Kapitel 1

Grundlagen der Algorithmik

Algorithmen und Daten sind zentrale Begriffe der Informatik. Unter **Daten** versteht man die (endlichen) Darstellungen von (abstrakten) Objekten und damit elementare Informationsträger. **Algorithmen** beschreiben effektive Verfahren zur Verarbeitung dieser Daten. Diese Verfahren müssen formalisiert werden (mittels Programmiersprachen) und als Programme auf entsprechenden Maschinen ausgeführt werden.
In diesem einleitenden Kapitel sollen grundlegende Aspekte zusammentragen und einige Fragen und Ansätze der Algorithmik andiskutiert werden. In Gestalt der Turing-Maschine wird zwar eine exakte Formalisierung des Algorithmenbegriffs vorgenommen werden, im Weiteren werden wir dann aber intuitiv vorgehen und einen Algorithmus als Programm einer höheren Programmiersprache verstehen. Ansätze zur Analyse von Algorithmen und einige algorithmische Prinzipien werden vorgestellt. Bei der Angabe konkreter Rechenzeiten ist stets zu beachten, dass diese teilweise auf älteren PCs gemessen wurden. Die Rechenzeiten sind heute wesentlich kürzer, Größenordnungsvergleiche sind aber weiterhin gültig.

1.1 Beispiel: Berechnung der Fibonacci-Zahlen

Wir vergleichen drei Ansätze zur Berechnung der **Fibonacci-Zahlen** f_n mit:

$f_1 = 1$,
$f_2 = 1$,
$f_n = f_{n-1} + f_{n-2}$, für $n \geq 3$.

1.1.1 Rekursive Berechnung

Eine direkte Umsetzung der Definition in ein `C`-Programm zur Berechnung dieser Zahlen wäre

```
int f(int n)
{
  if (n<3) return 1;
  else return f(n-1) + f(n-2);
}
```

Bezeichne $T_{\text{rek}}(n)$ die Zahl der arithmetischen Operationen zur Berechnung von f_n.
Offenbar gelten

$$T_{\text{rek}}(1) = T_{\text{rek}}(2) = 0$$

und die Beziehung

$$T_{\text{rek}}(n) = 3 + T_{\text{rek}}(n-1) + T_{\text{rek}}(n-2), \text{ für } n \geq 3.$$

Mit der Definition $D(n) = T_{\text{rek}}(n) + 3$ erhält man dann die einfachere Rekursion

$$D(n) = \begin{cases} 3, & \text{falls } n < 3, \\ D(n-1) + D(n-2), & \text{sonst.} \end{cases}$$

Lemma 1.1 *Es gilt* $D(n) = 3 \cdot f_n$.

Beweis. Der Beweis erfolgt durch Induktion. Die Behauptung gilt trivialerweise für $n = 1$ und $n = 2$. Für den Schritt von $n - 1$ nach n ergibt sich

$$\begin{aligned} D(n) &= D(n-1) + D(n-2) \\ &= 3 \cdot f_{n-1} + 3 \cdot f_{n-2} \\ &= 3 \cdot f_n. \end{aligned}$$

□

Satz 1.2 *Die Zahl der notwendigen Additionen und Subtraktionen zur Berechnung von* f_n *mit der rekursiven Implementierung ist* $T_{\text{rek}}(n) = 3 \cdot f_n - 3$.

□

Um diese Aussage würdigen zu können, muss man wissen, wie groß f_n ist.

Lemma 1.3 *Für* $n > 2$ *gilt*

$$2^{\lfloor \frac{n-1}{2} \rfloor} \leq f_n \leq 2^{n-2}.$$

Beweis. Die Folge der f_n ist offensichtlich monoton wachsend und es gilt

$$f_n = f_{n-1} + f_{n-2} \geq 2 \cdot f_{n-2}.$$

Durch wiederholte Anwendung dieser Beziehung erhalten wir

$$f_n \geq 2 \cdot f_{n-2} \geq 2 \cdot 2 \cdot f_{n-4} \geq \quad \ldots \quad \geq 2^{\lfloor \frac{n-1}{2} \rfloor} \cdot f_{n-2 \cdot \lfloor \frac{n-1}{2} \rfloor} = 2^{\lfloor \frac{n-1}{2} \rfloor},$$

da $2 \cdot \lfloor \frac{n-1}{2} \rfloor = 2 \cdot \frac{n-1}{2} = n - 1$, falls n ungerade, und $2 \cdot \lfloor \frac{n-1}{2} \rfloor = 2 \cdot \frac{n-2}{2} = n - 2$, falls n gerade. Auf der anderen Seite haben wir $f_{n-1} + f_{n-2} \leq 2 \cdot f_{n-1}$ und eine wiederholte Anwendung liefert dann

$$f_n \leq 2 \cdot f_{n-1} \leq 2^2 \cdot f_{n-2} \leq \quad \ldots \quad \leq 2^{n-2} \cdot f_2 = 2^{n-2}.$$

□

Für $n = 101$ werden also mehr also 2^{50} Operationen ausgeführt!

