) - 3 \\
T_{\text{worst}}(n-j) &\leq \tfrac{1}{2}(n-j+1)(n-j+2) - 3 \\
&= \tfrac{1}{2}(n^2 - nj + 2n - nj + j^2 - 2j + n - j + 2) - 3 \\
T_{\text{worst}}(j-1) + T_{\text{worst}}(n-j) &\leq \tfrac{1}{2}(n^2 + 3n + 2 + 2j^2 - 2j - 2nj) - 6 \\
&\leq \tfrac{1}{2}(n^2 + n + 2) - 6
\end{aligned}$$

weil $2j^2 - 2j - 2nj \leq -2n$ für $1 \leq j \leq n$.
(Begründung: Die Funktion $f(x) = 2x^2 - 2x - 2nx$ ist konvex, da $f''(x) = 4 > 0$ für alle x, und es gilt $f(1) = -2n$ und $f(n) = -2n$.)
Damit folgt jetzt die Behauptung des Induktionsschlusses:

$$\begin{aligned}
T_{\text{worst}}(n) &\leq n + 1 + \frac{1}{2}(n^2 + n + 2) - 6 \\
&= \tfrac{1}{2}(n^2 + 3n + 2) - 5 \\
&= \tfrac{1}{2}(n+1)n + 2) - 3
\end{aligned}$$

q.e.d.

Folgerung $T_{\text{worst}} \in O(n^2)$.

Lemma Es gilt auch $T_{\text{worst}} \in \Omega(n^2)$.

Beweis: Sei $A[1] < A[2] < \dots < A[n]$. Mit dem Pivotelement $A[1]$ hat bei jedem Rekursionsschritt das erste Teilintervall $A[l \dots j]$ nur die Länge 1. Daher gilt

$$\begin{aligned}
T(A[1 \dots n]) &\geq n + T(1) + T(A[2 \dots n]) \\
&\geq n + (n-1) + T(A[3 \dots n]) \\
&\geq \dots \\
&\geq \sum_{k=1}^{n} k \\
&= \tfrac{1}{2}n(n+1) \\
&\in \Omega(n^2)
\end{aligned}$$

Fazit: $$T_{\text{worst}} \in \Theta(n^2)$$

Achtung! Falls als Pivotelement stets das erste Element gewählt wird, hat QuickSort quadratische Laufzeit, wenn die Datenfolge bereits sortiert ist. Deshalb gibt es andere Strategien für die Wahl des Pivotelementes, die aber dann in anderen Fällen zu quadratischer Laufzeit führen.

3.5.5.2 Verhalten im Mittel

$a_1, \ldots, a_n$ sei eine Folge paarweise verschiedener Schlüssel.

Die Anzahl der Vergleiche $T(a_1, \ldots, a_n)$, die QuickSort beim Sortieren der Folge $a_1, \ldots, a_n$ ausführt, hängt nur von den Ordnungsrelationen zwischen den a_j und nicht von ihren Werten ab. Daher hängt der Mittelwert

$$T_{\text{mittel}}(n) = \tfrac{1}{n!} \textstyle\sum_{\sigma \in \mathfrak{S}_n} T(a_{\sigma(1)}, \ldots, a_{\sigma(n)})$$

nicht von der speziellen Folge $a_1, \ldots, a_n$ ab. Dabei sei $\mathfrak{S}$ die Menge der Permutationen von $\{1, \ldots, n\}$.

Wir können daher o.B.d.A. $a_1 < \ldots < a_n$ voraussetzen.

Es gilt

$$T_{\text{mittel}}(n) = \frac{1}{n} \sum_{j=1}^{n} \frac{1}{(n-1)!} \sum_{\sigma \in \mathfrak{S}_n, \sigma(1)=j} T(a_j, a_{\sigma(2)}, \ldots, a_{\sigma(n)})$$

Wählt man als Pivotelement das erste Element, so wird die Folge $a_{\sigma(2)}, \ldots, a_{\sigma(n)}$ in eine Folge $A_<$ der Länge $j-1$ und eine Folge $A_>$ der Länge $n-j$ zerlegt. $A_<$ ist eine Umordnung von $a_1, \ldots, a_{j-1}$, und $A_>$ ist eine Umordnung von $a_{j+1}, \ldots, a_n$.

Zwei Permutationen $\sigma_1, \sigma_2 \in \mathfrak{S}_n$ liefern dieselbe Folge $A_<$, wenn $\sigma_1(1) = \sigma_2(1) = j$ und σ_1 und σ_2 auf $\{2, \ldots, j\}$ übereinstimmen. Es gibt $\binom{n-1}{j-1}(n-j)!$ solcher Permutationen; denn es gibt $\binom{n-1}{j-1}$ $(j-1)$-elementige Teilmengen $I \subset \{1, \ldots, n\} \setminus \{j\}$ und $(n-1-(j-1))! = (n-j)!$ bijektive Abbildungen von $\{j+1, \ldots, n\}$ auf $\{1, \ldots, n\} \setminus (I \cup \{j\})$.

Entsprechend gibt es $\binom{n-1}{n-j}(j-1)!$ Permutationen, die dieselbe Folge $A_>$ liefern. Somit folgt

$$\begin{aligned}
&\frac{1}{(n-1)!} \sum_{\sigma \in \mathfrak{S}_n, \sigma(1)=j} T(a_j, a_{\sigma(2)}, \ldots, a_{\sigma(n)}) \\
&= n + 1 + \frac{1}{(n-1)!}\binom{n-1}{j-1}(n-j)! \sum_{A_<} T(A_<) + \frac{1}{(n-1)!}\binom{n-1}{n-j}(j-1)! \sum_{A_>} T(A_>) \\
&= n + 1 + \frac{1}{(j-1)!} \sum_{A_<} T(A_<) + \frac{1}{(n-j)!} \sum_{A_>} T(A_>) \\
&= n + 1 + T_{\text{mittel}}(j-1) + T_{\text{mittel}}(n-j)
\end{aligned}$$

wobei $A_<$ alle Folgen durchläuft, die aus $a_2, \ldots, a_j$ durch Umordnung entstehen und $A_>$ alle Folgen, die aus $a_{j+1}, \ldots, a_n$ durch Umordnung entstehen.

Also ergibt sich insgesamt die Rekursionsgleichung

$$\begin{aligned} T_{\text{mittel}}(n) &= \frac{1}{n}\sum_{j=1}^{n}(n+1+T_{\text{mittel}}(j-1)+T_{\text{mittel}}(n-j)) \\ &= n+1+\frac{2}{n}\sum_{j=1}^{n}T_{\text{mittel}}(j) \end{aligned}$$

und $T_{\text{mittel}}(0) = T_{\text{mittel}}(1) = 0$.

Diese Rekursionsgleichung ist in D.1 gelöst. Es gilt

$$T_{\text{mittel}} \in O(n \log n)$$

3.5.6 Bester Fall

QuickSort benötigt am wenigsten Vergleiche, wenn bei der Zerlegung stets möglichst gleich lange Folgen entstehen und folgende Rekursionsungleichung gilt

$$T(n) \geq n + T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) \geq n + 2\,T(\lfloor \tfrac{n}{2} \rfloor)$$

Nach C.2.4 gilt $S \in \Theta(n \log n)$ für die Lösung von $S(n) = n + 2\,S(\lfloor \frac{n}{2} \rfloor)$. Mit 2.5.1 folgt

$$T \in \Omega(n \log n)$$

3.5.7 Bemerkung In praktischen Implementierungen sollte man die Rekursion spätestens bei $n = 10$ abbrechen, weil so kurze Folgen durch einfache Sortierverfahren schneller sortiert werden.

3.5.8 Speicherbedarf

Im Zerlegungsschritt braucht QuickSort außer dem zu sortierenden Array nur noch drei zusätzliche Speicherstellen (in Zeile (7) zum Vertauschen von $A[i]$ und $A[j]$ und für die Indizes i und j). Bei der Rekursion wird jedoch ein Stapel benötigt, um sich die lokalen Variablen l, j, r und die Rücksprungadressen usw. zu merken. Im schlechtesten Fall erreicht der Stapel die Tiefe $\Theta(n)$. Deshalb arbeitet QuickSort nicht am Ort.

Es gibt aber Modifikationen, die nur $O(\log n)$ oder sogar nur $O(1)$ zusätzlichen Speicher benötigen, also am Ort arbeiten (zu Lasten der Geschwindigkeit); siehe z.B. das Buch von Ottmann und Widmayer.

3.5.9 Schlußbemerkungen

In der Praxis ist Quick Sort oft etwas schneller als Merge Sort. Allerdings ist eine Laufzeit von $O(n \log n)$ nicht garantiert! Vor allem wenn die zu sortierenden Folgen nicht zufällig sind, sondern z.B. teilweise sortiert sind, kann Quick Sort langsam werden. Dem versucht man durch andere Auswahlen des Pivotelementes zu begegnen, z.B. indem man ein Element zufällig oder aus der Mitte der Folge wählt. Für jede Wahl gibt es aber einen schlechtesten Fall mit quadratischer Rechenzeit.

3.5.10 Quick Sort als C-Funktion

In der C-Bibliothek ist Quick Sort bereits als Funktion `qsort` implementiert; sie hat den Prototyp

```
void qsort( void *base, size_t nmemb, size_t size,
            int (*compar)(const void *, const void *));
```

`base` ist eine Zeiger auf den Anfang der Schlüsselfolge, `nmemb` ist ihre Länge und `size` die Speichergröße jedes Schlüssels. `compar` ist ein Zeiger auf eine Funktion, die Schlüssel vergleicht. Diese Funktion muß man selbst bereitstellen, denn sie wird je nach Typ der Schlüssel sehr unterschiedlich aussehen. Die Funktion `qsort` geht von keinem speziellen Schlüsseltyp aus; deshalb ist `base` auch als Zeiger auf `void` deklariert ebenso wie die Argumente der Vergleichsfunktion. Die Vergleichsfunktion muß einen negativen Wert zurückgeben, wenn der erste Schlüssel kleiner als der zweite ist, Null, wenn beide gleich sind, und einen positiven Wert sonst. Genaueres kann man in der Manualseite nachlesen.

Am leichtesten ist der Gebrauch von `qsort` vielleicht an folgendem Beispiel zu verstehen. Es sortiert eine Folge von `int`, allerdings mit einer selbst gemachten Vergleichsfunktion.

```
/***************************************
          qsort - Demo
***************************************/

#include <stdio.h>
#include <stdlib.h>

int IntegerVergleich( const void *x, const void *y) {
  return *(int *)x-*(int *)y;
}

int main(void) {
   int i;
   int (* compar)() = IntegerVergleich;
   int A[10] = { 3,4,100,1,2,2,8,9,7,0 };

   qsort(A,10,sizeof(A[0]),compar);

   for(i=0;i<10;i++) printf("%d ", A[i]);
   printf("\n");

   return 0;
}
```

3.6 Heap Sort

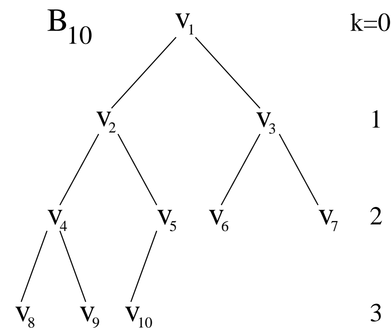
Heap Sort wurde 1964 von J.W.J. Williams und R.W. Floyd entwickelt. Bei Heap Sort wird wie bei Selection Sort immer ein Datensatz mit maximalem (oder mit minimalem) Schlüssel aus der noch nicht sortierten Restfolge ausgewählt. Um die quadratische Rechenzeit von Selection Sort zu reduzieren, wird die Restfolge stets zu einem Heap (Halde, Haufen) gemacht, wodurch das Finden eines maximalen Elements trivial wird.

3.6.1 Definition Ein **MaxHeap** über einer Schlüsselmenge S ist ein Binärbaum $B = (V, E)$ mit Knotenmarkierung $\sigma\colon V \to S$, so daß stets $\sigma(v) \geq \sigma(u)$, wenn u Kind von v ist. Gilt dagegen stets $\sigma(v) \leq \sigma(u)$, so spricht man von einem **MinHeap**.

Beachte: In einem MaxHeap ist der Schlüssel der Wurzel gleich dem Maximum der Schlüssel aller Knoten.

Wir werden im Folgenden meist nur einen MaxHeap betrachten und oft einfach schlechthin von Heap sprechen.

Für $n \in \mathbb{N}$ sei B_n der folgende Binärbaum:
Die Knotenmenge ist $V_n = \{v_1, \ldots, v_n\}$.
v_1 ist die Wurzel.
v_{2i} und v_{2i+1} sind Brüder; v_i ist ihr Vater. Die Knoten mit geradzahligen Indizes sind immer die linken Söhne.
Die Höhe von B_n ist $\lfloor \log_2 n \rfloor$. Die Knoten der Tiefe k (d.h. in der k-ten Etage von der Wurzel ($k = 0$) aus gezählt) sind genau die v_i mit $2^k \leq i < 2^{k+1}$. Die Etagen des Baumes sind also vollständig belegt, außer eventuell der $\lfloor \log_2 n \rfloor$-ten d.h. der untersten. In ihr befinden sich die Knoten möglichst weit links.



Die zu sortierende Folge von Schlüsseln befinde sich in dem Array $A[1 \ldots n]$. Wir definieren durch $\sigma(v_i) = A[i]$ eine Markierung der Knoten von B_n.

3.6.2 Skizzierung der Idee von Heap Sort

(1) Ordne $A[1 \ldots n]$ so um, daß B_n ein MaxHeap ist.

(2) **for** $(r = n; r \geq 2; r = r - 1\ )$ {

(3) vertausche $A[1]$ und $A[r]$;

(4) ordne $A[1 \ldots r-1]$ so um, daß B_{r-1} ein MaxHeap ist.

}

Am Ende ist das Array $A[1 \ldots n]$ sortiert.

Für die Effizienz dieses Algorithmus ist es wesentlich, daß man für die Umordnungen in (1) und (4) schnelle Prozeduren `BuildHeap` bzw. `RebuildHeap` findet. Es ist zu erwarten, daß die Umordnung in (4) leichter zu bewerkstelligen ist, weil ja höchstens

der Schlüssel der Wurzel die Heapbedingung verletzt.

3.6.3 Die Prozedur RebuildHeap

Idee: Lasse $\sigma(Wurzel)$ heruntersickern, bis die Heap-Eigenschaft erfüllt ist. Damit nach Vertauschen von $\sigma(v)$ mit dem Schlüssel eines Sohnes u die Heap-Eigenschaft hergestellt wird, wählt man für u stets den Sohn mit dem größeren Schlüssel.

Eingabe: Ein Binärbaum $B = (V, E)$ mit Knotenmarkierung $\sigma\colon V \to S$ derart, daß die Teilbäume, die als Wurzeln die Kinder der Wurzel von B haben, Max-Heaps sind.

Ausgabe: Eine durch Umordnung der Werte von σ enstandene Knotenmarkierung, so daß B ein MaxHeap ist.

```
RebuildHeap(B) {
    v = Wurzel von B;
    while ( v hat einen linken Sohn v_l ) {
        if ( v hat einen rechten Sohn v_r und σ(v_r) > σ(v_l) ) u = v_r;
        else u = v_l;
        if ( σ(v) < σ(u) ) {
            vertausche σ(v) und σ(u);
            v = u;
        }
        else return;
    }
}
```

3.6.4 Aufwand von RebuildHeap

Im Körper der **while**-Schleife wird eine konstante (von der Höhe von B unabhängige) Anzahl von Operationen ausgeführt. Die Anzahl der Schleifendurchläufe ist durch die Höhe von B beschränkt. Deshalb gibt es ein $c > 0$, so daß für jeden Binärbaum B gilt

$$\text{Anzahl der Operationen} \leq c \cdot \text{Höhe von } B$$

3.6.5 Die Prozedur BuildHeap

Für jeden Knoten $v \in V_n$ sei $B_n(v)$ der Teilbaum von B_n, dessen Wurzel v ist. Setze Höhe(v) = Höhe von $B_n(v)$.
Es gilt: Höhe$(v) = 0 \iff v$ ist ein Blatt
Sei $2^k \leq n < 2^{k+1}$, $k = \lfloor \log_2 n \rfloor$. Dann ist k = Höhe von B.
Für $0 \leq j \leq k$ gibt es $\left\lceil \frac{n}{2^{j+1}} \right\rceil$ Knoten der Höhe j in B_n. Dies sieht man durch Induktion nach j. Die Söhne eines Knotens mit Index i haben die Indizes $2i$ und $2i + 1$. Ein Knoten hat also genau dann die Höhe 0, wenn sein Index i so groß ist, daß $2i > n$ ist; dies trifft für alle Knoten mit den Indizes $\left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n$ zu, und davon gibt es $\left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{n}{2^{j+1}} \right\rceil$, wobei $j = 0$. Für die größeren Höhen j kann man ähnlich argumentieren.

Es gilt

$$\left\lceil \frac{n}{2^{j+1}} \right\rceil \leq \left\lceil \frac{2^{k+1}}{2^{j+1}} \right\rceil = \frac{2^k}{2^j} \leq \frac{n}{2^j}$$

Es gilt Höhe$(v_j) \geq$ Höhe(v_i) für $j \leq i$.

Die Idee von BuildHeap

Auf jedem Teilbaum $B_n(v)$ von B_n wird die von $A[1 \ldots n]$ induzierte Knotenmarkierung betrachtet.

Für jedes Blatt v (= Knoten der Höhe 0) ist $B_n(v)$ ein MaxHeap; denn $B_n(v)$ besteht nur aus v.

Sind v und u Brüder mit Vater w und sind $B_n(v)$ und $B_n(u)$ MaxHeaps, so mache $B_n(w)$ zu einem MaxHeap mit Hilfe der Prozedur RebuildHeap. Führe das solange durch, bis B_n ein MaxHeap ist.

BuildHeap($A[1 \ldots n]$) {
    for $(j = \left\lfloor \frac{n}{2} \right\rfloor; j \geq 1; j = j-1)$ RebuildHeap($B_n(v_j)$);
}

Beweis der Korrektheit: Der Algorithmus terminiert offensichtlich. Die Korrektheit zeigen wir durch vollständige Induktion nach j.
Für $\left\lfloor \frac{n}{2} \right\rfloor < j \leq n$ ist v_j ein Blatt und somit $B_n(v_j)$ ein MaxHeap. Sei $j \leq \left\lfloor \frac{n}{2} \right\rfloor$. Nach Induktionsvoraussetzung ist $B_n(v_i)$ für jedes $i > j$ ein MaxHeap. Die Söhne von v_j haben Indizes größer als j, sind also nach Induktionsvoraussetzung Wurzeln von MaxHeaps. In $B_n(v_j)$ ist daher die Heapbedingung höchstens bei der Wurzel v_j verletzt. Mit RebuildHeap($B_n(v_j)$) kann deshalb $B_n(v_j)$ zu einem MaxHeap gemacht werden. Zuletzt wird $B_n(v_1) = B_n$ zu einem MaxHeap gemacht. q.e.d.

3.6.6 Aufwandsanalyse für BuildHeap

Mit 3.6.4 folgt für $k = \lfloor \log_2 n \rfloor$, also $2^k \leq n < 2^{k+1}$,

$$\begin{aligned}
\text{Aufwand von BuildHeap(A[1 \ldots n])} &\leq c \cdot \sum_{j=1}^{\lfloor \frac{n}{2} \rfloor} \text{Höhe}(v_j) \\
&= c \cdot \sum_{v \in V_n} \text{Höhe}(v) \\
&= c \cdot \sum_{j=1}^{k} j \cdot \text{Anzahl der Knoten mit Höhe } j \\
&= c \cdot \sum_{j=1}^{k} j \left\lceil \frac{n}{2^{j+1}} \right\rceil \\
&\leq c \cdot \sum_{j=1}^{k} j \frac{n}{2^j} \quad \text{(nach 3.6.5)} \\
&\leq c\, n \sum_{j=1}^{n} j \left(\frac{1}{2}\right)^j
\end{aligned}$$

$$\begin{aligned}
&\leq c\,n \sum_{j=1}^{\infty} j \left(\frac{1}{2}\right)^j \\
&\leq c\,n \left[x \frac{d}{dx} \left(\sum_{j=0}^{\infty} x^j \right) \right]_{x=\frac{1}{2}} \\
&= c\,n \left[x \frac{d}{dx} \frac{1}{1-x} \right]_{x=\frac{1}{2}} \\
&= c\,n \left[x \frac{1}{(1-x)^2} \right]_{x=\frac{1}{2}} \\
&= 2\,c\,n
\end{aligned}$$

Der Aufwand von BuildHeap liegt also in $O(n)$.

3.6.7 Der Algorithmus Heap Sort

HeapSort($A[1 \ldots n]$) {
    BuildHeap($A[1 \ldots n]$);
    for $(r = n, r \geq 2; r = r - 1)$ {
        vertausche $A[1]$ und $A[r]$;
        RebuildHeap(B_{r-1});
    }
}

3.6.8 Folgerung Der Aufwand von HeapSort im schlechtesten Fall liegt in $O(n \log n)$.

Beweis: Nach 3.6.6 liegt der Aufwand von BuildHeap in $O(n)$. Die Prozedur RebuilHeap in der vierten Zeile hat nach 3.6.4 einen Aufwand in $O(\log_2 r)$. Für die $n-1$ Durchläufe durch die **for**-Schleife ergibt das einen Aufwand in $O(\log n + \log(n-1) + \ldots + \log 2) \subset O(n \log n)$. Insgesamt folgt $T_{\text{worst}} \in O(n \log n)$. q.e.d.