1.1.2 Iterative Berechnung

Klarerweise sollten die Fibonacci-Zahlen iterativ etwa wie folgt berechnet werden.

```
int f(int n)
{
  if (n<3) return 1;
  else {
    int a=1, b=1, z;
    for (int i=3; i<=n; i++){
      z = a + b;
      a = b;
      b = z;
    }
    return z;
  }
}
```

Der Anzahl $T_{\text{it}}(n)$ der erforderlichen arithmetischen Operationen lässt sich leicht bestimmen. Es gelten $T_{\text{it}}(1) = T_{\text{it}}(2) = 0$ und $T_{\text{it}}(n) = 2 \cdot (n - 2)$, für $n \geq 3$, denn für jedes i ist jeweils eine Addition zur Berechnung von z und eine Addition zur Erhöhung von i erforderlich.

Satz 1.4 *Die Zahl der notwendigen Additionen zur Berechnung von f_n mit der iterativen Implementierung ist $T_{\text{it}}(n) = \max\{0, 2(n-2)\}$.*

□

Diese Implementierung ist offenbar der rekursiven vorzuziehen. Man beachte insbesondere, dass nur n in die Anzahl der Operationen eingeht, nicht aber f_n.

1.1.3 Berechnung durch iteriertes Quadrieren

Wir entwickeln nun eine andere Formel zur Berechnung der Fibonacci-Zahlen, die neben Additionen allerdings auch Multiplikationen erfordert.
Benutzt man Vektoren (f_n, f_{n-1}), dann gilt mit der Matrix

$$F = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

der Zusammenhang

$$(f_n, f_{n-1}) = (f_{n-1} + f_{n-2}, f_{n-1}) = (f_{n-1}, f_{n-2}) \cdot F.$$

Lemma 1.5 *Für $n \geq 2$ gilt $(f_n, f_{n-1}) = (f_2, f_1) \cdot F^{n-2}$.*

Beweis. Dies ergibt sich leicht durch Induktion. Es gelten

$$(1,1) = (1,1) \cdot F^0 = (1,1) \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \text{ und } (f_3, f_2) = (2,1) = (1,1) \cdot F.$$

Allgemein ergibt sich

$$(f_{n-1}, f_{n-2}) \cdot F = (f_{n-2}, f_{n-3}) \cdot F^2 = \ldots = (f_2, f_1) \cdot F^{n-2}.$$

□

Die Idee zur schnellen Berechnung der Potenzen der Matrix F lässt sich an einem Beispiel wie folgt veranschaulichen. Statt etwa 2^8 mit 7 Multiplikationen zu berechnen, kann man auch einfach 2 dreimal quadrieren und erhält

$$\left(\left(2^2\right)^2\right)^2 = 2^8$$

gemäß den Gesetzen der Potenzrechnung.
Der nächste Algorithmus folgt diesem Prinzip. Allgemein ergibt sich offenbar die Formel

$$F^n = \begin{cases} F^{\frac{n}{2}} \cdot F^{\frac{n}{2}}, & \text{falls } n \text{ gerade,} \\ F^{\frac{n-1}{2}} \cdot F^{\frac{n-1}{2}} \cdot F, & \text{falls } n \text{ ungerade.} \end{cases}$$

Ob n gerade oder ungerade ist, kann man durch Testen des letzten Bits der Binärdarstellung von n feststellen. Für die Analyse verwenden wir die Bezeichnungen

$l(n)$ = Länge der Binärdarstellung von n (ohne führende Nullen),

$\mathbf{1}(n)$ = Anzahl der Einsen in der Binärdarstellung von n.

Es gilt offenbar

$$l(n) = \lfloor \log(n) \rfloor + 1 = \lceil \log(n+1) \rceil.$$

(Wenn nichts Anderes gesagt wird, verwenden wir immer Logarithmen zur Basis 2.)

Satz 1.6 *Zur Bildung von F^n sind $l(n) - 1$ Divisionen durch 2 und $l(n) + \mathbf{1}(n) - 2$ Multiplikationen von Potenzen von F notwendig.*

Beweis. Die Zahl der Multiplikationen wird per Induktion über $l(n)$ bewiesen.
Für $l(n) = 1$ ergibt sich $n = 1$ und es sind $0 = l(n) - 1$ Divisionen und $l(n) + \mathbf{1}(n) - 2 = 0$ Multiplikationen nötig.

Die Behauptung gelte für $l(n) = k$. Wir unterscheiden, ob n gerade oder ungerade ist.