3.6.9 Bemerkungen

1. Indem man geeignete Eingabefolgen konstruiert, kann man sich überlegen, daß für Heap Sort $T_{\text{worst}} \in \Theta(n \log n)$ und auch $T_{\text{average}} \in \Theta(n \log n)$ gilt.

2. So wie wir Heap Sort formuliert haben, ist der Aufwand im besten Fall kleiner als $n \log n$. Es gibt Varianten (Smooth Sort), deren Aufwand bei vorsortierten Eingabefolgen linear ist (im Gegensatz zu Quick Sort!).

3. Speicherbedarfsanalyse: Heap Sort arbeitet **am Ort**, d.h. außer dem Platz für die Eingabefolge wird nur ein konstanter (von der Länge der Eingabefolge unabhängiger) zusätzlicher Platz benötigt.

4. Heap Sort ist nicht stabil.

3.6.10 Realisierung von Prioritätswarteschlangen mit Heaps

Eine Prioritätswarteschlange ist ein Datentyp, der sich von dem der üblichen Warteschlange dadurch unterscheidet, daß jedem Element beim Einfügen in die Warteschlange (mit enqueue) ein Prioritätswert gegeben wird und daß bei bei den Operationen front und dequeue nicht das Element, welches am längsten in der Warteschlange ist, sondern das mit der höchsten Priorität ausgelesen bzw. herausgenommen wird.

Die beiden Operationen front und dequeue lassen sich schnell ausführen, wenn die Prioritätswarteschlange als MaxHeap realisiert wird; denn dann steht ja das Element mit größter Priorität in der Wurzel des Heaps. Somit erfordert front überhaupt keine Vergleichsoperationen. Bei dequeue kann man wie bei HeapSort vorgehen: das Element mit dem größten Index wird an die Stelle der Wurzel gesetzt und anschließend mit RebuildHeap wieder die Heapbedingung hergestellt. Der Aufwand dafür liegt in $O(\log(\text{Länge der Schlange}))$.

Auch die Einfügeoperation enqueue kann entsprechend schnell ausgeführt werden. Das einzufügende Element wird als ein neues Blatt rechts unten an den Heap angefügt, und dann läßt man es nach oben in Richtung der Wurzel wandern, bis seine Priorität kleiner als die des Elternknotens ist und somit wieder ein Heap vorliegt. Dies ist invers zu dem Heruntersickern bei RebuildHeap. Die Anzahl der Rechenschritte wächst ebenfalls höchstens linear mit der Höhe des Heaps, liegt also auch in $O(\log(\text{Länge der Schlange}))$.

3.7 Zusammenfassender Überblick

Genauere Untersuchungen obiger Sortierverfahren liefern folgende Ergebnisse.

Ungefähre Anzahl		Merge Sort	Quick Sort	Heap Sort	Selection Sort	Insertion Sort	Bubble Sort
Vergleiche	schlechtester	$n \log n$	$\frac{1}{2}n^2$	$2n \log n$	$\frac{1}{2}n^2$	$\frac{1}{2}n^2$	$\frac{1}{2}n^2$
Austauschoperationen	Fall				n	$\frac{1}{4}n^2$	$\frac{1}{2}n^2$
Vergleiche	im Mittel	$n \log n$	$1,44 \cdot n \log n$	$2n \log n$	$\frac{1}{2}n^2$	$\frac{1}{4}n^2$	$\frac{1}{2}n^2$
Speicherbedarf		$2n$ $+const.$	$n + \log n$ $+const.$	n $+const.$	n $+const.$	n $+const.$	n $+const.$
				am Ort	am Ort	am Ort	am Ort

Die Anzahl der anderen Operationen liegt stets in $\Theta(\text{Anzahl der Vergleiche})$.

Die tatsächliche Laufzeit auf Rechnern hängt von der jeweiligen Implementierung und der Rechnerplattform ab. Geschickte Implementierungen von Quick Sort sind oft im Mittel etwas schneller als Merge Sort oder Heap Sort. Allerdings darf man nicht vergessen, daß Quick Sort für manche Eingaben langsam ist.

3.8 Sortieren durch Verteilen in Fächer, Bin Sort

3.8.1 Counting Sort

3.8.1.1 Voraussetzung: Die Menge S der zulässigen Schlüssel ist ein endliches Intervall $S = \{a, \ldots, b\}$ in $\mathbb{Z}$.

Idee: Lege ein Array $H[a \ldots b]$ von Integers an und initialisiere alle Elemente mit 0. Dann durchlaufe die eingegebene Schlüsselfolge $a_1, \ldots, a_n$ und erhöhe an der Stelle i den Wert von $H[a_i]$ um eins. Dann durchlaufe das Array H von links nach rechts und gib an der Stelle j genau $H[j]$-mal die Zahl j aus. Die so gebildete Ausgabefolge ist eine aufsteigend sortierte Version von $a_1, \ldots, a_n$.
H nennt man das **Histogramm** der Folge $a_1, \ldots, a_n$. Dividiert man jede Komponente von H durch n, so erhält man die Häufigkeitverteilung der Folge $a_1, \ldots, a_n$.

```
CountingSort( A[1...n] ) {
    H[a...b] sei ein Array von int, dessen Elemente alle den wert 0 haben;
    for (i = 1; i ≤ n; i = i + 1)
        H[A[i]] = H[A[i]] + 1;
    i = 1;
    for (j = a; j ≤ b; j = j + 1) {
        for (k = 0; k < H[j]; k = k + 1) {
            A[i] = j;
            i = i + 1;
        }
    }
}
```

3.8.1.2 Aufwand: Bei dem Durchlauf durch das Array A werden $\approx c_1 \cdot n$ viele Operationen ausgeführt und bei dem anschließenden Durchlauf durch $H \approx c_2 \cdot \#S$ viele Operationen. Der Gesamtaufwand liegt also in $O(n)$. Um die Abhängigkeit von der Größe des zulässigen Schlüsselbereichs auszudrücken, wird oft $O(n+m)$ mit $m = \#S$ geschrieben, was aber streng genommen nach unserer Definition von O nichts anderes als $O(n)$ ist. Die Notation O haben wir nur für eine Veränderliche eingeführt.

3.8.1.3 Nachteile:

1. Obwohl der Aufwand linear ist, kann man dieses Verfahren nur manchmal sinnvoll einsetzen. Denn es benötigt soviel Integer-Speicherplätze, wie es zulässige Schlüsselwerte gibt. Für große Schlüsselmengen S ist es also praktisch nicht einsetzbar. Ist $n \ll \#S$, so wird der größte Teil des Arrays H überhaupt nicht benutzt, muß aber trotzdem durchlaufen werden. Sinnvoll ist es, Counting Sort zu benutzen, S nicht allzu groß ist und wenn zu erwarten ist, daß sich die Werte der a_i einigermaßen gleichmäßig über S verteilen.

2. Weil in dem Array H nur die Häufigkeiten der a_j abgelegt werden, kann man nur die Schlüssel sortieren, aber nicht die damit eventuell verbundenen Datensätze. Diese Einschränkung läßt sich aber durch eine Modifikation beheben, wie das Folgende zeigt.

3.8.2 Bucket Sort

3.8.2.1 Voraussetzung: Die Menge S der zulässigen Schlüssel ist endlich (und relativ klein). $\varphi\colon S \to \{1,\dots,m\} \subset \mathbb{N}$ sei eine bijektive, ordnungserhaltende Abbildung (d.h. $s_1 < s_2 \Longleftrightarrow \varphi(s_1) < \varphi(s_2)$).

3.8.2.2 Erste Version von Bucket Sort

```
BucketSort( A[1...n] ) {
        L[1...m] sei eine Array von leeren Listen;
        for (i = 1; i ≤ n; i = i + 1)
                hänge A[i] an die Liste L[φ(A[i])] an;
        durchlaufe die Listen L[1],...,L[m] in dieser Reihenfolge
        und lege die angetroffenen Elemente der Reihe nach in A[1...n] ab;
}
```

Jede Liste $L[1],\dots,L[n]$ wird ein Fach (bucket) genannt. In praktischen Implementierungen enthält $A[i]$ meist nicht nur Schlüssel, sondern auch Zeiger auf entsprechende Datensätze. Die Listen $L[i]$ können auf sehr unterschiedliche Weisen realisiert werden, z.B. als mit Zeigern verkettete Listen, aber auch als Teilarrays eines großen Arrays $B[1\dots n]$. Bucket Sort bekommt dann folgende Gestalt.

3.8.2.3 Zweite Version von Bucket Sort

```
BucketSort2( A[1...n] ) {
        B[1...n] sei ein Array von demselben Typ wie A[1...n];
        Z[1...m] sei ein Array vom Typ integer;
        initialisiere Z mit den Werten 0;
        for (i = 1; i ≤ n; i = i + 1) Z[φ(A[i])] = Z[φ(A[i])] + 1;
        for (j = 2; j ≤ m; j = j + 1) Z[j] = Z[j] + Z[j − 1];
        for (i = n; i ≥ 1; i = i − 1) {
                k = φ(A[i]));
                B[Z[k]] = A[i];
                Z[k] = Z[k] − 1;
        }
        return B;
}
```

Zunächst wird bestimmt, wie oft jeder Schlüssel vorkommt. Dann werden die $A[i]$ mit $\varphi(A[i]) = k$ in $B[Z[k-1]+1\dots Z[k]]$ abgelegt.

3.8.2.4 Aufwandanalyse

1. Version: Der Durchlauf durch die Eingabefolge braucht $\approx c_1 \cdot n$ Operationen.
Der Durchlauf durch $L[1],\dots,L[n]$ benötigt $\approx c_2 \cdot (m+n)$ Operationen.
Also insgesamt liegt die Anzahl der Operationen in $\Theta(n)$.

2. Version: 2 Durchläufe durch die Folge $\approx c_1 \cdot 2n$.
1 Durchlauf durch $Z \approx c_2 \cdot n$
Also insgesamt liegt die Anzahl der Operationen in $\Theta(n)$.

(Weil die Anzahl der Operationen linear mit $n+m$ wächst, wird oft etwas mißbräuchlich $\Theta(n+m)$ geschrieben.)

3.8.2.5 Speicherbedarf

1. Version: $2n$ Datensätze, m Listenköpfe
2. Version: $2n$ Datensätze, m Integerplätze.

3.8.2.6 Fazit: Bucket Sort bringt für kleine m Vorteile; für sehr große m ($\gg n$) ist es ineffizient. Die obigen Versionen sind stabil, arbeiten aber nicht am Ort.

3.8.2.7 Gröbere Facheinteilungen

Manchmal bietet es sich an, die Verteilung der Schlüssel $a_1, \ldots, a_n$ auf die Fächer nicht bezüglich der Gleichheit, sondern erst einmal bezüglich einer gröberen Äquivalenzrelation vorzunehmen. So kann man z.B. Wörter zunächst einmal nach dem Anfangsbuchstaben (oder allgemein dem j-ten Buchstaben) einteilen. Gibt es nur endlich viele Fächer (d.h. Äquivalenzklasse) $F_1, \ldots, F_r$ und kann man sie so numerieren, daß alle Schlüssel in F_α kleiner als die in F_β sind, wenn $\alpha < \beta$ ist, so braucht man nur noch jeweils die Schlüsselmenge in jedem Fach zu sortieren und diese sortierten Folgen hintereinander zu hängen. Wenn die Schlüssel einigermaßen gleichmäßig auf die Fächer verteilt sind, sind in jedem Fach also nur etwa $\frac{n}{r}$ Schlüssel. Ist $\frac{n}{r}$ relativ klein, kann man die Schlüssel in jedem Fach mit einem einfachen vergleichsbasierten Verfahren sortieren.

Eine andere Möglichkeit ist, die Schlüssel jedes Faches noch einmal auf die beschriebene Weise in neue Fächer zu verteilen, allerdings gemäß einer anderen Äquivalenzrelation; z.B. bei Wörtern gemäß des zweiten Buchstabens. Dies kann man eventuell eventuell wiederholen, bis schließlich in jedem Fach nur noch Schlüssel sind, die tatsächlich gleich sind. Dafür müssen die Äquivalenzrelationen geeignet gewählt sein. Besteht die Schlüsselmenge aus n-Tupeln mit der lexikografischen Ordnung, so kann man die Verteilung sukzessive nach den Komponenten vornehmen, wobei man mit der letzten beginnt. Für das so entstehende Sortierverfahren hat sich der Name Radix Sort eingebürgert.

3.8.3 Radix Exchange Sort

Voraussetzung: Die Schlüsselmange S hat die Gestalt $S = M^k$, wobei M eine endliche geordnete Menge und $k \in \mathbb{N}$ sind. Auf S wird die induzierte lexikografische Ordnung verwendet.

```
RadixSort( A[1...n] ) {
    for (j = k; j ≥ 1; j = j − 1)
        sortiere A[1...n] mit BucketSort bezüglich der j-ten Komponente;
}
```

Das funktioniert, weil BucketSort stabil ist.

Die Anzahl der ausgeführten Operationen wächst linear mit $n \cdot k$. Dafür wird oft (etwas mißbräuchlich) $\Theta(n \cdot k)$ oder $\Theta(n \log \#S)$ geschrieben.

3.9 Eine untere Laufzeitschranke für vergleichsbasierte Sortierverfahren

Bucket Sort und Radix Sort verwenden keine expliziten Vergleiche von Schlüsseln bezüglich der gegebenen Ordnung. Dadurch daß jedoch der Schlüssel zur Indizierung der Fächer benutzt wird, wird jeder Schlüssel in der Eingabefolge sogar einer Entscheidung zwischen m Alternativen unterworfen. Dadurch gelingt es, lineare Laufzeit zu erreichen.

Die anderen zitierten Sortierverfahren verwenden dagegen nur Entscheidungen zwischen zwei Alternativen, nämlich $a_i > a_j$ oder $a_i < a_j$ (oder ähnliches). Solche Verfahren nennt man vergleichsbasiert. Genauer meint man damit Algorithmen, die nach dem Start stets durch eine Binärentscheidung (einen Schlüsselvergleich) in den nächsten Zustand kommen, bis sie einen Endzustand erreichen und terminieren. Sie sind durch einen vollständigen Binärbaum beschreibbar, bei dem jeder Knoten eine Binärentscheidung ist, die je nach Ausgang zu einem der Söhne führt. Da das Ergebnis des Sortierens einer Folge $a_1, \ldots, a_n$ eine Permutation von $\{1, \ldots, n\}$ ist, muß jedem vergleichsbasierten Sortieralgorithmus, der Folgen der Länge n sortiert, ein Entscheidungsbaum mit mindestens $n!$ Blättern entsprechen. Ein Binärbaum mit $n!$ Blättern muß mindestens die Höhe $\lfloor \log_2(n!) \rfloor$ haben. Daher gibt es mindestens eine Eingabefolge $a_1, \ldots, a_n$, für die der Sortieralgorithmus $\lfloor \log_2(n!) \rfloor$ Vergleiche ausführt. Somit gilt

$$\text{Anzahl der Vergleiche im schlechtesten Fall} \geq \lfloor \log_2(n!) \rfloor$$

Nach A.2.10 c) gilt $\log(n!) \in \Theta(n \log n)$. Damit ergibt sich:

Jedes vergleichsbasierte Sortierverfahren benötigt im schlechtesten Fall $\Omega(n \log n)$ Vergleiche.

Kapitel 4

Berechenbarkeit

Die Theorie der Berechenbarkeit untersucht, welche Probleme algorithmisch gelöst werden können und welche Funktionen algorithmisch berechnet werden können. Dazu muß erst einmal der Begriff des Algorithmus präzisiert werden. Intuitiv ist ein Algorithmus eine Vorschrift, die die Berechnung einer Funktion oder die Entscheidung der Wahrheit einer Aussage völlig schematisch durchzuführen erlaubt, ohne beim Rechnenden irgendwelche Intelligenz oder Kreativität vorauszusetzen. Das bedeutet, daß so eine Rechenvorschrift im Prinzip von einer Maschine durchgeführt werden können muß (in der Realität werden durch Speicher und Geschwindigkeit Grenzen gesetzt).

Seit etwa 1900 wurde versucht, diese Vorstellung mathematisch zu präzisieren, so daß exakte Beweise geführt werden können. Dabei wurden völlig unterschiedliche Ansätze gewählt:

- durch Rechnermodelle wie Turingmaschinen (1936, Alan M. Turing (1912-1954)) und Registermaschinen (1963, I.C. Sheperdson, H.E. Sturgis).
- durch rekursive Funktionen (J. Herbrand (1908-1931), K. Gödel, S.C. Kleene).
- durch Termersetzungskalküle (A. Church 1936, E.L. Post 1943, A.A. Markov 1951, Gödel, Herbrand, Kleene).

Überraschenderweise haben sie sich alle als äquivalent erwiesen, so daß es heute einen gut etablierten Begriff der Berechenbarkeit gibt.

Wir werden hier nur einen Überblick geben und einige der Grundideen skizzieren. Unserer Darstellung fehlt aufgrund ihrer Kürze leider völlig die mathematische Präzision, welche die Theorie der Berechenbarkeit von einer philosophischen Spekulation abhebt und zu einer harten Theorie macht.

4.1 Einige Entscheidungsproblem

Historisch gesehen waren für die Entwicklung der Theorie der Berechenbarkeit unter anderem folgende Probleme wichtig (die wir allerdings nicht in chronologischer Reihenfolge zitieren).

4.1.1 Das Halteproblem

Gibt es einen Algorithmus, der für jedes Rechnerprogramm entscheiden kann, ob es nach endlich vielen Schritten terminiert oder nicht?

4.1.2 Das Korrektheitsproblem

Gibt es einen Algorithmus, der für jedes Rechnerprogramm entscheiden kann, ob es eine vorgegebene Funktion berechnet oder nicht?

4.1.3 Das Äquivalenzproblem

Gibt es einen Algorithmus, der für je zwei Rechnerprogramme entscheiden kann, ob sie dasselbe berechnen oder nicht?

4.1.4 Das zehnte Hilbertsche Problem

(1900 von D.Hilbert auf dem Internationalen Mathematiker-Kongress vorgetragen)

Gibt es einen Algorithmus, mit dem man für jede diophantische Gleichung entscheiden kann, ob sie lösbar ist oder nicht?

Eine diophantische Gleichung ist eine Gleichung der Form $P(x_1,\dots,x_n)=0$, wobei $P\in\mathbb{Z}[X_1,\dots,X_n]$ ein Polynom mit ganzzahligen Koeffizienten ist. Sie heißt lösbar, wenn es $x_1,\dots,x_n\in\mathbb{Z}$ mit $P(x_1,\dots,x_n)=0$ gibt?

4.1.5 Das Wortproblem für Gruppen

Gibt es einen Algorithmus, der für jede Gruppe, die durch endlich viele Erzeugende mit endlich vielen Relationen zwischen ihnen definiert ist, und für beliebige aus den Erzeugenden gebildeten Wörtern w_1 und w_2 entscheidet, ob w_1 und w_2 dasselbe Gruppenelement darstellen?

4.1.6 Das Tautologieproblem der Prädikatenlogik 1. Stufe

Gibt es einen Algorithmus, der für jede prädikatenlogische Formel (erster Stufe) entscheidet, ob sie allgemeingültig ist oder nicht? (Erster Stufe bedeutet grob gesagt, daß nur Quantoren über Elementvariablen vorkommen und keine über Prädikate oder Teilmengen.)

Das Tautologieproblem spielt eine Rolle bei der Frage, ob man im Prinzip automatische Theorembeweiser bauen kann. Sind nämlich Axiome $A_1,\dots,A_n$ und eine Aussage B gegeben, so gilt:

$$\begin{aligned}
B \text{ ist eine Folgerung von } A_1,\dots A_n &\iff A_1\wedge\dots\wedge A_n \longrightarrow B \text{ ist allgemeingültig}\\
&\iff \neg(A_1\wedge\dots\wedge A_n)\vee B \text{ ist allgemeingültig}\\
&\iff A_1\wedge\dots\wedge A_n\wedge\neg B \text{ ist unerfüllbar}
\end{aligned}$$

4.2 Gödelisierung

Diese Entscheidungsprobleme kann man auch auffassen als Fragen, ob eine gewisse Funktion f algorithmisch berechenbar ist, nämlich diejenige Funktion, die auf der Menge der jeweils betrachteten Objekte definiert ist und den Wert 1 hat, wenn die Antwort positiv ist, und 0 sonst. Es genügt, Funktionen mit Definitionsbereich $\mathbb{N}$ zu betrachten, weil die mathematischen Objekte, die als Funktionsargumente vorkommen, durch Wörter über einem endlichen Alphabet beschrieben werden, die man

ihrerseits durch natürliche Zahlen codieren kann. Solche Codierungen heißen Gödelisierungen nach K.Gödel (1906-1978), der sie 1936 erstmals verwendete.

4.3 Algorithmisch unentscheidbare Probleme

Mittlerweile ist bekannt, daß die oben aufgeführten Fragen negative Antworten haben d.h. daß es keine Algorithmen gibt, die die jeweils angegebenen Probleme entscheiden können.