(i) $l(n) = k + 1$ und n gerade.
Dann ist $l(\frac{n}{2}) = k$ und die Anzahl der Multiplikationen ist

$$\begin{aligned} 1 + \text{ Anzahl Multiplikationen für } F^{\frac{n}{2}} \\ &= 1 + l(\tfrac{n}{2}) + \mathbf{1}(\tfrac{n}{2}) - 2 \\ &= 1 + (l(n) - 1) + \mathbf{1}(n) - 2 \\ &= l(n) + \mathbf{1}(n) - 2. \end{aligned}$$

(ii) $l(n) = k + 1$ und n ungerade.
Dann ist $l(\lfloor \frac{n}{2} \rfloor) = k$ und die Zahl der Multiplikationen ist

$$\begin{aligned} 2 + \text{ Anzahl Multiplikationen für } F^{\lfloor \frac{n}{2} \rfloor} \big(= F^{\frac{n-1}{2}}\big) \\ &= 2 + l(\lfloor \tfrac{n}{2} \rfloor) + \mathbf{1}(\lfloor \tfrac{n}{2} \rfloor) - 2 \\ &= 2 + (l(n) - 1) + (\mathbf{1}(n) - 1) - 2 \\ &= l(n) + \mathbf{1}(n) - 2. \end{aligned}$$

Die Anzahl der Divisionen durch 2 ist klar. □

Jetzt kann der folgende Satz für die Anzahl $T_q(n)$ von arithmetischen Operationen zur Berechnung von f_n mit der neuen Methode gezeigt werden.

Satz 1.7 *Die Zahl der arithmetischen Operationen zur Berechnung von f_n mittels iterierter Quadrierung ist*

$$T_q(n) = 13 \cdot \lfloor \log(n-2) \rfloor + 12 \cdot \mathbf{1}(n-2) - 10.$$

Beweis. Die Multiplikation zweier $(2,2)$-Matrizen erfordert (ohne Vereinfachungen) acht Multiplikationen und vier Additionen ganzer Zahlen.
Die Berechnung von F^{n-2} benötigt also

$$\begin{aligned} 12 \cdot (l(n-2) + \mathbf{1}(n-2) - 2) + l(n-2) - 1 \\ &= 13 \cdot l(n-2) + 12 \cdot \mathbf{1}(n-2) - 25 \\ &= 13 \cdot (\lfloor \log(n-2) \rfloor + 1) + 12 \cdot \mathbf{1}(n-2) - 25 \\ &= 13 \cdot \lfloor \log(n-2) \rfloor + 12 \cdot \mathbf{1}(n-2) - 12 \end{aligned}$$

arithmetische Operationen.
Jetzt muss noch $(1,1) \cdot F^{n-2}$ ausgerechnet werden, bzw. zur Bestimmung von f_n genügt es, die erste Spalte aufzuaddieren. Dann kommt zu Anfang noch eine Subtraktion zur Bildung von $n-2$ hinzu und wir erhalten die Behauptung für T_q. □

Nehmen wir an, dass eine arithmetische Operation eine Mikrosekunde benötigt, so ergeben sich als maximale Werte von n, für die f_n in einer Millisekunde, einer Sekunde, einer Minute bzw. einer Stunde bei den einzelnen Varianten bestimmt werden kann, die folgenden Zahlen.

	1 ms	1 s	1 min	1 h
rekursiv	14	28	37	45
iterativ	500	$5 \cdot 10^5$	$3 \cdot 10^7$	$2 \cdot 10^9$
iteriertes Quadrieren	10^{12}	10^{12000}	10^{700000}	10^{10^8}

Bemerkung

a) *Vergrößert sich die Rechengeschwindigkeit um den Faktor* 1000*, kann mit der rekursiven Methode in einer Sekunde auch nur f_{43} berechnet werden.*

b) *Bei den Zeiten ist allerdings ignoriert, dass sehr große Zahlen auftreten und arithmetische Operationen dann natürlich auch entsprechend länger dauern.*

c) *Man kann die Berechnung noch etwas beschleunigen, wenn man ausnutzt, dass die Matrizen F^i symmetrisch sind.*

□

1.2 Algorithmen

Jeder hat sicher eine intuitive Vorstellung der wesentlichen Eigenschaften eines Algorithmus und würde etwa folgende Definitionen geben.

> *„Rechenverfahren, das in genau festgelegten Schritten vorgeht."*
>
> *„Rechenvorschrift, die mit endlichen Mitteln beschreibbar ist und angibt, wie zu jeder Eingabe in endlich vielen Rechenschritten eine Ausgabe produziert wird."*

Natürlich muss der Algorithmusbegriff aber weiter gefasst und nicht auf Rechenproblemem beschränkt werden. Auch Alltagsvorschriften wie Kochrezepte, Bastelanleitungen oder Partituren haben algorithmischen Charakter. Generell kann man einen Algorithmus als Vorschrift sehen, die eine Eingabe in eine Ausgabe umwandelt und so ein vorgegebenes Problem löst. Zum engen Zusammenhang zwischen Algorithmen und Daten seien noch zwei Zitate von Wirth, dem Entwickler der Programmiersprachen Pascal und MODULA, angegeben.