Das zehnte Hilbertsche Problem wurde 1970 von Matijasevic gelöst; das Wortproblem für Gruppen 1955 von Boone und Novikov; das entsprechende Wortproblem für Semi-Thue-Systeme 1947 von Post und Markov; das Entscheidungsproblem der Prädikatenlogik 1936 von Church und Turing. Die Lösungen des Halteproblems und des Korrektheitsproblems folgen recht einfach aus einem Satz von Rice (1953), waren aber schon Turing bekannt.

4.4 Turingmaschinen

Turing wollte das Rechnen mit Bleistift, Papier und Radiergummi formalisieren. Das Papier wird dabei durch ein nach beiden Seiten hin unendliches Band idealisiert, das in einzelne Felder unterteilt ist. Der Rechnende mit dem Bleistift und dem Radiergummi wird durch einen Schreib-Lese-Kopf ersetzt, der sich zu jedem Zeitpunkt auf einem der Felder, dem sogenannten Arbeitsfeld, befindet. Der Schreib-Lese-Kopf kann folgende Befehle ausführen:

`CLR`	löscht das Zeichen auf dem Arbeitsfeld
`PRT`	druckt das Zeichen * auf das Arbeitsfeld
`BEQ`	testet, ob das Arbeitsfeld leer ist
`LFT`	bewegt den Kopf um ein Feld nach links
`RHT`	bewegt den Kopf um ein Feld nach rechts

4.4.1 Steuerprogramme für Turingmaschinen

Ein Algorithmus besteht nun aus einer Abfolge solcher Befehle, durch die festgelegt wird, was der Schreib-Lese-Kopf tut. Um Schleifen und Verzweigungen modellieren zu können, werden die einzelnen Rechenschritte mit Kennmarken numeriert, und nach jedem Befehl wird eine Sprungmarke angegeben, welche als Kennmarke des nächsten Rechenschrittes dient. Jeder Rechenschritt wird somit als eine Anweisung der Gestalt

$$n_1 \texttt{ Befehl } n_2$$

geschrieben, wobei `Befehl` einer der obigen Befehle außer `BEQ` ist und $n_1 \in \mathbb{N}$ die Kennmarke und $n_2 \in \mathbb{N}$ die Sprungmarke sind, oder der Gestalt

$$n_1 \texttt{ BEQ } n_3\ n_4$$

wobei $n_1 \in \mathbb{N}$ die Kennmarke, $n_3 \in \mathbb{N}$ die Sprungmarke und $n_4 \in \mathbb{N}$ die Verzweigungsmarke sind. Ist das Arbeitsfeld leer, so wird als nächstes eine Anweisung mit Kennmarke n_3 ausgeführt, ansonsten eine mit Kennmarke n_4.

Ein Steuerprogramm (A, n_0) für eine Turingmaschine besteht aus einer Menge A von Steueranweisungen und der Kennmarke n_0 genau einer dieser Anweisungen; sie heißt Anfangsmarke. Durch ein Steuerprogramm (A, n_0) wird folgender Rechenvorgang beschrieben:

Zunächst wird der Befehl in derjenigen Anweisung a_0, deren Kennmarke die Anfangsmarke n_0 ist, ausgeführt. Dann geht man zu einer Anweisung $a \in A$, deren Kennmarke mit der Sprungmarke von a_0 (oder im Falle, daß a_0 eine Testanweisung ist und das Arbeitsfeld nicht leer ist, mit der Verzweigungsmarke von a_0) übereinstimmt. Gibt es keine solche Anweisung a, so wird der Rechenvorgang beendet. Deshalb nennt man alle Sprung- oder Verzweigungsmarken, die nicht Kennmarke einer Anweisung aus A sind, die Endmarken von A. Dieser Vorgang wird wiederholt, bis man auf eine Endmarke stößt.

4.4.2 Bemerkungen

1. Ein Steuerprogramm für eine Turingmaschine muß nicht immer nach endlich vielen Schritten enden!

2. So wie wir Steuerprogramme definiert haben, können sie nichtdeterministisch sein d.h. mehrere Anweisungen können dieselbe Kennmarke haben, so daß der Rechenablauf nicht eindeutig festgelegt ist. Das ist in der Komplexitätatheorie sinnvoll, wo man die Komplexität von Algorithmen analysiert. In der Theorie der Berechenbarkeit betrachtet man meist nur **deterministische Steuerprogramme**, bei denen jede Nummer höchstens einmal als Kennmarke vorkommt und somit der Rechenablauf eindeutig festgelegt ist.

4.4.3 Turingmaschinen-berechenbare Funktionen

Mit B bezeichnen wir ein leeres Feld und mit $*$ ein beschriebenes. Den Inhalt des Bandes kann man als unendlich langes Wort über dem Alphabet $\{B, *\}$ auffassen; sind nur endlich viele Felder beschrieben, so kann man ihn in der Form

$$B^\infty \, u \, \underline{z} \, v \, B^\infty$$

schreiben, wobei $u, v \in \{B, *\}^*$ und $z \in \{B, *\}$ sind. Das Feld, auf dem der Schreib-Lese-Kopf gerade steht, wird unterstrichen.

Um mit der Turingmaschine Abbildungen von $\mathbb{N}^k \to \mathbb{N}$ berechnen zu können, muß man die Argumente und Werte der Abbildung geeignet durch Bandinschriften codieren.

Die k-stellige Eingabefunktion ein_k ordnet jedem Tupel $(n_1, \ldots, n_k) \in \mathbb{N}^k$ die Bandinschrift

$$B^\infty \, \underline{B} \, *^{n_1} \; B \; *^{n_2} \; B \ldots B \; *^{n_k} \; B^\infty$$

zu. Und die Ausgangsfunktion aus ordnet jeder Bandinschrift der Gestalt

$$B^\infty \, u \, \underline{z} \, *^n \; B^\infty$$

den Wert $n \in \mathbb{N}^l$ zu.

Weil das Programm einer Turingmaschine nicht für jede anfängliche Bandinschrift terminieren muß, betrachtet man partielle Funktionen $f\colon \mathbb{N}^k \to \mathbb{N}$, das sind Funktionen, deren Defininitionsbereich $D(f)$ nicht unbedingt ganz $\mathbb{N}^k$ ist, sondern eine echte

Teilmenge davon sein kann. Eine partielle Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ heißt durch eine Turingmaschine berechenbar, wenn es ein Programm P für die Turingmaschine gibt, so daß $x \in D(f)$ damit äquivalent ist, daß P bei Eingabe von $ein_k(x)$ nach endlich vielen Rechenschritten stoppt mit einer Bandinschrift, die von aus auf $f(x)$ abgebildet wird.

4.5 Andere Maschinenmodelle

Programme für Turingmaschinen sind mühsam zu schreiben, weil selbst einfachste Rechnungen schon zu langen, unübersichtlichen Programmen führen. Deshalb hat man andere theoretische Maschinenmodelle entworfen, die mehr den heutigen Mikroprozessoren entsprechen.

4.5.1 Registermaschine

Statt der Felder einer Turingmaschine verwendet man Speicher für natürliche Zahlen. Dementsprechend wird der Befehlssatz sinnvoll abgeändert, indem man Inkrement- und Dekrementbefehle hinzunimmt. Man unterscheidet m-Registermaschinen, bei denen nur m Register zur Verfügung stehen und unbegrenzte Registermaschinen (URM), bei denen beliebig viele Register verwendet werden dürfen.

4.5.2 RAM

Eine Random Acces Machine (RAM) ensteht aus einer Registermaschine, indem man noch Kopierbefehle hinzunimmt.

4.5.3 RASP

Eine Random Acces Stored Program MAchine (RASP) ist eine RAM, bei der das Steuerprogramm in geeigneter Weise codiert in den Registern der Maschine abgespeichert wird. Infolgedessen kann so eine Maschine ihr eigenes Programm modifizieren.

4.5.4 Welche Funktionen können berechnet werden?

Es ist nicht verwunderlich, daß auf obigen Maschinen die Turingmaschine simuliert werden kann und somit jede Funktion, die mit der Turingmaschine berechnet werden kann, auch mit einer dieser Maschinen berechnet werden kann. Erstaunlicherweise kann man aber auch die Umkehrung beweisen: Jede Funktion, die von einer dieser Maschinen berechnet werden kann, kann auch mit einer Turingmaschine berechnet werden.

4.6 Rekursive Funktionen

Ganz unabhängig von jedem Maschinenbegriff kann man Klassen von intuitiv berechenbaren Funktionen angeben. Dabei geht man einigen einfachen Grundfunktionen aus und bildet aus ihnen durch gewisse Operationen kompliziertere Funktionen, aber so, daß sie in intuitivem Sinn berechenbar bleiben.

4.6.1 Definition der Grundfunktionen

a) Die konstanten Funktionen $\mathbb{N}^n \to \mathbb{N}$

b) Die Projektionen $\pi_i^n\colon \mathbb{N}^n \to \mathbb{N},\ (x_1, \dots x_n) \mapsto x_i$

c) Die Nachfolgerfunktion $S\colon \mathbb{N} \to \mathbb{N},\ z \mapsto z+1$

$\mathcal{F}^n$ sei die Menge der partiell (d.h. eventuell nur auf einer Teilmenge) definierten Funktionen von $\mathbb{N}^n$ nach $\mathbb{N}$.
Setze $\mathcal{F} = \bigcup_{n\in\mathbb{N}} \mathcal{F}^n$.

4.6.2 Definition der Grundoperatoren

a) Die **Substitution**

$$\mathcal{E}\colon \mathcal{F}^n \times (\mathcal{F}^n)^m \to \mathcal{F}^m$$

ist definiert durch

$$\mathcal{E}(f, g_1, \ldots, g_n)(x_1, \ldots, x_m) = f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$

b) Die **primitive Rekursion**

$$\mathcal{R}\colon \mathcal{F}^n \times \mathcal{F}^{n+2} \to \mathcal{F}^{n+1}$$

ist definiert durch

$$\mathcal{R}(g,h)(x_1, \ldots, x_n, x+1) = h(x_1, \ldots, x_n, x, \mathcal{R}(g,h)(x_1, \ldots, x_n, x))$$

c) Der **unbeschränkte μ-Operator**

Sei $f \in \mathcal{F}^{n+1}$. Für jedes $x = (x_1, \ldots, x_n) \in \mathbb{N}^n$ sei Z_x die Menge aller $z \in \mathbb{N}$ derart, daß $f(z,x) = 1$ gilt und (y,x) im Definitionsbereich von f liegt für jedes $y \leq z$.

Der unbeschränkte μ-Operator

$$\mu\colon \mathcal{F}^{n+1} \to \mathcal{F}^n$$

ist definiert durch

$$\text{Definitionsbereich von } \mu f = \{x \in \mathbb{N}^n \colon Z_x \neq \emptyset\}$$

$$\mu f(x) = \min Z_x \quad \text{für } x \text{ im Definitionsbereich}$$

4.6.3 Primitiv rekursive Funktionen

Eine Funktion heißt **primitiv rekursiv**, wenn sie durch endlichfaches Anwenden von $\mathcal{E}$ und $\mathcal{R}$ aus Grundfunktionen entsteht oder selbst eine Grundfunktion ist.

Jede primitiv rekursive Funktion ist somit intuitiv berechenbar, und sie ist total d.h. ihr Definitionsbereich ist ganz $\mathbb{N}^n$, wenn sie n Veränderliche hat.

Die Klasse der primitiv rekursiven Funktionen enthält z.B. die Addition $\mathbb{N}^2 \to \mathbb{N}$, die Multiplikation, die Exponentialfunktion, die Fakultät und die die Signumfunktion; man kann aber intuitiv berechenbare Funktionen konstruieren, die nicht primitiv rekursiv sind. Eine berühmte solche Funktion ist die Ackermann-Funktion (nach F.W. Ackermann, 1896-1962):

$$A\colon \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$

$$A(n,m) = \begin{cases} m+1 & \text{falls } n = 0 \\ A(n-1,1) & \text{falls } m = 0 \\ A(n-1, A(n,m-1)) & \text{sonst} \end{cases}$$

Die Ackermann-Funktion wächst stärker als es primitiv rekursiven Funktionen möglich ist. Man kann leicht nachrechnen, daß $A(1,1) = 3$, $A(2,2) = 7$, $A(3,3) = 61$ gilt; $A(4,4)$ ist jedoch bereits größer als $10^{10^{19\,000}}$.

4.6.4 Partiell rekursive Funktionen

Die Funktionen, die man aus den Grundfunktionen durch Anwenden aller drei Arten von Grundoperatoren konstruieren kann, nennt man die **partiell rekursiven Funktionen**. Sie sind nicht unbedingt total d.h. auf ganz $\mathbb{N}^n$ definiert, weil bei Anwendung des μ-Operators der Definitionsbereich schrumpfen kann. Diejenigen partiell rekursiven Funktionen, die total sind, heißen **rekursive Funktionen**. Mit dem μ-Operator stößt man an die Grenze dessen, was man vielleicht noch als intuitiv berechenbar bezeichnen möchte. Man mache sich aber klar, daß man für alle x im Definitionsbereich den Wert $\mu f(x)$ in endlich vielen Rechenschritten tatsächlich berechnen kann, wenn man f berechnen kann.

Man kann beweisen, daß die Klasse der partiell rekursiven Funktionen mit der Klasse der durch eine Turingmaschine berechenbaren Funktionen übereinstimmt. Die beiden völlig unterschiedlichen Ansätze, berechenbare Funktionen zu definieren, führt also auf denselben Begriff von Berechenbarkeit!

4.7 Entscheidbarkeit

Mit Hilfe einer Gödelisierung führt man die Frage, ob ein Problem algorithmisch entschieden werden kann, auf die Frage zurück, ob die charakteristischen Funktion einer bestimmten Teilmenge von $\mathbb{N}$ berechenbar ist, also rekursiv ist. Und diese Frage läßt sich häufig mit einem bekannten Satz von Rice beantworten, der z.B. im Falle des Korrektheitsproblems gerade besagt, daß die Menge der Gödelnummern derjenigen Programme, die eine vorgegebene Funktion berechnen, keine rekursive charakteristische Funktion hat.

4.8 Bemerkungen

Um 1900 bestand bei vielen Mathematikern die Vorstellung, daß man die Mathematik vielleicht algorithmisieren könne. Dies ist aber wegen der Unentscheidbarkeit des Tautologieproblems der Prädikatenlogik erster Stufe nicht möglich. Allerdings gibt es einen Algorithmus (Herleitungskalkül), der genau die allgemeingültigen Formeln der Prädikatenlogik erster Stufe erzeugt. Dies folgt aus Gödels berühmtem Vollständigkeitssatz.

Die Prädikatenlogik erster Stufe ist allerdings nicht sehr ausdrucksstark. So ist es z.B. möglich, Modelle für die Peano-Axiome anzugeben, die nicht isomorph zu $\mathbb{N}$ sind, in

denen aber dieselben Aussagen erster Stufe gelten; ebenso gibt es Nichtstandardmodelle von $\mathbb{R}$. Üblicherweise verwendet man in der Mathematik aber Aussagen höherer Stufe d.h. man benutzt nicht nur Quantoren, die über Elementvariablen laufen, sondern auch solche, die über Teilmengenvariablen laufen. Man kann beweisen, daß es keinen Algorithmus gibt, der genau die allgemeingültigen Formeln dieser Prädikatenlogik zweiter Stufe generiert.

Kapitel 5

NP-Vollständigkeit

5.1 Effiziente Algorithmen

Üblicherweise wird ein Algorithmus **effizient** genannt, wenn seine Rechenzeit höchstens polynamial mit der Größe n der Eingabe wächst d.h. wenn sie in in $O(P(n))$ liegt, wobei P ein Polynom ist. Wenn der Grad des Polynoms hoch ist (z.B. $P(n) = n^{10}$) oder wenn der Koeffizient des höchsten Monoms groß ist (z.B. $P(n) = 10^6 n$) , wird so ein Algorithmus in der Praxis nicht wirklich effizient einsetzbar sein. Die Erfahrung zeigt aber, daß Probleme aus der Praxis, sofern sie überhaupt mit effizienten Algorithmen gelöst werden können, meist auch mit Algorithmen gelöst werden können, deren Rechenzeit durch ein Polynom niedrigen Grades beschränkt ist.

Abgesehen von dieser praktischen Erfahrung hat die Untersuchung von Problemen, zu denen kein Lösungsalgorithmus mit polynomialer Rechenzeit bekannt ist, zu einer reichhaltigen Theorie der Komplexität von Algorithmen geführt. Es gibt viele, auch praxisrelevante Probleme, zu denen kein Lösungsalgorithmus mit polynomialer Rechenzeit bekannt ist. Möglicherweise ist so ein Algorithmus einfach noch nicht entdeckt worden. Wahrscheinlicher ist jedoch, daß es keinen effizienten Lösungsalgorithmus gibt.

Probleme lassen sich in gewissem Sinne in Klassen gleich schwieriger Probleme einteilen. Neben der Klasse P der in polynomialer Zeit lösbaren Probleme ist die bekannteste die Klasse NP der NP-vollständigen Probleme. Für sie gilt: Gibt es für irgendein NP-vollständiges Problem einen Lösungsalgorithmus mit polynomialer Rechenzeit, so auch für alle anderen. Und weil unter den NP-vollständigen Problemen welche zu finden sind, von denen eigentlich niemand glaubt, daß sie in polynomialer Rechenzeit zu lösen sind, nimmt man an, daß $\mathsf{P} \neq \mathsf{NP}$ gilt. Das ist aber bisher trotz vieler Anstrengungen noch nicht bewiesen, sondern lediglich eine Vermutung.

Die Bezeichnung NP-vollständig kommt von *nichtdeterministisch polynomial* her und nicht etwa von *nicht polynomial*, wie man meinen könnte. Wir wollen im Folgenden skizzieren, was es damit auf sich hat.

5.2 Nichtdeterministische Algorithmen

Wir hatten bisher nur Algorithmen betrachtet, bei denen zu jedem Zeitpunkt eindeutig festgelegt war, welcher Rechenschritt als nächster ausgeführt wird. Dies ist zur Lösung praktischer Probleme sicher sinnvoll und notwendig. Für theoretische Komplexitätsuntersuchungen dagegen haben sich nichtdeterministische Algorithmen als äußerst fruchtbar erwiesen.

Nichtdeterminismus bedeutet in diesem Zusammenhang nichts Geheimnisvolles, sondern lediglich, daß nicht immer nur genau ein Rechenschritt als nächster ausgeführt wird, sondern daß eventuell unter mehreren (aber stets nur endlich vielen) gewählt werden kann. Am einfachsten ist das vielleicht zu verstehen, wenn man einen nichtdeterministischen Algorithmen als nichtdeterministisches Programm einer Turingmaschine realisiert. Das ist ein Programm, bei dem mehrere Anweisungen dieselbe Kennmarke haben dürfen. Stößt man bei der Programmausführung auf eine Sprung- oder Verzweigungsmarke, die Kennmarke mehrerer Anweisungen ist, so darf man irgendeine dieser Anweisungen auswählen, um den Programmlauf fortzusetzen. Für jede Wahl ergibt sich also ein anderer Programmablauf. Es kann mehrere Stellen geben, an denen man so eine Wahl treffen muß. Die verschiedenen Programmabläufe, die auf diese Weise möglich sind, nennen wir die Rechnungen des Programms.

Man sagt nun, ein nichtdeterministischer Algorithmus löse ein vorgegebenes Problem, wenn es (mindestens) eine Rechnung gibt, die zu einer Lösung führt. Der Begriff Nichtdeterminismus ist hier also eher etwas irreführend, weil es sich nicht so sehr um eine Unbestimmtheit handelt, sondern mehr um eine Art von Parallelität.

Ein nichtdeterministischer Algorithmus hat natürlich mehr Chancen, durch eine geeignete Rechnung mit wenig Rechenschritten eine Lösung zu finden, als ein deterministischer. Man definiert als Rechenzeit $T(n)$, die ein nichtdeterministischer Algorithmus zur Lösung eines Problems mit Eingabegröße n benötigt, die minimale Anzahl von Rechenschritten, die eine Rechnung braucht, die nach endlich vielen Schritten zur Lösung führt.

5.2.1 Beispiel: Das Erfüllbarkeitsproblem der Aussagenlogik, SAT

Im Gegensatz zur Prädikatenlogik kommen in der Aussagenlogik keine Quantoren, Prädikate und Funktionen vor, sondern nur Variablen $v_j, j \in \mathbb{N}$, die die Werte *wahr* oder *falsch* annehmen können, und die logischen Operationen $\wedge$ (=*und*), $\vee$ (=*oder*) und $\neg$ (=*nicht*). Die aussagenlogischen Formeln werden induktiv konstruiert:

1. Jede Variable ist eine aussagenlogische Formel.
2. Ist A eine aussagenlogische Formel, so ist auch $\neg A$ eine aussagenlogische Formel (die oft mit $\overline{A}$ bezeichnet wird).
3. Sind A und B aussagenlogische Formeln, so sind auch $A \vee B$ und $A \wedge B$ aussagenlogische Formeln.

Um die Hierarchie des Aufbaus einer Formel deutlich zu machen, verwendet man runde Klammern wie bei arithmetischen Ausdrücken.

Beispiel: $(v_1 \vee v_2) \wedge (v_2 \vee v_4 \vee v_1) \wedge (\neg v_2 \vee v_4)$

Die Variablen spielen die Rolle von Platzhaltern für Aussagen im üblichen intuitiven Sinn.