> *„Programme sind letztlich konkrete Formulierungen abstrakter Algorithmen, die sich auf bestimmte Darstellungen und Datenstrukturen stützen."*
>
> *„Erstellung von Programmen und Datenstrukturierung sind untrennbar ineinander gehende Themen."*

Algorithmen besitzen die folgenden charakteristischen Eigenschaften.

- **Abstrahierung**

  Ein Algorithmus löst eine Klasse von Problemen. Ein zu lösendes konkretes Problem wird durch Parameter spezifiziert.

- **Finitheit**

  Die Beschreibung des Algorithmus besitzt eine endliche Länge (**statische Finitheit**). Zu jedem Zeitpunkt der Abarbeitung belegt der Algorithmus mit seinen Daten und Zwischenergebnissen nur endlich viel Platz (**dynamische Finitheit**).

- **Terminierung**

  Algorithmen, die für jede Eingabe nach endlich vielen Schritten anhalten und ein Ergebnis liefern, heißen **terminierend**, sonst **nichtterminierend**. In der Regel ist man nur an terminierenden Algorithmen interessiert, es gibt aber auch sinnvolle nichtterminierende (Betriebssysteme, Überwachungsprogramme).

- **Determinismus**

  Ein Algorithmus heißt **deterministisch**, falls es zu jedem Zeitpunkt seiner Ausführung genau eine Möglichkeit zur Fortsetzung gibt. Gibt es mehrere Möglichkeiten, ist der Algorithmus **nichtdeterministisch**.

Wird die Auswahl der Möglichkeiten über Wahrscheinlichkeiten gesteuert, heißt er **stochastisch** oder **randomisiert**.

- **Determiniertheit**

  Ein Algorithmus ist **determiniert**, wenn zur gleichen Eingabe immer die gleiche Ausgabe geliefert wird. (Er muss dazu nicht deterministisch sein.)

- **Korrektheit**

  Es sollte natürlich beweisbar sein, dass der Algorithmus ein vorgegebenes Problem löst.

Zentrale Aufgabe der Informatik ist es, Probleme durch Algorithmen zu lösen. Aber nicht nur die prinzipielle algorithmische Lösbarkeit ist von Interesse, sondern auch die Frage nach dem bestmöglichen Algorithmus (bzgl. Zeit- oder Speicherbedarf).
Es gibt verschiedene Möglichkeiten, Algorithmen zu beschreiben.

a) **Iterative Darstellung**

   Ein Algorithmus besteht aus einer Folge von Operationen zusammen mit einer Vorschrift, in welcher Reihenfolge die Operationen auszuführen sind.

b) **Rekursive/induktive Darstellung**

   Der Algorithmus gibt an, wie das Ergebnis in einigen Spezialfällen erhalten wird, und führt andere Berechnungen schrittweise hierauf zurück (wobei er seine eigene Beschreibung verwenden darf).

c) **Spezifikation**

   Der Algorithmus wird indirekt durch Eigenschaften beschrieben (spezifiziert), die das Ergebnis und mögliche Zwischenergebnisse haben. Der Algorithmus selbst muss hieraus entwickelt werden.

Aus diesen Beschreibungsmöglichkeiten für Algorithmen lassen dich Paradigmen für den Entwurf von Programmiersprachen ableiten.

Imperative Programmiersprachen (Assembler, `FORTRAN`, `C`, `C++`, Java)

Ausgehend von elementaren Operationen gibt man eine Folge von Anweisungen an, aus der die Reihenfolge, in der die Operationen auszuführen sind, hervorgeht. Darstellungselemente sind etwa

- Verwendung von Speicherplätzen,
- Fallunterscheidungen und Iterationen,
- Prozeduren,
- rekursive Prozeduren.

Funktionale Programmiersprachen (LISP, Haskell)

Auf die Angabe von Speicherplätzen oder einer Ablaufsteuerung wird verzichtet. Die Beschreibung erfolgt mit

- Einsetzungen,
- Alternativen,
- Rekursionen.

Prädikative Programmiersprachen (PROLOG)

Es werden Beziehungen zwischen Objekten durch Prädikate beschrieben. Angegeben werden

- Fakten (explizite Angabe von Objekten, für die gewisse Prädikate gelten),
- Regeln (Vorschriften zur Ableitung neuer Beziehungen aus gegebenen).

Man versucht dann, neue Fakten zu schaffen, bis die Lösung gefunden ist.