Wahrheitswerte, die man den Variablen einer Formel zugeordnet hat, pflanzen sich über den hierarchischen Aufbau der Formel zu einem eindeutigen Wahrheitswert der Formel fort. Eine Formel heißt erfüllbar, wenn man ihren Variablen Wahrheitswerte so zuordnen kann, daß sie den Wert *wahr* bekommt.

Das **Erfüllbarkeitsproblem der Aussagenlogik (satisfiabilty problem, SAT)** besteht darin, einen möglichst schnellen Algorithmus zu finden, der für jede aussagenlogische Formel entscheiden kann, ob sie erfüllbar ist oder nicht. Daß es überhaupt einen Algorithmus gibt, der dies leistet, ist klar; man braucht ja nur für alle möglichen Zuordnungen von Wahrheitswerten zu den Variablen jeweils den Wahrheitswert der gesamten Formel auszurechnen. Kommen in der Formel n Variablen vor, so benötigt dieser Algorithmus im schlechtesten Fall $\Theta(2^n)$ viele Rechenschritte, also exponentiell wachsenden Aufwand.

Man kann daraus einen nichtdeterminischen Algorithmus machen, der polynomiale Laufzeit hat, indem man gleich zu Anfang 2^n mögliche Programmfortsetzungen vorsieht, nämlich für jede der 2^n möglichen Belegungen der Variablen mit Wahrheitswerten. Die weitere Programmausführung besteht in jedem dieser Fälle dann nur noch aus der Berechnung des Wahrheitswertes der gesamten Formel, die in linearer Zeit möglich ist.

Es ist eine offene Frage, ob es einen deterministischen Algorithmus gibt, der das SAT-Problem in polynomialer Zeit löst.

5.3 Polynomialzeitbeschränkte Reduktionen

Zur Vereinfachung beschränken wir uns auf Entscheidungsprobleme, bei denen die Antwort *ja* oder *nein* ist. Gegeben seien zwei Entscheidungsprobleme, die E_1 bzw. E_2 als Menge der zulässigen Eingaben haben. $Y_i \subset E_i$ bestehe genau aus den Eingaben mit positiver Antwort. Jeder zulässigen Eingabe sei eine natürliche Zahl als Größe zugeordnet.

Das Poblem 1 heißt in Polymialzeit auf das Problem 2 reduzierbar, wenn es einen Algorithmus A mit folgenden Eigenschaften gibt:

- A berechnet für jedes $x_1 \in E_1$ ein $x_2 \in E_2$, so daß $x_1 \in Y_1$ genau dann gilt, wenn $x_2 \in Y_2$.
- Die Rechenzeit von A ist nach oben durch ein Polynom in der Größe der Eingabe x_1 beschränkt.

5.3.1 Folgerung Ist Problem 1 in Polynomialzeit auf Problem 2 reduzierbar und gibt es für Problem 2 einen Lösungsalgorithmus mit polynomialer Rechenzeit, so auch für Problem 1.

5.4 NP-Vollständigkeit

P sei die Klasse der Probleme, für die es einen deterministischen Lösungsalgorithmus mit höchstens polynomialer Rechenzeit gibt.

NP sei die Klasse der Probleme, für die es einen nichtdeterministischen Lösungalgorithmus mit höchstens polynomialer Rechenzeit gibt.

Es gilt natürlich $\mathsf{P} \subset \mathsf{NP}$. Es ist naheliegend zu vermuten, daß die Gleichheit nicht gilt. Dies ist aber bis heute nicht bewiesen, sondern eine offene Frage, an der sich schon viele die Zähne ausgebissen haben.

Bei der Untersuchung von P und NP entstand der Begriff der NP-vollständigen Probleme, die in gewissem Sinn typisch für die Klasse NP sind.

5.4.1 Definition

Ein Problem X heißt NP-hart, wenn jedes Problem in NP in Polynomialzeit auf X reduzierbar ist.

Wenn auch nur von einem NP-harten Problem nachgewiesen wird, daß es in Polynomialzeit lösbar ist, so folgt aus 5.3.1, daß $\mathsf{P} = \mathsf{NP}$ gilt.

5.4.2 Definition

Ein Problem X heißt NP-vollständig wenn X NP-hart ist und in NP liegt.

Die NP-vollständigen Probleme sind alle in Polynomialzeit aufeinander reduzierbar und somit in gewissem Sinn gleich schwierig. Mittlerweile sind viele hundert Probleme ganz unterschiedlicher Art als NP-vollständig nachgewiesen. Einige wohlbekannte wollen wir zitieren.

5.4.3 NP-vollständige Probleme

5.4.3.1 SAT

Entscheide für jede aussagenlogische Formel, ob sie erfüllbar ist.

5.4.3.2 Hamilton-Kreis

Entscheide für jeden Graphen, ob er einen Hamiltonschen Kreis besitzt, d.h. ob er einen geschlossenen Kantenweg besitzt, der durch jeden Knoten genau einmal geht.

5.4.3.3 Das Problem des Handlungsreisenden

Finde in jedem gewichteten, vollständigen Graphen eine kürzeste Rundreise, die jeden Knoten genau einmal besucht.

5.4.3.4 Färbungsproblem

Entscheide für jeden ungerichteten Graphen, ob die Knoten mit drei Farben so eingefärbt werden können, daß adjazente Knoten stets unterschiedliche Farben haben.

5.4.3.5 Partitionsproblem

Entscheide für jede endliche Menge $A \subset \mathbb{Z}$, ob es eine Teilmenge $B \subset A$ gibt, so daß

$$\sum_{x \in B} x = \sum_{x \in A \setminus B} x$$

Anhang A

Mathematische Ergänzungen

A.1 Einige Grundlagen

A.1.1 Die Logarithmusfunktion

Für $b > 0$ bezeichne $\log_b\colon \mathbb{R}^+ \to \mathbb{R}$ die Logarithmusfunktion zur Basis b; das ist die Umkehrfunktion der Exponentialfunktion $x \mapsto b^x$ d.h. es gilt $\log_b y = x$ genau dann, wenn $y = b^x$ gilt.

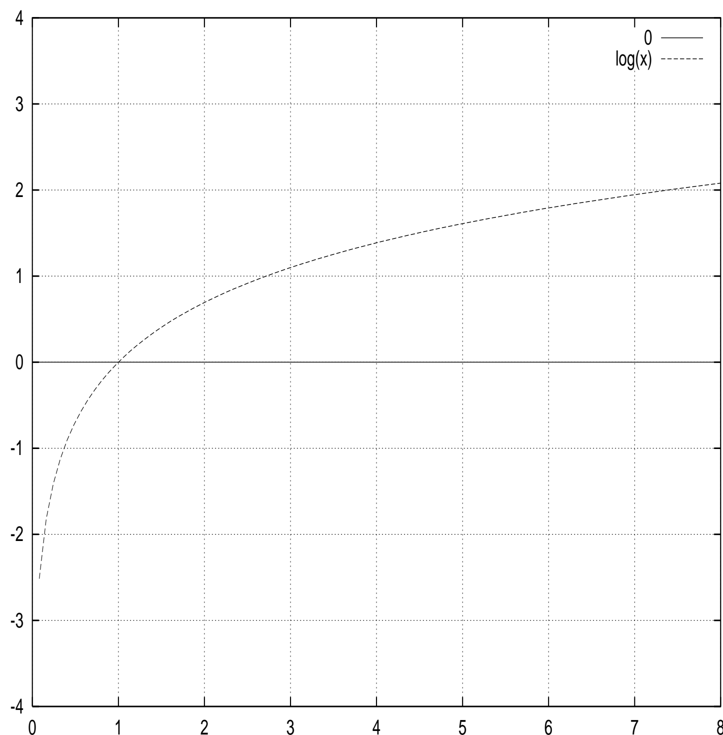
Oft schreibt man *ln* statt log_e und *lg* statt $\log_2$. Falls die Basis b in einem Kontext keine Rolle spielt, schreiben wir einfach log.

Es gilt $\log_a x = \frac{\log_b x}{\log_b a} = \frac{\ln x}{\ln a}$.

Die Logarithmusfunktion ist monoton wachsend und stetig, ja sogar beliebig oft differenzierbar. Für die Ableitung gilt $\log_b'(x) = \frac{1}{x \ln b}$.

Für jedes $b > 0$ gilt $\log_b 1 = 0$, $\lim_{x\to 0} \log_b x = -\infty$ und $\lim_{x\to\infty} \log_b x = \infty$.

Der Logarithmus erfüllt die Funktionalgleichung $\log_b(u \cdot v) = \log_b u \ + \ \log_b v$ für alle $u, v > 0$.



A.1.2 Die Gaußklammern

A.1.2.1 Definition

Untere Gaußklammer (floor): Für $x \in \mathbb{R}$ sei $\lfloor x \rfloor := \max\{a \in \mathbb{Z}\colon a \leq x\}$.
Obere Gaußklammer (ceiling): Für $x \in \mathbb{R}$ sei $\lceil x \rceil := \min\{a \in \mathbb{Z}\colon a \geq x\}$.

A.1.2.2 Lemma

1. Für $x \in \mathbb{R}$ gilt $x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$

2. Für $a > 0, x \in \mathbb{R}$ gilt $a \cdot \lfloor \frac{x}{a} \rfloor \le x$ und $a \cdot \lceil \frac{x}{a} \rceil \ge x$

3. Für $x \in \mathbb{R}$ gilt $\lfloor -x \rfloor = -\lceil x \rceil$ und $\lceil -x \rceil = -\lfloor x \rfloor$

4. Für $n \in \mathbb{Z}$ gilt

$$\left\lfloor \frac{n}{2} \right\rfloor = \begin{cases} \frac{n}{2} & \text{, falls } n \text{ gerade} \\ \frac{n-1}{2} & \text{, falls } n \text{ ungerade} \end{cases} \qquad \left\lceil \frac{n}{2} \right\rceil = \begin{cases} \frac{n}{2} & \text{, falls } n \text{ gerade} \\ \frac{n+1}{2} & \text{, falls } n \text{ ungerade} \end{cases}$$

und

$$\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n$$

5. Für $n, a, b \in \mathbb{Z}$ mit $a \neq 0$ und $b > 0$ gilt

$$\left\lceil \frac{\lceil \frac{n}{a} \rceil}{b} \right\rceil = \left\lceil \frac{n}{ab} \right\rceil$$

6. Für jedes $n \in \mathbb{N}$ der Gestalt $n = 2^k + 1$ mit $k \in \mathbb{N}$ gilt

$$\left\lceil \log_2 \left\lfloor \frac{n}{2} \right\rfloor \right\rceil = \left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil - 1$$

Für jedes andere $n \in \mathbb{N}$ gilt

$$\left\lceil \log_2 \left\lfloor \frac{n}{2} \right\rfloor \right\rceil = \left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil$$

7. Für jedes $n \in \mathbb{N}, n > 1$, gilt

$$\left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil = \lceil \log_2 n \rceil - 1$$

8. Für $x \ge 0, m \in \mathbb{Z}, n \in \mathbb{N}$ gilt

$$\begin{aligned} \left\lfloor \frac{x+m}{n} \right\rfloor &= \left\lfloor \frac{\lfloor x \rfloor + m}{n} \right\rfloor \\ \left\lceil \frac{x+m}{n} \right\rceil &= \left\lceil \frac{\lceil x \rceil + m}{n} \right\rceil \end{aligned}$$

9. Für $m, n \in \mathbb{Z}$ gilt

$$\begin{aligned} \left\lfloor \frac{n+m}{2} \right\rfloor + \left\lfloor \frac{n-m+1}{2} \right\rfloor &= n \\ \left\lceil \frac{n+m}{2} \right\rceil + \left\lceil \frac{n-m+1}{2} \right\rceil &= n+1 \end{aligned}$$

10. Die Funktion $f\colon \mathbb{R} \to \mathbb{R}$ sei stetig und streng monoton wachsend, und es sei $f^{-1}(\mathbb{Z}) \subset \mathbb{Z}$. Dann gilt für jedes $x \in \mathbb{R}$
$$\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor \quad \text{und} \quad \lceil f(x) \rceil = \lceil f(\lceil x \rceil) \rceil$$

11. Für $x \geq 0$ gilt $\lfloor \sqrt{x} \rfloor = \left\lfloor \sqrt{\lfloor x \rfloor} \right\rfloor$ und $\lceil \sqrt{x} \rceil = \left\lceil \sqrt{\lceil x \rceil} \right\rceil$.

12. Für $n \in \mathbb{N}$ ist $\lfloor \log_2 n \rfloor + 1$ die Länge der Dualdarstellung von n ohne führende Nullen.

A.2 Der Kalkül mit den Symbolen O, Θ, Ω

A.2.1 Definition $D \subset \mathbb{R}^+$ sei eine nichtleere, unbeschränkte Teilmenge.

a) Eine Funktion $f\colon D \to \mathbb{R}$ heißt bei ∞ positiv bzw. nichtnegativ, wenn es ein $x_0 \in \mathbb{R}$ gibt, so daß $f(x) > 0$ bzw. $f(x) \geq 0$ für jedes $x \in D$ mit $x \geq x_0$.

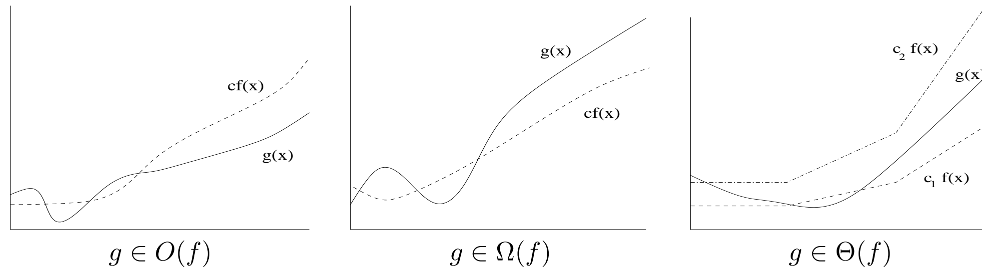
b) Sei $f\colon D \to \mathbb{R}$ bei ∞ nichtnegativ. Dann definiert man
$$O(f) = \{\, g\colon D \to \mathbb{R} \mid \exists\, c > 0\ \exists\, x_0 \in \mathbb{R}\ \forall\, x \in D \qquad x > x_0 \Rightarrow 0 \leq g(x) \leq c \cdot f(x) \,\}$$
d.h. $O(f)$ ist die Menge aller Funktionen $g\colon D \to \mathbb{R}$ derart, daß ein $c > 0$ und ein $x_0 \in \mathbb{R}$ existieren, so daß für alle x in D, die größer als x_0 sind, $0 \leq g(x) \leq c \cdot f(x)$ gilt.

$$\Omega(f) = \{\, g\colon D \to \mathbb{R} \mid \exists\, c > 0\ \exists\, x_0 \in \mathbb{R}\ \forall x \in D \qquad x > x_0 \Rightarrow 0 \leq c \cdot f(x) \leq g(x) \,\}$$
d.h. $\Omega(f)$ ist die Menge aller Funktionen $g\colon D \to \mathbb{R}$ derart, daß ein $c > 0$ und ein $x_0 \in \mathbb{R}$ existieren, so daß für alle x in D, die größer als x_0 sind, $0 \leq c \cdot f(x) \leq g(x)$ gilt.

$$\Theta(f) = \{\, g\colon D \to \mathbb{R} \mid \exists\, c_1, c_2 > 0\ \exists\, x_0 \in \mathbb{R}\ \forall x \in D \qquad x > x_0 \Rightarrow 0 \leq c_1 \cdot f(x) \leq g(x) \leq c_2 \cdot f(x) \,\}$$
d.h. $\Theta(f)$ ist die Menge aller Funktionen $g\colon D \to \mathbb{R}$ derart, daß $c_1 > 0, c_2 > 0$ und ein $x_0 \in \mathbb{R}$ existieren, so daß für alle x in D, die größer als x_0 sind, $0 \leq c_1 \cdot f(x) \leq g(x) \leq c_2 \cdot f(x)$ gilt.

Die Mengen $O(f), \Omega(f), \Theta(f)$ werden auch Wachstumsklassen genannt.

Die folgenden Diagramme sollen illustrieren, daß das Verhalten für kleine Argumente keine Rolle spielt.



$g \in O(f)$ $\qquad$ $g \in \Omega(f)$ $\qquad$ $g \in \Theta(f)$

Bemerkung: Bei der Rechenzeitanalyse von Algorithmen ist $D = \mathbb{N}$ und x beschreibt die Problemgröße.

A.2.2 Sprechweisen

$$\begin{aligned} g \in O(f) &\iff g \text{ ist in Groß-}O\text{ von } f \\ &\iff g \text{ wächst asymptotisch höchstens so stark wie } f \\ &\iff g \text{ ist höchstens in der Größenordnung von } f \end{aligned}$$

$$\begin{aligned} g \in \Omega(f) &\iff g \text{ ist in Groß-}\Omega\text{ von } f \\ &\iff g \text{ wächst asymptotisch mindestens so stark wie } f \\ &\iff g \text{ ist mindestens in der Größenordnung von } f \end{aligned}$$

$$\begin{aligned} g \in \Theta(f) &\iff g \text{ ist in Groß-}\Theta\text{ von } f \\ &\iff g \text{ wächst asymptotisch genau so so stark wie } f \\ &\iff g \text{ ist genau in der Größenordnung von } f \end{aligned}$$

Die Werte von f und g in einem beschränkten Intervall spielen keine Rolle.

A.2.3 Schreibweisen Folgende Schreibweisen sind üblich:
$g = O(f)$ statt $g \in O(f)$, analog $g = \Omega(f)$ statt $g \in \Omega(f)$ und ebenso $g = \Theta(f)$ statt $g \in \Theta(f)$.

Ist f ein Monom, also $f(x) = x^k$ für $x \in D$, so schreibt man meist $O(x^k)$ statt $O(f)$ (entsprechend für Θ und Ω). Ähnlich geht man bei anderen Funktionen vor z.B. $O(x \log x), O(2^x)$. Der Name der Variablen ist unbedeutend; es muß nur klar sein, bezüglich welcher Variablen die asymptotische Betrachtung angestellt wird. Bei der Rechenzeitanalyse ist $D = \mathbb{N}$, und die Variable wird meist n genannt. Man schreibt dann also $O(n^k), O(n \log n)$ usw.

A.2.4 Beispiele

1) Meist bezeichnet man die Funktion, die konstant gleich einem Wert $a > 0$ ist, einfach wieder mit a. Dann ist $O(a)$ die Menge aller Funktionen $g \colon D \to \mathbb{R}$, die außerhalb eines beschränkten Intervalls beschränkt sind oder etwas lax formuliert bei ∞ beschränkt sind. Der Wert von $a > 0$ spielt keine Rolle, alle $O(a)$ sind gleich; daher wählt man meist $a = 1$ und schreibt $O(1)$.

2) Für $k \leq m$ ist $x^k \in O(x^m)$.
Beweis: Wegen $m - k \geq 0$ ist $\frac{1}{x^{m-k}} \leq 1$ für $x \geq 1$. Mit $c = 1$ und $x_0 = 1$ gilt also $0 \leq x^k \leq cx^m$ für $x \geq x_0$.

3) Für $k > m$ ist $x^k \notin O(x^m)$, aber $x^k \in \Omega(x^m)$.
Beweis: Denn x^{k-m} strebt monoton wachsend gegen ∞ für $x \to \infty$.

4) Für $l < k$ ist $x^l + x^k \in \Theta(x^k)$.
Beweis: Wegen $x^l \in O(x^k)$ gibt es $c > 0$ und $x_0 > 0$, so daß $x^l \leq cx^k$ für $x \geq x_0$. Mit $c_1 = 1$ und $c_2 = 1 + c$ folgt $0 \leq c_1 x^k \leq x^k + x^l \leq c_2 x^k$ für $x \geq x_0$.

A.2.5 Bezeichnungen h, f, f_1, f_2 seien Funktionen, die bei ∞ nichtnegativ sind. Dann setzt man

$$\begin{aligned} h + O(f) &:= \{\, h + g \colon g \in O(f) \,\} \\ O(f_1) + O(f_2) &:= \{ g_1 + g_2 \colon g_1 \in O(f_1) \text{ und } g_2 \in O(f_2) \,\} \\ O(f_1) \cdot O(f_2) &:= \{\, g_1 \cdot g_2 \colon g_1 \in O(f_1) \text{ und } g_2 \in O(f_2) \,\} \end{aligned}$$

Entsprechend für die Symbole Ω und Θ und für andere Rechenoperationen als $\cdot$ und $+$.

Eine Gleichung, auf deren beiden Seiten Wachstumsklassen stehen, ist als Gleichheit von Funktionenmengen zu verstehen. Infolgedessen ist die Gleichung $x^2 + O(x) = O(x^2)$ **falsch!** Denn es gilt zwar $x^2 + O(x) \subset O(x^2)$, aber nicht $x^2 + O(x) \supset O(x^2)$, weil z.B. die Funktion $2x^2$ in $O(x^2)$ liegt, aber nicht in der Form $x^2 + h(x)$ mit $h \in O(x)$ dargestellt werden kann.

Achtung! In der Literatur findet sich auch eine andere Interpretation, nämlich daß eine Gleichung zwischen Wachstumsklassen lediglich als Inklusion der linken in der rechten Seite zu verstehen ist; dann ist also z.B. $x^2 + O(x) = O(x^2)$ als $x^2 + O(x) \subset O(x^2)$ zu interpretieren. **Dies ist ein sehr verwirrender Gebrauch des Gleichheitszeichens, denn man darf solche "Gleichungen" stets nur von links nach rechts lesen!** Obwohl diese Verwendung des Gleichheitszeichens sehr üblich ist, werden wir hier meist die klarere Schreibweise mit $\in$ und $\subset$ verwenden.