1.3 Turing-Maschinen

Wir wollen zumindest kurz einen klassischen Ansatz zur Formalisierung des Algorithmusbegriffs skizzieren: die **Turing-Maschine** (benannt nach dem englischen Mathematiker Alan Turing). Die Turing-Maschine (TM) ist nur ein Gedankenmodell, lässt sich aber anschaulich mit einer primitiven Hardware erklären. Die TM besteht aus einem linearen beidseitig unbeschränkten **Band**, das in **Zellen** eingeteilt ist. Zum Belegen der Zellen steht ein endliches **Alphabet** A zur Verfügung, ein besonderes **Leerzeichen** ist ausgezeichnet. Gelesen und beschrieben werden Zellen über einen **Lese-Schreib-Kopf** (LS-Kopf), der immer über einer Zelle (**Arbeitsfeld**) steht und nach links oder rechts verschoben werden kann.
Gesteuert wird die TM über eine Steuereinheit, in der ein Programm (**Turing-Programm**) abläuft. Dieses Programm ist eine endliche, durchnumerierte Folge von Instruktionen:

```
1:   Instruktion_1
2:   Instruktion_2
     :
     :
r:   Instruktion_r
r+1: stop
```

Es gibt in der Literatur verschiedene Instruktionssätze, die aber nichts an der Mächtigkeit der TM ändern. Wir verwenden den folgenden Instruktionssatz:

`a`	Schreibe 'a' auf das Arbeitsfeld
`R`	Bewege den LS-Kopf um eine Zelle nach rechts
`L`	Bewege den LS-Kopf um eine Zelle nach links
`goto k`	Gehe zur Instruktion `k`
`a ? k: l`	Falls 'a' auf den Arbeitsfeld steht, gehe zu Instruktion `k`, sonst zu `l`
`stop`	Beende die Berechnung

Es wird angenommen, dass die **Eingabe** auf dem Band steht. Der LS-Kopf steht auf der ersten Zelle links von der Eingabe, die Speicherzellen rechts und links von der Eingabe enthalten Leerzeichen. Die TM beginnt mit der Ausführung der ersten Instruktion. Liegt kein Sprungbefehl vor, wird jeweils mit der nächsten Instruktion fortgefahren. Die **Ausgabe** einer TM ist durch den Bandinhalt bei `stop` gegeben.

Beispiel Zu einem Eingabewort „$\texttt{a}^n\texttt{\_b}^m$“ wird die Ausgabe „$\texttt{b}^{n+m+1}$“ mit dem folgenden Programm geliefert.

```
1: R
2: a ? 3 : 6
3: b
4: R
5: goto 2
6: b
7: R
8: b ? 7 : 9
9: stop
```

Hier ist nicht vorgesehen, dass „falsche“ Eingaben erfolgen. □

Die TM realisiert also eine Funktion $f : A_1^* \to A_2^*$, die ein Wort über dem Eingabealphabet A_1 in ein Wort über dem Ausgabealphabet A_2 überführt. Diese Funktion kann partiell sein, denn die TM hält unter Umständen für manche Eingaben nicht an. Die TM kann natürlich auch „richtige“ Funktionen berechnen, wenn man die Eingaben und Ausgaben als Zahlen interpretieren kann.
Turing formulierte die These, dass Turing-Maschinen mächtig genug sind, um alle Algorithmen realisieren können.

Turingsche These *Der intuitive Algorithmusbegriff stimmt mit dem exakt definierten Begriff der Turing-Maschine überein.*

□

Obwohl sie sehr einfach ist, bildet die TM doch die wesentlichen Eigenschaften realer Computer und Programmabläufe nach.
In der Berechenbarkeitstheorie werden prinzipielle Fragen und Konzepte bei Algorithmen anhand von Turing-Maschinen (oder ähnlichen exakten Modellen) untersucht.
Das **Halteproblem** etwa besteht darin, für eine gegebene TM M und ein Eingabewort w zu entscheiden, ob M angewendet auf w anhält oder nicht. Dieses Problem ist TM-unentscheidbar, d. h. es gibt keine TM, die dieses Problem allgemein lösen kann. Nach der Turingschen These gibt es also keinen Algorithmus, der dieses Problem löst.
Ebenfalls TM-unentscheidbar sind das **Korrektheitsproblem** „*Berechnet M die gewünschte Funktion f?*“ und das **Äquivalenzproblem** „*Berechnen M und M' die gleiche Funktion?*“.
Interessant sind auch Fragen nach der Komplexität von TM-Rechnungen, d. h. Informationen über

$\text{Time}_M(w)$: Anzahl der Schnitte von M bei Anwendung auf w,

$\text{Space}_M(w)$: Anzahl der benutzten Zzellen des Bandes bei Anwendung von M auf w.

Die Verwendung einer Turing-Maschine als exaktes Algorithmusmodell ist nur für die Theorie tauglich. Wir werden einen pragmatischen Zugang wählen und die folgenden Annahmen machen.

- Ein Algorithmus ist durch ein `C`-Programm spezifiziert.
- Der Computer ist eine Maschine, die `C`-Programme ausführen kann.
- Der Speicher ist beliebig groß.
- Speicherzellen können beliebig große Zahlen aufnehmen.
- Alle elementaren Operationen (Lesen, Speichern, Sprung, Addition, Multiplikation, Vergleich) benötigen die gleiche Zeit (**Einheitskostenmodell**).
- Die Laufzeit ist die Anzahl ausgeführter elementarer Operationen.
- Der Speicherbedarf ist die Anzahl notwendiger Speicherzellen.

Diese Vereinfachungen sind bis auf die Zahlengrößen unwesentlich. Im Prinzip ist hier die sogenannte **Random Accesss Machine** (**RAM**) skizziert.

1.4 Analyse von Algorithmen

Wir wollen in diesem Abschnitt einige Grundgedanken zur Analyse und damit zum Vergleich von Algorithmen diskutieren.
Als Beispiel wird das **Sortierproblem** betrachtet, bei dem eine Menge vorgegebener Zahlen aufsteigend zu sortieren ist.

Sortierproblem

Gegeben ist ein Folge von n Zahlen $a_1, a_2, \ldots, a_n$. Es ist eine Anordnung $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$ der Zahlen zu bestimmen, so dass $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_n}$.

Im Unterschied zu dieser allgemeinen Definition eines **Problems** sprechen wir von einem konkreten **Problembeispiel**, wenn konkrete (und vollständige) Eingabedaten vorliegen.

Normalerweise gibt es mehrere Möglichkeiten zur Lösung eines Problems. Speziell mit dem Sortierproblem werden wir uns noch ausführlich auseinander setzen. Wir betrachten hier nur einen einfachen naheliegenden Ansatz, das **Sortieren durch Einfügen**.
Das folgende C-Programm implementiert diesen Algorithmus.

```
void insertionSort(
      int *A,
      int  n
)
// Sortiere A[0]...A[n-1] aufsteigend
{
      int i,j,key;
(1)   for (j=1; j<n; j++) {
(2)       key = A[j];
          // Fuege A[j] in Folge A[0]...A[j-1] ein
(3)       i = j - 1;
          // Verschiebe, bis Einfuegeposition erreicht
(4)       while ( (i>=0) && (A[i]>key) ) {
(5)             A[i+1] = A[i];
(6)             i--;
          }
(7)       A[i+1] = key;
      }
}
```

Beispiel Die Folge „`5 2 4 6 1 3`“ wird schrittweise wie folgt umgeordnet:

```
5 2 4 6 1 3
2 5 4 6 1 3
2 4 5 6 1 3
2 4 5 6 1 3
1 2 4 5 6 3
1 2 3 4 5 6
```

□

Neben der Korrektheit eines Algorithmus sind natürlich **Zeit-** und **Speicherbedarf** von Interesse. (Je nach Anwendung könnten auch Eigenschaften wie erforderliche Rechengenauigkeit, Kommunikationsbandbreite oder benötigte Peripheriegeräte relevant sein.)
Wir werden uns fast immer mit der **Laufzeit** auseinander setzen. Hier stellt sich zunächst die Frage, wie Laufzeit eigentlich spezifiziert werden kann. Unhandlich bzw. meist unbrauchbar ist die Angabe der Laufzeit über Zeitmessungen auf einer konkreten Maschine für mehrere Beispieldaten. Diese Laufzeiten sind nicht uninteressant, aber in der Regel untauglich zur Beantwortung allgemeiner Fragen wie:

- Wie vergrößert sich die Laufzeit im schlechtesten Fall in Abhängigkeit von der Größe des Problems?
- Wie sehen günstige oder ungünstige Eingaben aus?

Außerdem möchte man die Laufzeit möglichst maschinenunabhängig charakterisieren. Die Laufzeit wird als Funktion der Größe der Eingabe angegeben, wobei die Eingabegröße sinnvoll definiert werden muss. Im Falle des Sortierproblems ist sicherlich n, d. h. die Anzahl der zu sortierenden Elemente, eine vernünftige Charakterisierung der Problemgröße. Im Allgemeinen kommt etwa auch die Datenlänge im Binärformat in Betracht.
Es soll untersucht werden, wie sich die Laufzeit der Sortierens durch Einfügen mit wachsendem n erhöht. Wir wollen dazu nicht alle Elementarschritte zählen, sondern nehmen an, dass die Ausführung einer Programmzeile $(1), \ldots, (7)$ jeweils konstante Zeit $c_1, \ldots, c_7$ erfordert.

Sei t_j die Anzahl der Ausführungen des Tests in Zeile (4) für das aktuelle j. Dann ergibt sich die Laufzeit als

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) \\ &\quad + c_4 \cdot \sum_{j=1}^{n-1} t_j + c_5 \cdot \sum_{j=1}^{n-1} (t_j - 1) \\ &\quad + c_6 \cdot \sum_{j=1}^{n-1} (t_j - 1) \\ &\quad + c_7 \cdot (n-1). \end{aligned}$$

Wir analysieren zwei konkrete Fälle.