A.2.6 Lemma f, g, h seien bei ∞ nichtnegativ.

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

b) $g \in O(f) \iff f \in \Omega(g)$

c) $g \in \Theta(f) \iff f \in \Theta(g) \iff \Theta(f) = \Theta(g)$

d) $f \in O(g) \Longrightarrow O(f) \subset O(g)$ und $f \in \Omega(g) \Longrightarrow \Omega(f) \subset \Omega(g)$

e) $O(f) = O(g) \iff \Theta(f) = \Theta(g) \iff \Omega(f) = \Omega(g)$

f) $O(a \cdot f) = O(f)$ für $a > 0$

g) $O(f + g) = O(f) + O(g) = O(\max\{f, g\})$ und
$\Omega(f + g) = \Omega(f) + \Omega(g) \subset \Omega(\min\{f, g\})$

h) $\Theta(f+g) = \Theta(f) + \Theta(g) = \Theta(\max\{f,g\})$

i) $O(f \cdot g) = O(f) \cdot O(g)$

j) $g \in \Theta(x^k)$ für jedes Polynom $g(x) = x^k + \sum_{j=0}^{k-1} a_j x^j$.

Beweis:
a) Sei $h \in \Theta(f)$. Dann gibt es $c_1, c_2 > 0, x_0 \in \mathbb{R}$ mit $c_1 f(x) \le h(x) \le c_2 f(x)$ für alle $x > x_0, x \in D$. Folglich $h \in O(f)$ und $h \in \Omega(f)$.
Sei $h \in O(f) \cap \Omega(f)$. Dann gibt es $c_1 > 0$ und $x_1 \in \mathbb{R}$, so daß $c_1 f(x) \le h(x)$ für $x > x_1$, und $c_2 > 0$ und $x_2 \in \mathbb{R}$, so daß $h(x) \le c_2 f(x)$ für $x > x_2$. Setze $x_0 = \max\{x_1, x_2\}$. Damit ist die Definition von $h \in \Theta(f)$ erfüllt.

b) folgt direkt aus den Definitionen.

c) Sei $g \in \Theta(f)$. Dann gibt es $c_1, c_2 > 0$ und $x_0 \in \mathbb{R}$ mit $c_1 f(x) \le g(x) \le c_2 f(x)$ für $x > x_0, x \in D$. Daraus folgt $\frac{1}{c_2} g(x) \le f(x) \le \frac{1}{c_1}$ für $x > x_0$, also $f \in \Theta(g)$. Die Umkehrung ist symmetrisch dazu.

Zweite Äquivalenz: $\Leftarrow$ ist trivial.
$\Rightarrow$: Es gelte $f \in \Theta(g)$. Wir zeigen $\Theta(f) \subset \Theta(g)$. Sei $h \in \Theta(f)$. Dann gibt es $c_1 > 0, c_2 > 0$ und $x_0 \in \mathbb{R}$, so daß $c_1 f(x) \le h(x) \le c_2 f(x)$ für $x > x_0$. Wegen $f \in \Theta(g)$ gibt es $a_1 > 0, a_2 > 0$ und $x_1 \in \mathbb{R}$, so daß $a_1 g(x) \le f(x) \le a_2 g(x)$ für $x > x_1$. Zusammengesetzt erhält man $c_1 a_1 g(x) \le h(x) \le c_2 a_2 g(x)$ für $x > \max\{x_0, x_1\}$, also $h \in \Theta(g)$.
Die umgekehrte Inklusion $\Theta(g) \subset \Theta(f)$ ergibt sich entsprechend, weil $f \in \Theta(g) \Rightarrow g \in \Theta(f)$

d) folgt direkt aus den Definitionen.

e) Erste Äquivalenz:

$$\begin{aligned} O(f) = O(g) &\iff f \in O(g) \text{ und } g \in O(f) \\ &\overset{b)}{\iff} f \in O(g) \text{ und } f \in \Omega(g) \\ &\iff f \in O(g) \cap \Omega(g) \\ &\overset{a)}{\iff} f \in \Theta(g) \\ &\overset{c)}{\iff} \Theta(f) = \Theta(g) \end{aligned}$$

Zweite Äquivalenz: analog.

f) folgt direkt aus den Definitionen.

g), h), i), j) als Übung.

q.e.d.

A.2.7 Lemma (Vergleich des asymptotischen Verhaltens mit Hilfe von Grenzwerten
Seien $f \colon D \to \mathbb{R}$ bei ∞ nichtnegativ und $g \colon D \to \mathbb{R}$ bei ∞ positiv. Dann gilt:

a) $f \in O(g) \iff$ es gibt ein $x_0 \in \mathbb{R}$, so daß $\left\{ \frac{f(x)}{g(x)} \colon x \in D, x \ge x_0 \right\}$ beschränkt ist.

b) $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ existiert und ist positiv $\implies$ $O(f) = O(g)$ und $\Theta(f) = \Theta(g)$

c) $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \quad \Longrightarrow \quad f \in O(g)$ und $g \notin O(f)$

d) $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty \quad \Longrightarrow \quad f \in \Omega(g)$ sowie $f \notin \Theta(g)$ und $f \notin O(g)$

Beweis:

a) Weil g bei ∞ positiv ist, gibt es ein $x_1 \in \mathbb{R}$, so daß $g(x) > 0$ für $x > x_1$.

$f \in O(g) \iff$ es gibt $c > 0$ und $x_2 \in \mathbb{R}$, so daß $0 \le f(x) \le c\, g(x)$ für $x > x_2$

$\iff$ es gibt ein $c > 0$ und ein $x_3 \in \mathbb{R}$, so daß $0 \le \dfrac{f(x)}{g(x)} \le c$ für $x > x_3$

$\iff$ es gibt ein $x_0 \in \mathbb{R}$, so daß $\left\{ \dfrac{f(x)}{g(x)} : x \in D, x > x_0 \right\}$ beschränkt ist

b) Sei $a := \lim_{x\to\infty} \frac{f(x)}{g(x)} > 0$. Dann gibt es ein $x_0 \in \mathbb{R}$, so daß $\left| \frac{f(x)}{g(x)} - a \right| < \frac{a}{2}$ für $x \ge x_0$, also $\frac{a}{2} < \frac{f(x)}{g(x)} < \frac{3}{2}\, a$ und $\frac{a}{2}\, g(x) < f(x) < \frac{3}{2}\, a\, g(x)$ für $x \ge x_0$.
Folglich gilt $f \in \Theta(g)$. Mit A.2.6c) und e) folgt $\Theta(f) = \Theta(g)$ und $O(f) = O(g)$.

d) Wegen $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$ gibt es zu jedem $n \in \mathbb{N}$ ein $x_n \in \mathbb{R}$, so daß $\frac{f(x)}{g(x)} \ge n$ für $x > x_n$. Dies zeigt $f \in \Omega(g)$ und $f \notin \Theta(g)$. q.e.d.

Der Limes $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ läßt sich oft ausrechnen mit Hilfe folgender

A.2.8 Regel von de l'Hospital (1661-1704)

$f, g\colon]a, \infty[\to \mathbb{R}$ seien differenzierbare Funktionen. Es gelte $g(x) \ne 0$ für $x \in]a, \infty[$ und $\lim_{x\to\infty} f(x) = \infty$ sowie $\lim_{x\to\infty} g(x) = \infty$. Wenn dann $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ existiert (eigentlich oder uneigentlich), so existiert auch $\lim_{x\to\infty} \frac{f(x)}{g(x)}$, und es gilt

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

Die Regel läßt sich erweitern auf den Fall, daß auch die Ableitungen von f und g für $x \to \infty$ gegen ∞ streben:

$f\colon]a, \infty[\to \mathbb{R}$ und $g\colon]a, \infty[\to \mathbb{R}$ seien n-mal stetig differenzierbare Funktionen. Für jedes $k = 0, \ldots, n-1$ gelte $g^{(k)}(x) \ne 0$ für $x \in]a, \infty[$ und $\lim_{x\to\infty} f^{(k]}(x) = \infty$ sowie $\lim_{x\to\infty} g^{(k)}(x) = \infty$. Wenn dann $\lim_{x\to\infty} \frac{f^{(n)}(x)}{g^{(n)}(x)}$ existiert (eigentlich oder uneigentlich), so existiert auch $\lim_{x\to\infty} \frac{f(x)}{g(x)}$, und es gilt

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f^{(n)}(x)}{g^{(n)}(x)}$$

Dabei bezeichnet $f^{(k)}$ die k-te Ableitung $\frac{d^k f}{dx^k}$.

A.2.9 Beispiele

1. Es gilt $\log_b x \in O(x)$ für jedes $b > 1$ (eigentlich sollte man $\log_b \in O(id)$ schreiben).

   Beweis: Mit A.2.8 folgt $\lim_{x\to\infty} \frac{\log_b(x)}{x} = \lim_{x\to\infty} \frac{\log_b'(x)}{1} = \lim_{x\to\infty} \frac{1}{x \ln b} = 0$. Mit A.2.7 c) folgt die Behauptung.

2. Für $0 < \alpha \leq \beta$ gilt $x^\alpha \in O(x^\beta)$.

   Beweis: $\frac{x^\alpha}{x^\beta} = \frac{1}{x^{\beta-\alpha}}$ bleibt beschränkt für $x \to \infty$. Mit A.2.7 a) folgt die Behauptung.

3. Für $\alpha > 0$ und $a > 1$ gilt $x^\alpha \in O(a^x)$.

   Beweis: Folgt mit der erweiterten Hospitalschen Regel und A.2.7 c).

4. Für $a > 1$ gilt

$$O(\log x) \subset O(\sqrt{x}) \subset O(x) \subset O(x\ \log x) \subset O(x^2) \subset O(x^x)$$

   Alle Inklusionen sind echt (also keine Gleichheiten).

   Beweis: Folgt mit der erweiterten Hospitalschen Regel und A.2.7.

A.2.10 Lemma Sei $D = \mathbb{N}$.

a) Für $f(n) = \sum_{m=1}^{n} \frac{1}{m}$ gilt $f \in \Theta(\log)$.

b) Für $\alpha > 0$ und $f(n) = \sum_{m=1}^{n} m^\alpha$ gilt $f \in \Theta(n^{\alpha+1})$.

c) $\log(n!) \in \Theta(n \log n)$.

Beweis:
a) Indem man die Summe als Untersumme eines Integrals auffaßt, ergibt sich $f(n) = 1 + \sum_{m=2}^{n} \frac{1}{m} \leq 1 + \int_1^n \frac{1}{t} dt = 1 + \log n$. Weil $1 + \log \in O(\log n)$, ist folglich auch $f \in O(\log)$.

Umgekehrt kann man die Summe auch als Obersumme auffassen und erhält $f(n) = \sum_{m=1}^{n-1} \frac{1}{m} + \frac{1}{n} \geq \int_1^n \frac{1}{t} dt + \frac{1}{n} \geq \log n$, also $f \in \Omega(\log n)$.

b) Indem man die Summe als Untersumme eines Integrals auffaßt, ergibt sich $f(n) = \sum_{m=1}^{n-1} m^\alpha + n^\alpha \leq \int_1^n t^\alpha dt + n^\alpha = \frac{1}{\alpha+1}(n^{\alpha+1} - 1) + n^\alpha$, also $f \in O(n^{\alpha+1})$.

Umgekehrt kann man die Summe auch als Obersumme auffassen und erhält $f(n) = \sum_{m=2}^{n} m^\alpha + 1 \geq \int_1^n t^\alpha dt = \frac{1}{\alpha+1}(n^{\alpha+1} - 1)$, also $f \in \Omega(n^{\alpha+1}$.

c) Sei $n > 1$. Schätzt man $\int \log t dt$ durch Unter- und Obersummen ab, so erhält man

$$\sum_{j=1}^{n-1} \log j \leq \int_1^n \log t\ dt \leq \sum_{j=1}^{n} \log j$$

Den linken und den rechten Term kann man mit Hilfe der Funktionalgleichung des Logarithmus umwandeln, den mittleren Integralterm kann man ausrechnen, weil man

zu $\log t$ eine Stammfunktion, nämlich $t \log t - t$, kennt. Damit erhält man die folgenden Ungleichungen

$$\log\left(\prod_{j=1}^{n-1} j\right) \leq [t\ \log t\ - t]_1^n \leq \log\left(\prod_{j=1}^{n} j\right)$$

und somit

$$\log((n-1)!) \overset{(1)}{\leq} n\ \log n\ - n + 1 \overset{(2)}{\leq} \log(n!)$$

Daraus folgt

$$\log(n!) = \log((n-1!) + \log n \leq \log((n-1)!) + n \overset{\text{mit (1)}}{\leq} n\ \log n + 1 \leq 2n \log n$$

also $\log(n!) \in O(n \log n)$.

Außerdem folgt

$$n \log n \overset{\text{mit (2)}}{\leq} \log(n!) + n - 1 \in O(\log(n!) + n - 1) = O(\log(n!))$$

denn $n - 1 \leq n - b + b = \log_b b^{n-b} + b \leq \log_b(n!) + b$, also $n - 1 \in O(\log_b(n!))$.

Aus A.2.6 a) und b) folgt schließlich $\log(n!) \in \Theta(n \log n)$. q.e.d.

A.2.11 Lemma Seien $D = \mathbb{N}$ und $f, g\colon \to \mathbb{R}^+$ monton wachsende Funktionen. Es gebe $a > 0, b > 0, c > 0$, so daß für alle $n \in \mathbb{N}$ gilt:

(1) $a\ g(n) \leq g(2n)$

(2) $f(2n) \leq b\ f(n) + c\ g(n)$

(3) $b < a$

Dann ist $f \in O(g)$.

Beweis: Setze $h(n) = \frac{f(n)}{g(n)}$. Sei $k \in \mathbb{Z}, k \geq 0$.

1. Schritt: Für $2^k < n \leq 2^{k+1}$ gilt

$$h(n) = \frac{f(n)}{g(n)} \leq \frac{f(2^{k+1})}{g(2^k)} \overset{(2)}{\leq} \frac{b\ f(2^k) + c\ g(2^k)}{g(2^k)} = b\ h(2^k) + c$$

2. Schritt: Für $k \geq 1$ gilt

$$h(2^k) = \frac{f(2^k)}{g(2^k)} \overset{(1)(2)}{\leq} \frac{b\ f(2^{k-1}) + c\ g(2^{k-1})}{a\ g(2^{k-1})} = \frac{b}{a} h(2^{k-1}) + \frac{c}{a}$$

3. Schritt: Aus Schritt 2 folgt

$$h(2^k) \leq \left(\frac{b}{a}\right)^k h(1) + \frac{c}{a}\sum_{j=0}^{k-1}\left(\frac{b}{a}\right)^j \leq \left(\frac{b}{a}\right)^k h(1) + \frac{c}{a}\sum_{j=0}^{\infty}\left(\frac{b}{a}\right)^j = \left(\frac{b}{a}\right)^k h(1) + \frac{c}{a-b}$$

4. Schritt: Aus Schritt 1 und 3 folgt für $n > 2$ und $k \geq 1$

$$h(n)\ \leq\ b\ h(2^k) + c\ \leq\ b\left(\frac{b}{a}\right)^k h(1) + \frac{bc}{a-b} + c\ \leq\ b\ h(1) + \frac{bc}{a-b} + c$$

also $f \in O(g)$. q.e.d.

Anhang B

Lineare Rekursionsgleichungen

B.1 Einführung

B.1.1 Definition

Seien $k \in \mathbb{N},\ a_0, \ldots, a_k \in \mathbb{R}, a_0 \neq 0, a_k \neq 0,\ f\colon \mathbb{N} \cup \{0\} \to \mathbb{R}$ mit $f \not\equiv 0$. Dann heißt

$$\sum_{j=0}^{k} a_j T(n-j) = f(n) \tag{B.1}$$

eine **inhomogene lineare Rekursionsgleichung mit konstanten Koeffizienten von der Ordnung k für T** und

$$\sum_{j=0}^{k} a_j T(n-j) = 0 \tag{B.2}$$

eine **homogene lineare Rekursionsgleichung mit konstanten Koeffizienten von der Ordnung k für T**, die sogenannte zu B.1 **zugehörige homogene Gleichung**.
Jede Funktion $T\colon \mathbb{N} \cup \{0\} \to \mathbb{C}$, die für $n \geq k$ eine dieser Gleichungen erfüllt, heißt eine Lösung. Die Werte $T(0), \ldots, T(k-1)$ einer Lösung T heißen die **Anfangswerte** von T.

Rekursionsgleichungen tauchen in der Informatik immer wieder auf, wenn rekursive Strukturen betrachtet werden. Sie sind nicht immer linear, aber in manchen wichtigen Fällen wie z.B. der Rechenzeit von *Divide-et-Impera*-Strategien kann man die vorkommenden Rekursionsgleichungen in lineare transformieren. Lineare Rekursionsgleichungen haben den Vorteil, daß man eine einigermaßen systematische Lösungstheorie für sie hat.

Aber auch in anderen Bereichen stößt man häufig auf Rekursionsgleichungen; denn sie sind ein diskretes Analogon zu gewöhnlichen Differentialgleichungen und darauf

basieren sehr viele Modellbildungen zur Beschreibung von Prozessen in Natur und Technik. Vielfach kann man sich auch hier mit mehr oder weniger Berechtigung auf lineare Rekursionsgleichungen beschränken. Ein klassisches Beispiel dafür ist die lineare Verarbeitung eindimensionaler Signale (wie z.B. Audiosignale); sie hat eine herausragende Bedeutung in der gesamten heutigen Technik. Die linearen Rekursionsgleichungen tauchen dort als **rekursive Digitalfilter** auf. Ein sehr einfaches Beispiel möge einen Eindruck geben.

B.1.2 Exponentiell abfallendes Mittel

$f\colon \mathbb{N} \cup \{0\} \to \mathbb{R}$ sei eine Funktion (z.B. ein zeitlich diskretes Audiosignal oder der Verlauf eines Aktienwertes oder einer Temperatur). Um schnelle Schwankungen von f zu dämpfen, kann man statt f die Funktion $g\colon \mathbb{N} \cup \{0\} \to \mathbb{R}$ betrachten, die folgendermaßen definiert ist:
Es sei $g(0) = f(0)$ und für $n > 0$ sei $g(n) = \frac{1}{2} f(n) + \frac{1}{2} g(n-1)$
g ist also eine Art gleitender Mittelwert aus dem momentanen Signalwert $f(n)$ und dem vorigen Wert $g(n-1)$. Man kann hier die Rekursion leicht auflösen

$$\begin{aligned} g(n) &= \tfrac{1}{2} f(n) + \tfrac{1}{2} g(n-1) \\ &= \tfrac{1}{2} f(n) + \tfrac{1}{2} \left(\tfrac{1}{2} f(n-1) + \tfrac{1}{2} g(n-2)\right) \\ &= \tfrac{1}{2} f(n) + \left(\tfrac{1}{2}\right)^2 f(n-1) + \left(\tfrac{1}{2}\right)^2 g(n-2) \\ &= \ldots \\ &= \tfrac{1}{2} f(n) + \ldots + \left(\tfrac{1}{2}\right)^n f(1) + \left(\tfrac{1}{2}\right)^n f(0) \\ &= \sum_{j=0}^{n-1} \left(\tfrac{1}{2}\right)^{j+1} f(n-j) + \left(\tfrac{1}{2}\right)^n f(0) \end{aligned}$$

Man beachte den hohen Rechenaufwand verglichen mit der rekursiven Darstellung.

In der Signalverarbeitung ist es nützlich, für die Koeffizienten auch komplexe Zahlen zuzulassen. Die Lösungstheorie ändert sich dadurch nicht. Weil jedoch bei der Rechenzeitanalyse von Algorithmen nur reelle Koeffizienten auftreten, wollen wir uns hier darauf beschränken.

B.2 Die Lösung homogener linearer Rekursionsgleichungen

Gegeben sei die homogene lineare Rekursionsgleichung der Ordnung k

$$\sum_{j=0}^{k} a_j T(n-j) \;=\; 0 \tag{B.3}$$

mit konstanten reellen Koeffizienten $a_0, \ldots a_k$, $a_0 \neq 0$, $a_k \neq 0$.

B.2.1 Satz

Der Raum $\mathcal{L}_\mathbb{C}$ aller Lösungen von B.3 ist ein k-dimensionaler Vektorraum über $\mathbb{C}$. Der Raum $\mathcal{L}_\mathbb{R}$ aller reelwertigen Lösungen von B.3 ist ein k-dimensionaler Vektorraum

über $\mathbb{R}$. Die Abbildungen $\mathcal{L}_{\mathbb{C}} \to \mathbb{C}^k$ und $\mathcal{L}_{\mathbb{R}} \to \mathbb{R}^k$, die jedem T die Anfangswerte $(T(0), \ldots, T(k-1))$ zuordnen, sind Isomorphismen.

Beweis: Durch Nachrechnen sieht man, daß jede Linearkombination von Lösungen wieder eine Lösung ist. $\mathcal{L}_{\mathbb{C}}$ und $\mathcal{L}_{\mathbb{R}}$ sind also Vektorräume. Die Abbildungen $\mathcal{L}_{\mathbb{C}} \to \mathbb{C}^k$, $T \mapsto (T(0), \ldots, T(k-1))$ und $\mathcal{L}_{\mathbb{R}} \to \mathbb{R}^k$, $T \mapsto (T(0), \ldots, T(k-1))$ sind offensichtlich linear. Wegen $a_0 \neq 0$ kann man B.3 umformen zu $T(n) = -\sum_{j=1}^{k} \frac{a_j}{a_0} T(n-j)$. Durch Induktion nach n sieht man:

1. Jede Lösung T ist vollständig durch ihre Anfangswerte $T(0), \ldots, T(k-1)$ bestimmt.
2. Zu jedem $(\alpha_0, \ldots, \alpha_{k-1}) \in \mathbb{C}^k$ gibt es eine Lösung, die die Anfangswertbedingungen $T(j) = \alpha_j$ für $j = 0, \ldots, k-a$ erfüllt.
3. Eine Lösung T ist genau dann reellwertig, wenn ihre Anfangswerte reell sind.

Das bedeutet gerade, daß $T \mapsto (T(0), \ldots, T(k-1))$ Isomorphismen $\mathcal{L}_{\mathbb{C}} \to \mathbb{C}^k$ bzw. $\mathcal{L}_{\mathbb{R}} \to \mathbb{R}^k$ sind. q.e.d.

B.2.2 Bemerkung

Ist $T \in \mathcal{L}_{\mathbb{C}}$, so ist auch $\overline{T} \in \mathcal{L}_{\mathbb{C}}$, denn es gilt

$$\sum_{j=0}^{k} a_j \overline{T}(n-j) = \sum_{j=0}^{k} \overline{a_j}\, \overline{T(n-j)} = \overline{\sum_{j=0}^{k} a_j T(n-j)} = 0$$

Folglich sind auch der Realteil $\Re(T) = \frac{1}{2}(T+\overline{T})$ und der Imaginärteil $\Im(T) = \frac{1}{2i}(T-\overline{T})$ Lösungen, und es ist $\Re(T) \in \mathcal{L}_{\mathbb{R}}$ und $\Im(T) \in \mathcal{L}_{\mathbb{R}}$. aus einer Basis $T_1, \ldots, T_k$ von $\mathcal{L}_{\mathbb{C}}$ erhält man also das Erzeugendensystem $\Re(T)_1, \ldots, \Re(T_k)$, $\Im(T_1), \ldots, \Im(T_k)$ von $\mathcal{L}_{\mathbb{R}}$, aus dem man eine Basis auswählen kann.

B.2.3 Heuristischer Lösungsansatz

Setze $T(n) = x^n$ in die Rekursionsgleichung B.3 ein. Dies liefert die Gleichungen

$$\sum_{j=0}^{k} a_j x^{n-j} \;=\; 0 \text{ für } n \geq k \tag{B.4}$$

Wegen $a_k \neq 0$ ist $x = 0$ keine Lösung. Ein $x \neq 0$ löst die Gleichung B.4 genau dann, wenn x die sogenannte **charakteristische Gleichung**

$$\sum_{j=0}^{k} a_j x^{k-j} \;=\; 0 \tag{B.5}$$

löst; denn die anderen Gleichungen in B.4 mit $n > k$ erhält man durch Multiplikation mit x^{n-k}. Das Polynom $\sum_{j=0}^{k} a_j x^{k-j}$ heißt das **charakteristische Polynom** der Rekursionsgleichung.

B.2.4 Satz
Gegeben sei die homogene lineare Rekursionsgleichung der Ordnung k

$$\sum_{j=0}^{k} a_j T(n-j) \;=\; 0 \tag{B.6}$$

mit konstanten reellen Koeffizienten $a_0, \dots a_k$, $a_0 \neq 0$, $a_k \neq 0$.
$\lambda_1, \dots, \lambda_L$ seien die Nullstellen des charakteristischen Polynoms $\sum_{j=0}^{k} a_j x^{k-j}$, und r_l sei die Vielfachheit von λ_l. (Man beachte, daß wegen $a_k \neq 0$ kein λ_j gleich 0 ist.)
Dann bilden die Funktionen $T_{l,m}$ mit $T_{l,m}(n) = n^m \lambda_l^n$ für $l \in \{1, \dots, L\}$ und $m \in \{0, \dots, r_l - 1\}$ eine Basis des Lösungsraumes $\mathcal{L}_{\mathbb{C}}$ von B.6.
Basis: $\lambda_1^n, n\lambda_1^n, \dots, n^{r_1-1}\lambda_1^n,\ \lambda_2^n, \dots, n^{r_2-1}\lambda_2^n,\ \dots\ , \lambda_L^n, n\lambda_L^n, \dots, n^{r_L-1}\lambda_L^n$

Spezialfall: Hat das charakteristische Polynom nur einfache Nullstellen, so ist $L = k$ und die Funktionen $\lambda_1^n, \dots, \lambda_k^n$ bilden eine Basis von $\mathcal{L}_{\mathbb{C}}$.

Beweisskizze: Die angegebenen Funktionen sind linear unabhängig (Beweis!). Zu zeigen ist nur, daß sie B.2 lösen. Sei λ eine Nullstelle des charakteristischen Polynoms mit der Vielfachheit r; es gibt also ein Polynom q, so daß $\sum_{j=0}^{k} a_j x^{k-j} = (x - \lambda)^r q(x)$. Es seien $0 \leq s < r$ und $T(n) = n^s \lambda^n$ für $n \in \mathbb{N} \cup \{0\}$. Wir definieren den Differentialoperator D durch $Df = x\frac{df}{dx}$.

Behauptung: Für jedes $j \in \mathbb{N} \cup \{0\}$ und $m \in \mathbb{N}$ gilt $D^j x^m := \underbrace{D \dots D}_{j-\text{mal}} x^m = m^j x^m$.

Beweis: durch vollständige Induktion nach j.
Induktionsanfang $j = 0:\ D^0 x^m = x^m = m^0 x^m$.
Induktionsschluß $j \mapsto j+1:$
$D^{j+1}x^m = DD^j x^m = x\frac{d}{dx}D^j x^m \overset{\text{Ind.vor.}}{=} x\frac{d}{dx}(m^j x^m) = x(m^{j+1}x^{m-1}) = m^{j+1}x^m$.

Wir zeigen nun, daß T die Rekursionsgleichung B.6 löst.
Für jedes $n \geq k$ gilt:

$$\begin{aligned}
\sum_{j=0}^{k} a_j (n-j)^s x^{n-j} &= \sum_{j=0}^{k} a_j D^s x^{n-j} \\
&= D^s \left(\sum_{j=0}^{k} a_j x^{n-j} \right) \\
&= D^s \left(x^{n-k} \sum_{j=0}^{k} a_j x^{k-j} \right) \\
&= D^s \left(x^{n-k} (x-\lambda)^r q(x) \right) \\
&= D^s \left((x-\lambda)^r x^{n-k} q(x) \right) \\
&= \sum_{j=0}^{s} \binom{s}{j} D^j \left((x-\lambda)^r \right) D^{s-j} \left(x^{n-k} q(x) \right) \quad \text{(Produktregel von Leibniz)}
\end{aligned}$$

Wegen $s < r$ enthält jeder Summand den Faktor $(x - \lambda)$. Infolgedessen ergibt sich

mit $x = \lambda$

$$\sum_{j=0}^{k} a_j T(n-j) \ = \ \sum_{j=0}^{k} a_j (n-j)^s \lambda^{n-j} \ 0 \ 0$$

d.h. T erfüllt die Rekursionsgleichung B.6. q.e.d.

B.2.5 Bemerkung

Um zu zeigen, daß die angegebenen Funktionen linear unabhängig sind, genügt es nachzuweisen, daß die Vektoren $\left(T_{l,m}(0), \ldots, T_{l,m}(k-1)\right)^\top$ linear unabhängig sind oder daß die daraus gebildete $k \times k$-Matrix regulär ist. Falls das charakteristische Polynom nur einfache Nullstellen $\lambda_1, \ldots, \lambda_k$ hat, ist

$$\begin{pmatrix} T_1(0) & \cdots & T_k(0) \\ T_1(1) & \cdots & T_k(1) \\ \vdots & & \vdots \\ T_1(k-1) & \cdots & T_k(k-1) \end{pmatrix} = \begin{pmatrix} 1 & \cdot & 1 \\ \lambda_1 & \cdots & \lambda_k \\ \lambda_1^2 & \cdots & \lambda_k^2 \\ \vdots & & \vdots \\ \lambda_1^{k-1} & \cdots & \lambda_k^{k-1} \end{pmatrix}$$

eine Vandermondsche Matrix, deren Determinante gleich $\prod_{j<m}(\lambda_j - \lambda_m) \neq 0$ ist.

B.3 Beispiele

B.3.1 $T(n) - 3T(n-1) - 4T(n-2) = 0$

Koeffizienten $a_0 = 1, a_1 = -3, a_2 = -4$

charakt. Gleichung $x^2 - 3x - 4 = 0$

Nullstellen $\lambda_1 = -1, \lambda_2 = 4$

Lösungsraum $\{\alpha_1(-1)^n + \alpha_2 4^n \colon \alpha_1, \alpha_2 \in \mathbb{C} \text{ (oder } \mathbb{R})\}$

Die Lösung mit den Anfangswerten $T(0) = 0$ und $T(1) = 1$ erhält man, indem man $T(n) = \alpha_1(-1)^n + \alpha_2 4^n$ in die Anfangsbedingungen einsetzt

$$0 = T(0) = \alpha_1(-1)^0 + \alpha_2 4^0 = \alpha_1 + \alpha_2$$
$$1 = T(1) = \alpha_1(-1) + \alpha_2 4 = -\alpha_1 + 4\alpha_2$$

und das entstehende Gleichungssystem nach α_1 und α_2 auflöst. Dies ergibt hier $-\alpha_1 = \alpha_2$ und somit $\alpha_1 = -\frac{1}{5}$ und $\alpha_2 = \frac{1}{5}$. Die gesuchte Lösung lautet also

$$T(n) = -\frac{1}{5}(-1)^n + \frac{1}{5}4^n.$$

B.3.2 $T(n) = T(n-1) + T(n-2)$

oder $T(n) - T(n-1) - T(n-2) = 0$

Koeffizienten	$a_0 = 1,\ a_1 = -1,\ a_2 = -1$
charakt. Gleichung	$x^2 - x - 1 = 0$
Wurzeln	$\lambda_1 = \frac{1}{2}(1+\sqrt{5}),\ \lambda_2 = \frac{1}{2}(1-\sqrt{5})$
Lösungsraum	$\left\{\alpha_1\left(\frac{1+\sqrt{5}}{2}\right)^n + \alpha_2\left(\frac{1-\sqrt{5}}{2}\right)^n : \alpha_1, \alpha_2 \in \mathbb{C} \text{ (oder } \mathbb{R})\right\}$

Bestimmung der Lösung mit den Anfangswerten $T(0) = 0,\ T(1) = 1$:
Einsetzen der allgemeinen Lösung in die Anfangsbedingungen ergibt

$$0 = \alpha_1 + \alpha_2$$

und

$$1 = \alpha_1 \frac{1+\sqrt{5}}{2} + \alpha_2 \frac{1-\sqrt{5}}{2}$$

also $\alpha_1 = -\alpha_2$ und somit $1 = \alpha_2\left(-\frac{1+\sqrt{5}}{2} + \frac{1-\sqrt{5}}{2}\right) = -\alpha_2\sqrt{5}$, also $\alpha_2 = -\frac{1}{\sqrt{5}}$.
Die gesuchte Lösung lautet folglich

$$T(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Bemerkung: Die obige Rekursionsgleichung mit den Anfangswerten $T(0) = 0,\ T(1) = 1$ definiert die Folge der *Fibonacci*-Zahlen; $T(n)$ ist die n-te Fibonacci-Zahl. Wir haben also eine explizite Darstellung der Fibonacci-Zahlen erhalten. Die Fibonacci-Zahlen spielen in etlichen Problemen eine Rolle, weil die sie definierende Rekursionsgleichung dort vorkommt.

B.3.3 $\quad T(n) = 2T(n-1) - 2T(n-2)$

oder $\quad T(n) - 2T(n-1) + 2T(n-2) = 0$

Koeffizienten	$a_0 = 1,\ a_1 = -2,\ a_2 = 2$
charakt. Gleichung	$x^2 - 2x + 2 = 0$
Wurzeln	$\lambda_1 = 1+i,\ \lambda_2 = 1-i$
Lösungsraum	$\{\alpha_1(1+i)^n + \alpha_2(1-i)^n : \alpha_1, \alpha_2 \in \mathbb{C}\}$

Reellwertige Lösungen: Es gilt

$$1 + i = \sqrt{2}e^{i\frac{\pi}{4}} \text{ und } 1 + i = \sqrt{2}e^{-i\frac{\pi}{4}}$$

also

$$(1+i)^n = (\sqrt{2})^n e^{in\frac{\pi}{4}} \text{ und } (1-i)^n = (\sqrt{2})^n e^{-in\frac{\pi}{4}}$$

und

$$(1+i)^n + (1-i)^n = (\sqrt{2})^n 2\cos n\tfrac{\pi}{4} \text{ und } (1+i)^n - (1-i)^n = i\,(\sqrt{2})^n 2\sin n\tfrac{\pi}{4}$$

Die Funktionen $n \mapsto (\sqrt{2})^n 2\cos n\frac{\pi}{4}$ und $n \mapsto (\sqrt{2})^n 2\sin n\frac{\pi}{4}$ sind ebenfalls eine Basis.

Bestimmung der Lösung mit den Anfangswerten $T(0) = 0,\ T(1) = 1$:
Einsetzen der allgemeinen Lösung in die Anfangsbedingungen ergibt

$$\begin{aligned} 0 &= \alpha_1 + \alpha_2 \\ 1 &= \alpha_1(1+i) + \alpha_2(1-i) \end{aligned}$$

also $\alpha_1 = -\alpha_2$ und $1 = -\alpha_2(1+i)+\alpha_2(1-i) = \alpha_2(-2i)$ und somit $\alpha_2 = -\frac{1}{2i},\ \alpha_1 = \frac{1}{2i}$.

Die gesuchte Lösung lautet damit

$$T(n) = \frac{1}{2i}\big((1+i)^n - (1-i)^n\big) = \left(\sqrt{2}\right)^n \sin n\tfrac{\pi}{4} = 2^{\frac{n}{2}} \sin n\tfrac{\pi}{4}$$

B.3.4 $\quad T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3)$

oder $\quad T(n) - 5T(n-1) + 8T(n-2) - 4T(n-3) = 0$

Koeffizienten $\quad a_0 = 1,\ a_1 = -5,\ a_2 = 8,\ a_3 = -4$

charakt. Gleichung $\quad 0 = x^3 - 5x^2 + 8x - 4 = (x-1)(x-2)^2$

Wurzeln $\quad \lambda_1 = 1$ einfach, $\lambda_2 = 2$ zweifach

Basis des Lösungsraums $\quad 1^n,\ 2^n,\ n2^n$

Allgemeine Lösung $\quad \alpha_1 + \alpha_2 2^n + \alpha_3 n 2^n$

Bestimmung der Lösung mit den Anfangswerten $T(0) = 0,\ T(1) = 1,\ T(2) = 2$:
Einsetzen der allgemeinen Lösung in die Anfangsbedingungen ergibt

$$\begin{aligned} 0 &= \alpha_1 + \alpha_2 \\ 1 &= \alpha_1 + 2\alpha_2 + 2\alpha_3 \\ 2 &= \alpha_1 + 4\alpha_2 + 8\alpha_3 \end{aligned}$$

Dieses lineare Gleichungssystem hat die Lösung $\alpha_1 = -2,\ \alpha_2 = 2,\ \alpha_3 = -\frac{1}{2}$. Somit lautet die gesuchte Lösung

$$T(n) = -2 + 2^{n+1} - n2^{n-1}$$

B.4 Die Lösung inhomogener linearer Rekursionsgleichungen

Gegeben sei die inhomogene lineare Rekursionsgleichung der Ordnung k

$$\sum_{j=0}^{k} a_j T(n-j) \; = \; f(n) \tag{B.7}$$

mit konstanten reellen Koeffizienten $a_0, \ldots a_k$, $a_0 \neq 0$, $a_k \neq 0$.

B.4.1 Satz

Ist T_0 eine Lösung von B.7 und ist $\mathcal{L}$ der Lösungsraum der zugehörigen homogenen Gleichung, so ist $T_0 + \mathcal{L} = \{T_0 + T \colon T \in \mathcal{L}\}$ die Menge der Lösungen von B.7. T_0 wird auch eine partikuläre Lösung von B.1 genannt.

Beweis: Sei T_1 eine Lösung von B.7. Dann gilt $\sum_{j=0}^{k} a_j \big(T_1(n-j) - T_0(n-j)\big) = 0$. Also ist $T_1 - T_0$ eine Lösung der zugehörigen homogenen Gleichung, d.h. $T_1 = T_0 + T_1 - T_0 \in T_0 + \mathcal{L}$.
Umgekehrt gilt für jede Funktion $T_0 + T$ mit $T \in \mathcal{L}$, daß $\sum_{j=0}^{k} a_j \big(T_0(n-j) - T(n-j)\big) = \sum_{j=0}^{k} a_j T_0(n-j) - \sum_{j=0}^{k} a_j T(n-j) = f(n) + 0$; also ist $T_0 + T$ eine Lösung von B.7.

B.4.2 Bemerkungen

1. Man beachte die Analogie zur Lösungstheorie linearer Gleichungssysteme und linearer gewöhnlicher Differentialgleichungen. Wenn man die Lösungen der homogenen Gleichungen kennt, muß man nur eine einzige Lösung der inhomogenen Gleichung finden, um alle Lösungen zu kennen. Leider gibt es kein allgemeines Verfahren zur Bestimmung einer partikulären Lösung. Aber immerhin gibt es für spezielle rechte Seiten f solche Verfahren.

2. Um vorgegebene Anfangsbedingungen zu erfüllen, muß man zu einer partikulären Lösung einfach eine geeignete Lösung der homogenen Gleichung addieren.

B.4.3 Rezept zur Lösung von Rekursionsgleichungen der Gestalt

$$\sum_{j=0}^{k} a_j T(n-j) = b^n p(n), \; n \geq k,$$

wobei $a_0 \neq 0$, $a_k \neq 0$, $b > 0$ und $p(x) = \sum_{r=0}^{D} d_r x^r$ ein Polynom vom Grad D seien.

Ist b eine Nullstelle des charakteristischen Polynoms $\sum_{j=0}^{k} a_j x^{k-j}$, so sei m ihre Vielfachheit; ansonsten sei $m = 0$.

Setze $T(n) = \sum_{l=m}^{D+m} c_l n^l b^n$ *in die linke Seite der Rekursionsgleichung ein und bestimme* $c_m, \ldots, c_{D+m} \in \mathbb{R}$ *so, daß die linke und die rechte Seite für alle* $n \geq k$ *gleich sind.*

B.4.4 Folgerung

Sind $b_1, \ldots, b_r$ positive reelle Zahlen, $a_0 \neq 0, a_k \neq 0$ reelle Zahlen und $p_1, \ldots, p_r$ Polynome, so kann man eine Lösung von

$$\sum_{j=0}^{k} a_j T(n-j) = \sum_{\rho=1}^{r} b_\rho^n p_\rho(n) \text{ für } n \geq k$$

dadurch bekommen, daß man für jedes $\rho \in \{1, \ldots, r\}$ wie in B.4.3 eine Lösung T_ρ von

$$\sum_{j=0}^{k} a_j T(n-j) = b_\rho^n p_\rho(n) \text{ für } n \geq k$$

bestimmt und dann $T = T_1 + \ldots + T_r$ setzt.

Wir wollen B.4.3 nur in drei Spezialfällen beweisen.

B.4.4.1 Erster Spezialfall: Die Rekursionsgleichung hat die Gestalt

$$\sum_{j=0}^{k} a_j T(n-j) = b^n \text{ für } n \geq k$$

und b ist keine Nullstelle des charakterischen Polynoms der homogenen Gleichung. Dann gibt es eine Lösung der Gestalt $T(n) = c_0 b^n$ mit geeignetem $c_0 \in \mathbb{R}$.

Beweis:

$$\sum_{j=0}^{k} a_j T(n-j) = \sum_{j=0}^{k} a_j c_0 b^{n-j} = b^n c_0 \sum_{j=0}^{k} a_j b^{-j}$$

Weil b keine Nullstelle von $\sum_{j=0}^{k} a_j x^{k-j}$ ist, gilt $\sum_{j=0}^{k} a_j b^{-j} \neq 0$. Setzt man $c_0 = (\sum_{j=0}^{k} a_j b^{-j})^{-1}$, so ergibt sich die rechte Seite. q.e.d.

B.4.4.2 Zweiter Spezialfall: Die Rekursionsgleichung hat die Gestalt

$$\sum_{j=0}^{k} a_j T(n-j) = n b^n \text{ für } n \geq k$$

und b ist keine Nullstelle des charakterischen Polynoms der homogenen Gleichung. Dann gibt es eine Lösung der Gestalt $T(n) = c_0 b^n + c_1 n b^n$ mit geeigneten $c_0, c_1 \in \mathbb{R}$.