(i) Die Daten sind bereits sortiert.

Dann ist $t_j = 1$ für alle j und wir erhalten

$$\begin{aligned} T(n) &= c_1 \cdot n + (c_2 + c_3 + c_7) \cdot (n-1) + c_4 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

für entsprechende Definitionen von a und b.
$T(n)$ wächst also in diesem Fall linear mit n.

(ii) Die Eingabe ist umgekehrt sortiert.

Der Test in (4) erfolgt dann für alle $i = j-1, j-2, \ldots 1, 0, -1$ und somit ist $t_j = j+1$ für alle $j = 1, 2 \ldots, n-1$.
Bekanntlich gelten die Formeln

$$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} \quad \text{und} \quad \sum_{j=1}^{n-1} j + 1 = \frac{n(n+1)}{2} - 1.$$

Einsetzen liefert dann

$$\begin{aligned} T(n) &= c_1 \cdot n + (c_2 + c_3) \cdot (n-1) \\ &\quad + c_4 \cdot \Big(\frac{n(n+1)}{2} - 1 \Big) \\ &\quad + (c_5 + c_6) \cdot \frac{n(n-1)}{2} + c_7 \cdot (n-1) \\ &= (c_4 + c_5 + c_6) \cdot \frac{n^2}{2} + (c_1 + c_2 + c_3 + c_7) \cdot n + (c_4 + c_5 + c_6) \cdot \frac{n}{2} \\ &\quad - c_2 - c_3 - c_n - c_7 \\ &= a \cdot n^2 + b \cdot n + c \end{aligned}$$

für entsprechendes Setzen von a, b und c.
Die Laufzeit nimmt also in diesem Fall quadratisch mit n zu.

Hauptsächlich zwei Laufzeitanalysen sind interessant.

- **Worst-Case-Laufzeit**: längste mögliche Laufzeit in Abhängigkeit von der Problemgröße.
- **Average-Case-Laufzeit**: durchschnittliche Laufzeit in Abhängigkeit von der Problemgröße.

Im Allgemeinen ist die durchschnittliche Laufzeit nur schwer zu analysieren (was ist eine durchschnittliche Eingabe?) und auch für den konkreten Fall nur bedingt aussagekräftig. Die Worst-Case-Laufzeit lässt sich meist angeben oder zumindest gut abschätzen und liefert immerhin eine Laufzeit, die garantiert nicht überschritten wird.
Es ist klar, dass man an einer derart detaillierten Analyse, wie wir sie hier vorgenommen haben, nicht unbedingt interessiert ist. Abgesehen davon ist sie sehr mühsam. Man betrachtet daher nur die Größenordnung des Wachstums von $T(n)$ mit wachsendem n.

Definition 1.8 *Sei $f : \mathbf{N_0} \to \mathbf{N_0}$ eine Funktion. Dann bezeichnen $O(f)$, $\Omega(f)$ und $\Theta(f)$ die folgenden Mengen:*

$$O(f) = \{g : \mathbf{N_0} \to \mathbf{N_0} \mid \exists c > 0\ \exists n_0\ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\},$$

$$\Omega(f) = \{g \mid \exists c > 0\ \exists n_0\ \forall n \geq n_0 : g(n) \geq c \cdot f(n)\},$$

$$\Theta(f) = \{g \mid \exists c > 0, d > 0\ \exists n_0\ \forall n \geq n_0 : c \cdot f(n) \leq g(n) \leq d \cdot f(n)\}.$$

□

Man spricht dann von **O-Notation**, bzw. **Ω-Notation** für obere bzw. untere (asymptotische) Schranken und von **Θ-Notation** für die genaue Wachstumsrate.

Bemerkungen

a) *Statt $g \in O(f)$ schreibt man einfach $g = O(f)$ und rechnet auch mit den Symbolen (z. B. $n^2 + 5n = n^2 + \Theta(n) = \Theta(n^2)$).*

b) *Manchmal wird auch $\Omega(f) = \{g \mid \exists c > 0$ mit $g(n) \geq c \cdot f(n)$ für unendlich viele $n\}$ definiert.*

Man vergleiche die beiden Definition von $\Omega(f)$ für die Funktion

$$T(n) = \begin{cases} 1, & n \text{ ungerade,} \\ n^2, & n \text{ gerade.} \end{cases}$$

□

Beispiel An einem Wettbewerb zum Sortieren von Zahlen nehmen Teilnehmer A und B teil mit den Spezifikationen:

A: Supercomputer mit 10^{12} Operationen/sec (1 Teraflop), guter Programmierer, Algorithmus *Insertionsort* mit Laufzeit $2n^2$,

B: PC mit 10^9 Operationen/sec (1 Gigaflop), schlechter Programmierer, Algorithmus *Mergesort* mit Laufzeit $50n \log n$.