Beweis:

$$\begin{aligned}
\text{Linke Seite} &= \sum_{j=0}^{k} a_j T(n-j) \\
&= \sum_{j=0}^{k} a_j \big(c_0 b^{n-j} + c_1 (n-j) b^{n-j}\big) \\
&= b^n \big(c_0 \sum_{j=0}^{k} a_j b^{-j} - c_1 \sum_{j=0}^{k} a_j b^{-j} j + c_1 n \sum_{j=0}^{k} a_j b^{-j}\big)
\end{aligned}$$

Nach Voraussetzung ist $\sum_{j=0}^{k} a_j b^{-j} \neq 0$. Damit die linke Seite für alle $n \geq k$ gleich der rechten ist, muß gelten

$$c_1 \sum_{j=0}^{k} a_j b^{-j} = 1 \text{ und } c_0 \sum_{j=0}^{k} a_j b^{-j} - c_1 \sum_{j=0}^{k} a_j b^{-j} j = 0$$

Dieses lineare Gleichungssystem hat die Lösung

$$c_1 = \frac{1}{\sum_{j=0}^{k} a_j b^{-j}} \text{ und } c_0 = \frac{\sum_{j=0}^{k} a_j b^{-j} j}{\left(\sum_{j=0}^{k} a_j b^{-j}\right)^2}$$

q.e.d.

B.4.4.3 Dritter Spezialfall: Die Rekursionsgleichung hat die Gestalt

$$\sum_{j=0}^{k} a_j T(n-j) = b^n \text{ für } n \geq k$$

und b ist eine einfache Nullstelle des charakterischen Polynoms der homogenen Gleichung.
Dann gibt es eine Lösung der Gestalt $T(n) = c_1 n b^n$ mit geeigneten $c_1 \in \mathbb{R}$.

Beweis:

$$\begin{aligned}
\text{Linke Seite} &= \sum_{j=0}^{k} a_j T(n-j) \\
&= \sum_{j=0}^{k} a_j c_1 (n-j) b^{n-j} \\
&= b^n \big(c_1 n \sum_{j=0}^{k} a_j b^{-j} - c_1 \sum_{j=0}^{k} j a_j b^{-j}\big) \\
&= -b^n c_1 \sum_{j=0}^{k} j a_j b^{-j}
\end{aligned}$$

Damit die linke Seite für alle n gleich der rechten Seite ist, muß $1 = -c_1 \sum_{j=0}^{k} j a_j b^{-j}$ gelten, also $c_1 = -\big(\sum_{j=0}^{k} j a_j b^{-j}\big)^{-1}$. Beachte, daß $\sum_{j=0}^{k} j a_j b^{-j} \neq 0$, denn

$$
\begin{aligned}
-\sum_{j=0}^{k} j a_j b^{-j} &= b^{-k+1} b^{k-1} \big(\underbrace{\sum_{j=0}^{k} k a_j b^{-j}}_{=0} - \sum_{j=0}^{k} j a_j b^{-j} \big) \\
&= b^{-k+1} \sum_{j=0}^{k} (k-j) a_j b^{k-j-1} \\
&= b^{-k+1} \frac{d}{dx} \big(\sum_{j=0}^{k} a_j x^{k-j} \big) |_{x=b} \\
&= \neq 0
\end{aligned}
$$

denn b ist nur eine einfache Nullstelle. q.e.d.

B.5 Beispiele

B.5.1 $T(n) - 2\,T(n-1) = 3^n$

Es ist also $p = 1$ und $b = 3$. Weil b keine Nullstelle des charakteristischen Polynoms $x - 2 = 0$ ist, macht man nach B.4.3 den Lösungsansatz $T(n) = c_0 3^n$. Um c_0 zu bestimmen, setzt man T in die Rekursionsgleichung ein und sieht, daß

$$c_0 3^n - 2 c_0 3^{n-1} = 3^n \quad \text{für alle } n \geq 1$$

genau dann gilt, wenn $c_0 = \frac{3^n}{3^n - 2 \cdot 3^{n-1}} = \frac{3^n}{3^{n-1}} = 3$ ist. Also ist $T(n) = 3^{n+1}$ eine Lösung der inhomogenen Gleichung.

Eine Lösung mit der Anfangsbedingung $T(0) = 0$ erhält man durch Addition einer geeigneten Lösung der zugehörigen homogenen

Rekursionsgleichung	$T(n) - 2\,T(n-1) = 0$
Koeffizienten	$a_0 = 1$, $a_1 = -2$
charakt. Gleichung	$x - 2 = 0$
Wurzeln	$\lambda_1 = 2$
Allgemeine Lösung	$\alpha 2^n$

Die Funktion $3^{n+1} + \alpha \cdot 2^n$ erfüllt die Anfangsbedingung genau dann, wenn $3 + \alpha = 0$, also $a = -3$. Somit lautet die gesuchte Lösung der inhomogenen Rekursionsgleichung mit der Anfangsbedingung $T(0) = 0$

$$T(n) = 3^{n+1} - 3 \cdot 2^n$$

B.5.2 $T(n) - 2\,T(n-1) = (n+5)3^n$

Es ist $a_0 = 1$, $a_1 = -2$, $k = 1$, $p(x) = x + 5$ und $b = 3$ ist keine Wurzel des charakteristischen Polynoms $x - 2$. Somit lautet nach B.4.3 der entsprechende

Lösungsansatz $c_0 3^n + c_1 n 3^n$

Einsetzen in die Rekursionsgleichung ergibt

$$\begin{aligned} c_0 3^n + c_1 n 3^n - 2c_0 3^{n-1} - 2c_1(n-1)3^{n-1} &= (n+5)3^n \\ n(3^n c_1 - 2c_1 3^{n-1}) + c_0 3^n - 2c_0 3^{n-1} + 2c_1 3^{n-1} &= 3^n n + 5 \cdot 3^n \end{aligned}$$

Damit diese Gleichungen für alle $n \geq 1$ erfüllt sind, müssen die Koeffizienten entsprechender Potenzen von n auf beiden Seiten übereinstimmen. Man macht also einen Koeffizientenvergleich.

Koeffizienten von n^1: $c_1 - \frac{2}{3}c_1 = 1$
Koeffizienten von n^0: $c_0 - \frac{2}{3}c_0 + \frac{2}{3}c_1 = 5$

Dieses lineare Gleichungssystem hat die Lösung $c_0 = 9$, $c_1 = 3$. Infolgedessen hat die obige inhomogene Rekursionsgleichung folgende partikuläre Lösung

$$T(n) = 9 \cdot 3^n + 3n3^n = 3^{n+2} + n3^{n+1}$$

B.5.3 Türme von Hanoi $T(n) = 2\,T(n-1) + 1$

Lösungsansatz $T(n) = c_0 \cdot 1^n = c_0$

Einsetzen in die Rekursionsgleichung ergibt $c_0 = 2c_0 + 1$, also $c_0 = -1$.

Somit ist $T_0(n) = -1$ eine Lösung. Die allgemeine Lösung der zugehörigen homogenen Gleichung hat die Gestalt $T(n) = \alpha\, 2^n$. Folglich hat die allgemeine Lösung der inhomogenen Gleichung die Form $T(n) = \alpha\, 2^n - 1$. Um die Anfangsbedingung $T(0) = 0$ zu erfüllen, muß $\alpha = 1$ sein, also $T(n) = 2^n - 1$.

B.5.4 $T(n) - T(n-1) = 1$

Es ist $a_0 = 1$, $a_1 = -1$, $k = 1$, $p(x) = 1$ und $b = 1$ ist eine Wurzel des charakteristischen Polynoms $x - 1$ der Vielfachheit 1. Somit lautet nach B.4.3 der entsprechende

Lösungsansatz $T(n) = c_1 n 1^n = c_1 n$

Einsetzen in die Rekursionsgleichung ergibt $c_1 n - c_1(n-1) = 1$. Dies ist für alle $n \geq 1$ genau dann erfüllt, wenn $c_1 = 1$ ist. Folglich ist $T(n) = n$ eine Lösung. (Das hätte man in diesem Fall auch erraten können.)

B.5.5 $T(n) - 2\,T(n-1) = 2^n - 1,\ n \geq 1$

Zerlege die rechte Seite in zwei Summanden und löse die folgenden beiden Gleichungen

$$T(n) = 2\,T(n-1) + 2^n \tag{B.8}$$
$$T(n) = 2\,T(n-1) - 1 \tag{B.9}$$

Die charakteristische Gleichung der homogenen Anteile ist $x - 2 = 0$, hat also die Wurzel 2.

Nach B.4.3 hat daher B.8 eine Lösung der Gestalt $T_1(n) = c_1 n\, 2^n$. Einsetzen in B.8 liefert

$$c_1 n\, 2^n - 2\, c_1(n-1)\, 2^{n-1} - 2^n = 0$$

und diese Gleichung gilt genau dann für alle n, wenn $c_1 = 1$ ist.

Nach B.4.3 hat B.9 eine Lösung der Gestalt $T_2(n) = c_2\, 1^n = c_2$. Einsetzen in B.9 liefert

$$c_2 - 2\, c_2 + 1 = 0$$

und diese Gleichung hat die Lösung $c_2 = 1$.

Zusammengesetzt erhält man somit $T_1(n) + T_2(n) = n\, 2^n + 1$ als Lösung der ursprünglichen Rekursionsgleichung. Alle Lösungen bekommt man, indem man die Lösungen der homogenen Gleichung $T(n) = 2\, T(n-1)$ dazu addiert; sie haben also die Gestalt $T(n) = n\, 2^n + 1 + \alpha\, 2^n$ mit $\alpha \in \mathbb{R}$.

Anhang C

Divide-et-Impera-Rekursionsgleichungen

Divide-et-Impera-Algorithmen sind meist für alle Problemgrößen n definiert, somit ist auch ihre Rechenzeit T auf ganz $\mathbb{N}$ definiert. T erfüllt eine mehr oder weniger komplizierte Rekursionsgleichung, die sich aber meist vereinfacht, wenn man nur spezielle Problemgrößen $n \in D := \{b^k n_0 \colon k \in \mathbb{N}\}$ betrachtet, wobei $\frac{n}{b}$ die Größe der Teilprobleme und n_0 die Abbruchschwelle der Rekursion sind. Typischerweise erhält man für solche n eine Rekursionsgleichung der Gestalt

$$T(n) = a\,T(\tfrac{n}{b}) + f(n),\ n \in D,\ T(n_0) = \alpha \tag{C.1}$$

Dabei sind $a \in \mathbb{R}$, $a > 0$, $b \in \mathbb{N}$, $b \geq 2$, $n_0 \in \mathbb{N}$ und $D = \{b^k n_0 \colon k \in \mathbb{N}\}$, $f \colon D \to \mathbb{R}$ und $\alpha \in \mathbb{R}$.

Solche auf D eingeschränkte Gleichungen sind oft relativ einfach zu lösen, weil man sie auf lineare Rekursionsgleichungen zurückführen kann; zumindest kann man asymptotische Wachstumsabschätzungen bestimmen.

Schwieriger ist es, Lösungen zu finden, die solche Gleichungen auf ganz $\mathbb{N}$ erfüllen, oder wenigstens geeignete asymptotische Wachstumsabschätzungen für sie auf $\mathbb{N}$ zu beweisen.

C.1 Lösungen auf $D := \{b^k n_0 \colon k \in \mathbb{N}\}$

Man kann C.1 auf eine lineare Rekursionsgleichung zurückführen, indem man $n = b^k n_0$ substituiert und $S(k) = T(b^k n_0)$ setzt; dann wird aus C.1 die folgende lineare Rekursionsgleichung

$$S(k) = a\,S(k-1) + f(b^k n_0),\ k > 0,\ S(0) = \alpha$$

Diese Gleichung können wir mit den Ergebnissen des vorigen Kapitels für einige Funktionen f lösen.

C.1.1 Lemma
Es seien $c \in \mathbb{R}$ und $l \in \mathbb{N}$. Dann ist eine Funktion $T\colon D \cup \{n_0\} \to \mathbb{R}$ genau dann eine Lösung der Rekursionsgleichung

$$T(n) = a\,T(\tfrac{n}{b}) + c\,n^l \text{ für } n \in D \tag{C.2}$$

wenn es ein $\alpha \in \mathbb{R}$ gibt, so daß für alle $n \in D$ gilt

- im Falle $a \neq b^l$ $\qquad T(n) = \dfrac{c}{1-\frac{a}{b^l}}\,n^l + \alpha\,n^{\log_b a}$
- im Falle $a = b^l$ $\qquad T(n) = \alpha\,n^l + c\,n^l \log_b n$

Beweis: Transformiere C.2 durch die Substitution $n = b^k n_0,\ S(k) = T(b^k n_0)$ in die lineare Rekursionsgleichung

$$S(k) = a\,S(k-1) + c\,(b^k n_0)^l \tag{C.3}$$

Die charakteristische Gleichung ihres homogenen Anteils ist $x-a$, hat also die Wurzel a. Die Inhomogenität $c(b^k n_0)^l = c\,n_0^l (b^l)^k$ hat die Gestalt $B^k p(k)$ mit $B = b^l$ und dem konstanten Polynom $p(x) = c\,n_0^l$ wie in B.4.3.

1. Fall: $a \neq b^l$**.** B ist keine Wurzel des charakteristischen Polynoms. Nach B.4.3 gibt es daher eine Lösung von C.3 der Gestalt $S_0(k) = c_1 B^k$ mit einem $c_1 \in \mathbb{R}$. Durch Einsetzen in C.3 erhält man für c_1 die Gleichung

$$c_1 B^k = a\,c_1 B^{k-1} + c\,n_0^l B^k \text{ für jedes } k \in \mathbb{N}$$

Sie ist genau dann erfüllt, wenn

$$c_1 = \frac{c\,n_0^l}{1-\frac{a}{B}}$$

Folglich ist

$$\begin{aligned} S_0(k) &= c_1 B^k = c_1 b^{kl} \\ &= \frac{c}{1-\frac{a}{b^l}}(n_0 b^k)^l \\ &= \frac{c}{1-\frac{a}{b^l}} n^l \end{aligned}$$

Nach B.4.1 hat jede Lösung von C.3 die Gestalt $S(k) = S_0(k) + \beta\,a^k$ mit beliebigem $\beta \in \mathbb{R}$. Wegen $a^k = b^{k \log_b a} = n_0^{-\log_b a}(n_0 b^k)^{\log_b a} = n_0^{-\log_b a} n^{\log_b a}$ ergibt sich durch Rücktransformation, daß jede Lösung von C.2 die Gestalt

$$T(n) = \frac{c}{1-\frac{a}{b^l}} n^l + \beta\,n_0^{-\log_b a} n^{\log_b a}$$

hat. Und $\alpha := \beta\,n_0^{-\log_b a}$ durchläuft ganz $\mathbb{R}$, wenn β ganz $\mathbb{R}$ durchläuft.

2. Fall: $a = b^l$**.** Dann ist $B = b^l = a$ eine Wurzel des charakteristischen Polynoms $x-a$. Nach B.4.3 gibt es daher eine Lösung von C.3 der Gestalt $S_0(k) = c_2 k\,a^k$. Durch Einsetzen in C.3 erhält man für c_2 die Gleichungen

$$c_2 k\,a^k = a\,c_2(k-1)\,a^{k-1} + c\,n_0^l a^k \text{ für jedes } k \in \mathbb{N}$$

die genau dann erfüllt sind, wenn $c_2 = c\,n_0^l$ ist. Also ist $S_0(k) = c\,n_0^l k\,a^k = c\,n_0^l(\log_b \frac{n}{n_0})\,b^{lk} = c\,n_0^l b^{lk}\log_b n - c\,n_0^l b^{lk}\log_b n_0 = c\,n^l \log_b n - c\,n^l \log_b n_0$.

Nach B.4.1 hat jede Lösung von C.3 die Gestalt $S(k) = S_0(k) + \beta\,a^k$ mit beliebigem $\beta \in \mathbb{R}$. Wegen $a^k = b^{lk} = n_0^{-l} n^l$ ergibt sich durch Rücktransformation, daß jede Lösung von C.2 folgende Gestalt hat

$$\begin{aligned} T(n) &= c\,n^l \log_b n - c\,(\log_b n_0)\,n^l + \beta\,n_0^{-l} n^l \\ &= c\,n^l \log_b n + \big(\frac{\beta}{n_0^l} - c\,\log_b n_0\big)\,n^l \end{aligned}$$

Und $\alpha := \beta/n_0^l - c\,\log_b n_0$ durchläuft ganz $\mathbb{R}$, wenn β ganz $\mathbb{R}$ durchläuft.

C.1.2 Korollar

Seien $a, c \in \mathbb{R}$ positiv, $b \in \mathbb{N}$, $b \geq 2$, $n_0 \in \mathbb{N}$, $l \in \mathbb{N} \cup \{0\}$. Dann gilt für jede Lösung T der Rekursionsgleichung

$$T(n) = a\,T(\frac{n}{b}) + c\,n^l \quad \text{für } n \in D = \{b^k n_0 \colon k \in \mathbb{N}\}$$

a) $T \in \Theta(n^l)$, falls $a < b^l$

b) $T \in \Theta(n^l \log_b n)$, falls $a = b^l$

c) $T \in \Theta(n^{\log_b a})$, falls $a > b^l$ und T asymptotisch nichtnegativ ist.

Beweis:

a) Nach C.1.1 gilt

$$T(n) = \frac{c}{1 - \frac{a}{b^l}}\,n^l + \alpha\,n^{\log_b a}$$

mit $\alpha \in \mathbb{R}$. Wegen $a < b^l$ ist $1 - \frac{a}{b^l} > 0$ und $\log_b a < l$. Nach A.2.9.2. und A.2.6 h) folgt $T \in \Theta(n^l) + \Theta(n^{\log_b a}) = \Theta(n^l)$.

b) Nach C.1.1 gilt $T(n) = \alpha\,n^l + c\,n^l \log_b n$ mit $\alpha \in \mathbb{R}$. Wegen $c > 0$ folgt $T \in \Theta(n^l \log_b n)$.

c) Nach C.1.1 gilt

$$T(n) = \frac{c}{1 - \frac{a}{b^l}} n^l + \alpha\,n^{\log_b a}$$

mit $\alpha \in \mathbb{R}$. Wegen $a > b^l$ ist $\frac{c}{1-\frac{a}{b^l}} < 0$ und $\log_b a > l$ und $\lim_{n\to\infty} \frac{n^{\log_b a}}{n^l} = \infty$. Wäre $\alpha \leq 0$, so wäre T nicht asymptotisch nichtnegativ. Also $\alpha > 0$ und wegen $n^l \in O(n^{\log_b a})$ folgt $T \in \Theta(n^{\log_b a})$. q.e.d.

C.1.3 Bemerkung Die Aussage c) wird falsch, wenn T nicht asymptotisch nichtnegativ ist; denn z.B. $T(n) = 4\,T(\frac{n}{2}) + n$ hat die Lösung $T(n) = -n$.

Achtung! Bisher haben wir diese Wachstumsabschätzungen nur auf $D = \{b^k n_0 \colon k \in \mathbb{N}\}$ bewiesen!

C.2 Lösungen auf ganz $\mathbb{N}$

Divide-et-Impera-Algorithmen sind meist für alle Problemgrößen n definiert; somit ist auch ihre Rechenzeit T auf ganz $\mathbb{N}$ definiert. T erfüllt eine mehr oder weniger komplizierte Rekursionsgleichung, die sich aber meist vereinfacht, wenn man nur spezielle Problemgrößen $n \in D = \{b^k n_0 \colon k \in \mathbb{N}\}$ betrachtet, wobei $\frac{n}{b}$ die Größe der Teilprobleme und n_0 die Abbruchschwelle der Rekursion sind. Typischerweise erhält man für solche n eine Rekursionsgleichung der Gestalt

$$T(n) = a\,T(\tfrac{n}{b}) + f(n),\ n \in D,\ T(n_0) = \alpha$$

Für die Lösungen solcher Rekursionsgleichungen haben wir bereits einige Aussagen bewiesen. Die gelten aber nur auf D!

Frage: Unter welchen Bedingungen kann man von den Wachstumseigenschaften von $T|D$ auf die von T (auf ganz $\mathbb{N}$ definiert) schließen? Wenn zum Beispiel $T(n) = a\,T(\frac{n}{b}) + f(n)$ für alle $n \geq n_0$ gilt (wobei $\frac{n}{b}$ als $\lfloor \frac{n}{b} \rfloor$ oder als $\lceil \frac{n}{b} \rceil$ interpretiert wird), kann man dann $T \in O(g)$ schließen, wenn $T|D \in O(g|D)$ gilt?

C.2.1 Definition und Lemma

a) D sei eine unendliche Teilmenge von $\mathbb{N}$. Eine Funktion $f \colon D \to \mathbb{R}$ heißt **asymptotisch nichtfallend**, wenn es ein $n_0 \in D$ gibt, so daß für alle $n, m \in D$ gilt: Ist $n_0 \leq n < m$, so gilt $f(n) \leq f(m)$.

b) Für $b \in \mathbb{N}$ sei $\mu_b \colon \mathbb{N} \to \mathbb{N},\ n \mapsto b\,n$ die Multiplikation mit b. Eine Funktion $f \colon \mathbb{N} \to \mathbb{R}$ heißt **glatt**, wenn sie asymtotisch nichtfallend und asymptotisch nichtnegativ ist und zusätzlich eine der beiden folgenden äquivalenten Eigenschaften (und somit beide) besitzt:

- **(i)** Es gibt ein $b \in \mathbb{N}$, $b > 1$, so daß $f \circ \mu_b \in O(f)$ d.h. es gibt $c > 0$, $n_0 \in \mathbb{N}$ mit $f(b\,n) \leq c\,f(n)$ für $n \geq n_0$. (Man nennt f dann b-glatt.)
- **(ii)** Für jedes $b \in \mathbb{N}$, $b > 1$, ist $f \circ \mu_b \in O(f)$.

Beweis der Äquivalenz: (ii) $\Rightarrow$ (i) ist trivial.
(i) $\Rightarrow$ (ii): Da f asymptotisch nichtfallend ist, gibte es ein $n_1 \in \mathbb{N}$, so daß für alle $n \geq n_1$ und $m \geq n_1$ gilt: $n \leq m \Rightarrow f(n) \leq f(m)$. Sei n_0 wie in (i) gewählt. Setze $n_2 = \max\{n_0, n_1\}$. Wir zeigen zunächst $f \circ \mu_2 \in O(f)$. Weil $2 \leq b$ und f asymptotisch nicht fällt, gilt $f(2n) \leq f(b\,n) \leq c\,f(n)$ für $n \geq n_2$, also $f \circ \mu_2 \in O(f)$. Insbesondere $f(4n) \leq c\,f(2n) \leq c^2\,f(n)$. Durch vollständige Induktion folgt $f(2^k n) \leq c^k\,f(n)$ für alle $k \in \mathbb{N}$ und $n \geq n_2$. Das bedeutet $f \circ \mu_{2^k} \in O(f)$ für jedes $k \in \mathbb{N}$. Ist nun $d \in \mathbb{N}$, $d > 1$, beliebig, so kann man ein $k \in \mathbb{N}$ finden mit $d \leq 2^k$. Dann gilt $f(d\,n) \leq f(2^k\,n) \leq c^k\,f(n)$ für $n \geq n_2$. Also $f \circ \mu_d \in O(f)$. q.e.d.

C.2.2 Lemma Seien $b \in \mathbb{N}$, $b > 1$, $n_0 \in \mathbb{N}$ und $D = \{b^k n_0 \colon k \in \mathbb{N}\}$. Die Funktion $g \colon \mathbb{N} \to \mathbb{R}$ sei glatt, und $T \colon \mathbb{N} \to \mathbb{R}$ sei asymptotisch nichtfallend und asymptotisch nichtnegativ. Dann gilt

a) $T|D \in O(g|D) \implies T \in O(g)$

b) $T|D \in \Omega(g|D) \implies T \in \Omega(g)$

c) $T|D \in \Theta(g|D) \implies T \in \Theta(g)$

Beweis:

a) Weil g glatt ist, gibt es ein $c_1 > 0$ und ein $n_1 \in \mathbb{N}$, so daß $0 \leq g(b\,n) \leq c_1\,g(n)$ für $n \geq n_1$ und $g(n) \leq g(m)$ für $n_1 \leq n \leq m$. Weil $T|D \in O(g|D)$ ist, gibt es ein $c_2 > 0$ und ein $n_2 \in \mathbb{N}$, so daß $T(n) \leq c_2\,g(n)$ für $n \geq n_2$, $n \in D$. Weil T asymptotisch nicht fällt, gibt es ein $n_3 \in \mathbb{N}$, so daß $T(n) \leq T(m)$ für $n_3 \leq n \leq m$. Weil T asymptotisch nichtnegativ ist, gibt es ein $n_4 \in \mathbb{N}$, so daß $T(n) \geq 0$ für $n \geq n_4$. Setze $N = \max\{n_1, n_2, n_3, n_4\}$ und wähle ein $k_0 \in \mathbb{N}$ mit $b^{k_0} n_0\ geqN$. Dann gilt für jedes $k \geq k_0$ und jedes n mit $b^k n_0 \leq n \leq b^{k+1} n_0$

$$T(n) \leq T(b^{k+1}n_0) \overset{T|D\in O(g|D)}{\leq} c_2\,g(b^{k+1}n_0) \overset{g \text{ glatt}}{\leq} c_1 c_2\,g(b^k n_0) \overset{g \text{ asym. nichtfall.}}{\leq} c_1 c_2\,g(n)$$

Das bedeutet gerade $T \in O(g)$.

b) läuft genauso, nur wählt man n_2 so, daß $T(n) \geq c_2\,g(n)$ für $n \geq n_2$, $n \in D$. Dann erhält man für $k > k_0$ und $b^k n_0 \leq n \leq b^{k+1} n_0$

$$T(n) \geq T(b^k n_0) \overset{T|D\in O(g|D)}{\geq} c_2\,g(b^k n_0) \overset{g \text{ glatt}}{\geq} \frac{c_2}{c_1}\,g(b^{k+1}n_0) \geq \frac{c_2}{c_1}\,g(n)$$

Das bedeutet gerade $T \in \Omega(g)$.

c) Folgt aus a) und b), weil $\Theta(g) = O(g) \cap \Omega(g)$. q.e.d.

Bemerkung Die Voraussetzungen, daß g glatt und T asymptotisch nichtfallend ist, sind beide notwendig.

C.2.3 Lemma Seien $a, c \in \mathbb{R}$ positiv, $b \in \mathbb{N}$, $b \geq 2$, $n_0 \in \mathbb{N}$, $l \in \mathbb{N} \cup \{0\}$.
Die Funktion $T\colon \mathbb{N} \to \mathbb{R}$ sei asymptotisch nichtfallend und asymptotisch nichtnegativ. Erfüllt T die Rekursionsgleichung

$$T(n) = a\,T(\tfrac{n}{b}) + c\,n^l \quad \text{für } n \in D = \{b^k\,n_0 \colon k \in \mathbb{N}\},$$

so gilt

$$T \in \begin{cases} \Theta(n^l) & \text{, falls } a < b^l \\ \Theta(n^l \log_b n) & \text{, falls } a = b^l \\ \Theta(n^{\log_b a}) & \text{, falls } a > b^l \end{cases}$$

Beweis: Folgt aus C.1.2 und C.2.2, wenn n^l, $n^l \log_b n$, $n^{\log_b a}$ als glatt nachgewiesen werden. Diese Funktionen sind nichtnegativ und monoton wachsend. Sei $g(n) = n^\alpha$ mit $\alpha > 0$. Dann gilt $g(2\,n) = 2^\alpha n^\alpha = 2^\alpha\,g(n)$, also $g \circ \mu_2 \in O(g)$ d.h. g ist glatt. Sei nun $g(n) = n^l \log_b n$. Dann gilt $g(b\,n) = b^l\,n^l \log_b n + b^l\,n^l \log_b b = b^l\,n^l \log_b n + b^l\,n^l$, folglich $g \circ \mu_b \in O(g)$, d.h. g ist glatt. q.e.d.

C.2.4 Satz Seien $a, c \in \mathbb{R}$ positiv, $b \in \mathbb{N}, b \geq 2$, $n_0 \in \mathbb{N}$, $l \in \mathbb{N} \cup \{0\}$ und $\alpha_0, \ldots, \alpha_{n_0} \in \mathbb{R}$, $0 \leq \alpha_0 \leq \ldots \leq \alpha_{n_0}$.
Erfüllt $T\colon \mathbb{N} \cup \{0\} \to \mathbb{R}$ die Rekursionsgleichung

$$T(n) = a\,T(\tfrac{n}{b}) + c\,n^l \quad \text{für } n > n_0,\ T(n) = \alpha_n \text{ für } n \leq n_0,$$

wobei $\frac{n}{b}$ entweder als $\lfloor \frac{n}{b} \rfloor$ oder als $\lceil \frac{n}{b} \rceil$ zu interpretieren ist, so gilt

$$T \in \begin{cases} \Theta(n^l) & \text{, falls } a < b^l \\ \Theta(n^l \log_b n) & \text{, falls } a = b^l \\ \Theta(n^{\log_b a}) & \text{, falls } a > b^l \end{cases}$$

Beweis: Folgt aus C.2.3; es ist lediglich nachzuweisen, daß T asymptotisch nichtfallend und asymptotisch nichtnegativ ist. Es gilt

$$T(n+1) - T(n) = a\,(T(\tfrac{n+1}{b}) - T(\tfrac{n}{b})) + c\,((n+1)^l - n^l) \geq a\,(T(\tfrac{n+1}{b}) - T(\tfrac{n}{b}))$$

Wir zeigen $T(n+1) - T(n) \geq 0$ durch vollständige Induktion.

Induktionsanfang $n = n_0$: Wegen $b \geq 2$ ist $\frac{n}{b} \leq n_0$ und $\frac{n+1}{b} \leq \frac{n_0}{2} + \frac{1}{2} \leq n_0$. Somit gilt $T(n_0+1) - T(n_0) \geq a\,(\alpha_{\frac{n_0+1}{b}} - \alpha_{\frac{n_0}{b}}) \geq 0$.

Induktionsvoraussetzung: Für alle $m \in \mathbb{N}$ mit $n_0 \leq m \leq n$ gelte $T(m+1) - T(m) \geq 0$.

Induktionsbehauptung: Für alle $m \in \mathbb{N}$ mit $n_0 \leq m \leq n+1$ gilt $T(m+1) - T(m) \geq 0$.

Beweis: Für $m \leq n$ gilt die Behauptung nach Induktionsvoraussetzung. Sei $m = n+1$. Es gilt $|\frac{n+2}{b} - \frac{n+1}{b}| = \frac{1}{b} \leq \frac{1}{2}$, also $\lfloor \frac{n+2}{b} \rfloor - \lfloor \frac{n+1}{b} \rfloor \leq 1$ und auch $\lceil \frac{n+2}{b} \rceil - \lceil \frac{n+1}{b} \rceil \leq 1$. Damit folgt $T(m+1) - T(m) = T(n+2) - T(n+1) \geq a\,(T(\frac{n+2}{b}) - T(\frac{n+1}{b})) \geq 0$ nach Induktionsvoraussetzung.

Es folgt, daß T nicht negativ und nicht fallend ist. q.e.d.

In der Literatur findet man weitere Sätze von ähnlichem Typ, bei denen die Potenz n^l durch andere Funktionen ersetzt ist. Ein Beispiel wollen wir anführen.

C.2.5 Satz Seien $n_0 \in \mathbb{N}$, $0 \leq \alpha_0 \leq \ldots \leq \alpha_{n_0}$, $b \in \mathbb{N}$, $b > 1$, $c \in \mathbb{R}$, $c > 0$. Erfüllt $T\colon \mathbb{N} \cup \{0\} \to \mathbb{R}$ die Rekursionsgleichung

$$T(n) = b\,T(\tfrac{n}{b}) + c\,n \log_b n \text{ für } n > n_0, \quad T(n) = \alpha_n \text{ für } n \leq n_0,$$

wobei $\frac{n}{b}$ durch $\lfloor \frac{n}{b} \rfloor$ oder $\lceil \frac{n}{b} \rceil$ zu ersetzen ist, so gilt $T \in O(n\,(\log_b n)^2)$.

Beweis:

1. Wie im Beweis von C.2.4 folgt, daß T asymptotisch nichtfallend und asymptotisch nichtnegativ ist.

2. Wir zeigen, daß $f(n) = n\,(\log_b n)^2$ b-glatt ist:

f ist asymptotisch nichtfallend und asymptotisch nichtnegativ. Und es gilt $f(b\,n) = b\,n\,(\log_b(b\,n))^2 = b\,n\,(\log_b b + \log_b n)^2 = b\,n\,(1 + 2\log_b n + (\log_b n)^2)) \leq 3\,b\,n\,(\log_b n)^2$ für $n > b^2$.

3. Wegen C.2.2 genügt es zu zeigen, daß $T \in O(n\,(log_b n)^2 | D)$ mit $D = \{b^k : k \in \mathbb{N}\}$. Wähle $d > \max\{\frac{T(b)}{b}, 1\}$. Wir zeigen $T(n) \leq d\,n\,(\log_b n)^2$ für $n \in D$ durch Induktion. Ind.anfang $n = b$: $T(b) \leq d\,b = d\,b\,(\log_b b)^2$.

Ind.schluß $\frac{n}{b} \mapsto n$:

$$\begin{aligned}
T(n) &= b\,T(\frac{n}{b}) + c\,n\,\log_b n \\
&\leq b\,d\,\frac{n}{b}(\log_b \frac{n}{b})^2 + d\,n\,\log_b n \\
&= d\,n\,(\log_b n - 1)^2 + d\,n\,\log_b n \\
&= d\,n\,(\log_b n)^2 - 2\,d\,n\,\log_b n + d\,n + d\,n\,\log_b n \\
&= d\,n\,(\log_b n)^2 + d\,(-n\,\log_b n + n) \\
&\leq d\,n\,(\log_b n)^2
\end{aligned}$$

q.e.d.

Anhang D

Allgemeine Rekursionsgleichungen

Die allgemeine Gestalt einer Rekursionsgleichung ist

$$T(n) = R(T, n) \quad \text{für } n \geq k$$

wobei R ein arithmetischer Ausdruck ist, der nur von n und den Werten $T(m)$ mit $m < n$ abhängt. Es gibt keine einheitliche Lösungstheorie für allgemeine Rekursionsgleichungen. Wie bei Differentialgleichungen oder Integralen gibt es diverse Ansätze und Tricks, deren geschickte Verwendung Erfahrung voraussetzt.

D.1 Erraten einer Lösung

Das geht wohl nur mit viel Erfahrung. Dafür ist die Verifikation oft nicht allzu schwer.

D.2 Iteration

Ersetze die in der rechten Seite $R(T, n)$ vorkommenden Werte $T(m)$ mit $m \geq k$ durch $R(T, m)$. In dem entstehenden Ausdruck ersetze wiederum die auftretenden Werte $T(l)$ mit $l \geq k$ durch $R(T, l)$. Dies wiederholt man, bis nur noch Werte $T(l)$ mit $l < k$ vorkommen. Für jedes konkrete n kann man auf diese Weise den Wert $T(n)$ als Funktion der Anfangswerte $T(0), \ldots, T(k-1)$ bestimmen. Allerdings wird man oft keinen für alle $n \geq k$ gültigen, einfachen, geschlossenen Ausdruck erhalten. Trotzdem kann man manchmal auf diese Weise das asymptotische Verhalten der Lösung abschätzen.

D.2.1 Beispiel $T(n) = T(n-1) + n^3, \quad T(0) = 0$

Es gilt $T(n) = n^3 + T(n-1) = n^3 + (n-1)^3 + T(n-2) = \ldots = n^3 + \ldots + 1^3 = \sum_{j=1}^{n} j^3$. Nach A.2.10 b) gilt $T \in O(n^4)$. Man kann in diesem Fall auch in einer Formelsammlung finden, daß $\sum_{j=1}^{n} j^3 = \frac{1}{4}n^2(n+1)^2$ gilt.

D.3 Substitution

Wie bei Differentialgleichungen oder Integralen kann manchmal eine geschickte Substitution auf einen einfacheren Fall führen, den man lösen kann.

D.3.1 Beispiel: MergeSort

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + n - 1,\ n > 1,\ T(1) = 0$$

Betrachte nur Zweierpotenzen $n = 2^m$ und setze $S(m) = T(2^m)$. Dann erfüllt S die Rekursionsgleichung

$$S(m) = 2\,S(m-1) + 2^m - 1,\ m > 0,\ S(0) = 0$$

Dies ist eine lineare Rekursionsgleichung, die wir gemäß B.4.3 lösen können. Die allgemeine Lösung ist $S(m) = m\,2^m + 1 + \alpha\,2^m$ mit $\alpha \in \mathbb{R}$. Aus der Anfangsbedingung $S(0) = 0$ folgt $\alpha = -1$, also $S(m) = m\,2^m + 1 - 2^m$. Durch Rücktransformation ergint sich $T(n) = n\,\log_2 n + 1 - n$ für alle n, die Zweierpotenzen sind.

D.4 Vereinfachung von Rekursionsgleichungen mit voller Geschichte

Eine Rekursionsgleichung mit voller Geschichte ist eine, deren rechte Seite $R(T, n)$ von allen Werten $T(m)$ mit $m < n$ tatschlich abhängt. In manchen Fällen kann man sie in eine Rekursionsgleichung mit kürzerer Geschichte umformen, die man vielleicht einfacher lösen kann.

D.4.1 Beispiel

$$T(n) = \sum_{j=1}^{n-1} T(j) + a,\ n > 1,\ T(1) = \alpha,\ \text{wobei } a, \alpha \in \mathbb{R}$$

Für $n > 2$ gilt $T(n) - T(n-1) = \sum_{j=1}^{n-1} T(j) + a - (\sum_{j=1}^{n-2} T(j) + a) = T(n-1)$, also $T(n) - 2\,T(n-1) = 0$, und $T(2) = T(1) + a = \alpha + a$. Setzt man $S(m) = T(m+2)$ für $m \geq 0$, so erfüllt S die Rekursionsgleichung

$$S(m) - 2\,S(m-1) = 0 \text{ für } m \geq 1 \text{ und } S(0) = T(2)$$

Nach B.2.4 hat diese Gleichung die allgemeine Lösung $c\,2^m$; aus den Anfangsbedingungen folgt $c = T(1) + \alpha$. Damit ergibt sich $T(n) = S(n-2) = (\alpha + a)\,2^{n-2}$ als Lösung der ursprünglichen Rekursionsgleichung.

Beachte, daß sich durch die Substitution für $n = 2$ eine andere Rekursionsgleichung als für $n > 2$ ergibt. Deswegen setzt man $S(m) = T(m+2)$ und nicht $S(m) = T(m+1)$, und als Anfangsbedingung erhält man $S(0) = T(2) = T(1) + a = \alpha + a$ und nicht $S(0) = T(0) = \alpha$.

D.4.2 Beipiel

$$T(n) = \tfrac{2}{n} \sum_{j=1}^{n-1} T(j) + n + 1 \text{ für } n > 1 \text{ und } T(1) = 0 \tag{D.1}$$

Mit $n-1$ statt n lautet die Gleichung

$$T(n-1) = \tfrac{2}{n-1} \sum_{j=1}^{n-2} T(j) + n \text{ für } n > 2 \tag{D.2}$$

Multiplikation von D.1 mit n und von D.2 mit $n-1$ ergibt

$$\begin{aligned} n\,T(n) &= 2 \sum_{j=1}^{n-1} T(j) + n(n+1) \\ (n-1)\,T(n-1) &= 2 \sum_{j=1}^{n-2} T(j) + n(n-1) \end{aligned}$$

und anschließende Subtraktion der zweiten von der ersten Gleichung liefert

$$n\,T(n) - (n-1)\,T(n-1) = 2\,T(n-1) + 2\,n$$

also

$$T(n) = \tfrac{n+1}{n} T(n-1) + 2 \quad \text{für } n > 2 \tag{D.3}$$

$$T(2) = T(1) + 3 \tag{D.4}$$

Man beachte, daß man wiederum für $n = 2$ eine andere Rekursionsgleichung als für $n > 2$ bekommt, also für D.3 die Anfangsbedingung $T(2) = 3$.

D.3 können wir mit unseren bisher entwickelten Methoden nicht lösen, weil diese sich nur auf Rekursionsgleichungen mit konstanten Koeffizienten bezogen. Wir können aber einfach einmal den Iterationsansatz probieren.

$$\begin{aligned} T(n) &= \frac{n+1}{n} T(n-1) + 2 \\ &= \frac{n+1}{n} \left(\frac{n}{n-1} T(n-2) + 2 \right) + 2 \\ &= \frac{n+1}{n-1} T(n-2) + 2\,\frac{n+1}{n} + 2 \\ &= \frac{n+1}{n-1} \left(\frac{n-1}{n-2} T(n-3) + 2 \right) + 2\,\frac{n+1}{n} + 2 \\ &= \frac{n+1}{n-2} T(n-3) + 2\frac{n+1}{n-1} + 2\frac{n+1}{n} + 2 \end{aligned}$$

Das legt die Vermutung nahe, daß gilt

$$T(n) = \frac{n+1}{n-l+1} T(n-l) + 2(n+1) \sum_{j=n-l+2}^{n+1} \frac{1}{j}$$

l kann vergrößert werden bis $n - l = 2$, also $l = n - 2$. Das liefert für $n > 2$

$$\begin{aligned} T(n) &= \frac{n+1}{3} T(2) + 2(n+1) \sum_{j=4}^{n+1} \frac{1}{j} + 2 \\ &= 2(n+1) \sum_{j=4}^{n+1} \frac{1}{j} + n + 1 \end{aligned}$$

Durch Einsetzen in D.3 verifiziert man, daß T tatsächlich eine Lösung ist:

$$\begin{aligned} \frac{n+1}{n} T(n-1) + 2 &= \frac{n+1}{n} (2n \sum_{j=4}^{n} \frac{1}{j} + n) \\ &= 2(n+1) \sum_{j=4}^{n} \frac{1}{j} + n + 1 + 2\frac{n+1}{n+1} \\ &= 2(n+1) \sum_{j=4}^{n+1} \frac{1}{j} + n + 1 \\ &= T(n) \end{aligned}$$

q.e.d.

D.4.3 Bemerkung Mit A.2.10 a) folgt $\sum_{j=4}^{n} \frac{1}{j} \in O(\log n)$ und somit $T \in O(n \log n)$. Die Rekursionsgleichung D.1 beschreibt die Average-Case-Rechenzeit des Sortieralgorithmus Quicksort.