Für $n = 1\,000\,000\,000$ braucht A $2 \cdot (10^9)^2/10^{12} \approx 2\,000\,000$ Sekunden, d. h. etwa 23 Tage, während B nur $50 \cdot 10^9 \cdot \log 10^9/10^9 \approx 1\,495$ Sekunden, d. h. etwa 16:40 Minuten benötigt. □

Die bessere asymptotische Laufzeit hat auch praktische Bedeutung. Wie Computer-Hardware sind also auch Algorithmen Technologie!

Bisweilen betont man auch, dass sich Funktionen beliebig weit unterscheiden.

Definition 1.9 *Für $f : \mathbf{N_0} \to \mathbf{N_0}$ ist*

$$o(f) = \Big\{g : \mathbf{N_0} \to \mathbf{N_0} \;\Big|\; \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0\Big\},$$

$$\omega(f) = \Big\{g : \mathbf{N_0} \to \mathbf{N_0} \;\Big|\; \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\Big\}.$$

□

1.5 Algorithmische Prinzipien

Zur Lösung eines Problems gibt es normalerweise mehrere algorithmische Ansätze, die unterschiedliche Eigenschaften haben. Am Beispiel des **Maximum-Subarray-Problems** sollen einige Prinzipien vorgestellt werden.

Maximum-Subarray-Problem

Gegeben ist eine Folge von n ganzen Zahlen (in einem Array $A[0], A[1], \ldots, A[n-1]$). Zu bestimmen ist eine zusammenhängende Teilfolge mit maximaler Summe der Elemente. Die leere Folge mit Summe 0 ist erlaubt.

Beispiel Für die Eingabe „`31 -41 59 26 -53 58 97 -93 -23 84`“ ergibt sich als maximale Teilfolge „`59 26 -53 58 97`“ mit Summe 187. □

Algorithmus 1: Enumeration

Es werden einfach alle möglichen Anfangs- und Endpositionen von Teilfolgen enumeriert und diese jeweils aufsummiert.

```
int maxSubarray1(
      int *A,
      int  n
)
// Loesung des Maximum Subarray Problems durch Enumeration
{
      int l,r,i,summe,maxsum;
      maxsum = 0;
      // Betrachte alle Teilfolgen
(1)   for (l=0; l<n; l++) { // linkes Ende
(2)       for (r=l; r<n; r++) { // rechtes Ende
(3)           summe = 0;
(4)           for (i=l; i<=r; i++)  // Summation
(5)               summe += A[i];
(6)           if (summe>maxsum) maxsum = summe;
          }
      }
      return(maxsum);
}
```

Wir nehmen wieder an, dass die Ausführung einer Programmzeile $(1), \ldots, (6)$ jeweils konstante Zeit $c_1, \ldots, c_6$ erfordert. Eine Analyse der Worst-Case-Laufzeit in Abhängigkeit von n liefert

$$\begin{aligned} T(n) &= 2 + c_1 \cdot n + (c_2 + c_3 + c_6) \cdot \sum_{l=0}^{n-1} (n-l) \\ &\quad + (c_4 + c_5) \cdot \sum_{l=0}^{n-1} \sum_{r=l}^{n-1} (r-l+1) \\ &\qquad \text{(kleinere unwesentliche Ungenauigkeiten hier)} \\ &= 2 + O(n) + O(n^2) + O(n^3) \\ &= O(n^3). \end{aligned}$$

Man kann hier auch $T(n) = \Theta(n^3)$ schreiben, da die Laufzeit immer so groß ist.

Der Vorteil dieses naiven Ansatzes ist, dass er sehr schnell implementiert ist. Eine Laufzeit der Ordnung $O(n^3)$ könnte aber für die Anwendung bereits problematisch sein.

Algorithmus 2: Divide-and-Conquer

Dieser Algorithmus geht von den folgenden Überlegungen aus:

(i) Wird die Folge in der Mitte in die Teile (1) und (2) geteilt, so liegt die maximale Teilfolge entweder ganz in (1) oder ganz in (2) oder sie umfasst die Trennstelle.

(ii) Im letzten Fall gilt für die beiden Teilfolgen, aus denen sich die maximale Folge zusammensetzt, dass die Summe der Elemente maximal unter den Teilfolgen ist, die das Randelement an der Trennstelle enthalten.

Man löst das Problem also durch Aufteilen in Teilprobleme und konstruiert dann die Lösung auf Basis der Lösungen der Teilprobleme (mit eventueller Zusatzarbeit).

Beispiel Die Eingabe sei „`31 -41 59 26 -53 58 97 -93`“. Im ersten Schritt wird in der Mitte geteilt. Das linke Randmaximum ist 85 und das rechte 102. Dies liefert bereits eine Lösung mit Wert 187. Die beiden Teilprobleme sind P1 („`31 -41 59 26`“) und P2 („`-53 58 97 -93`“).
Der Ablauf des Verfahrens ist in Abb. 1.1 dargestellt.

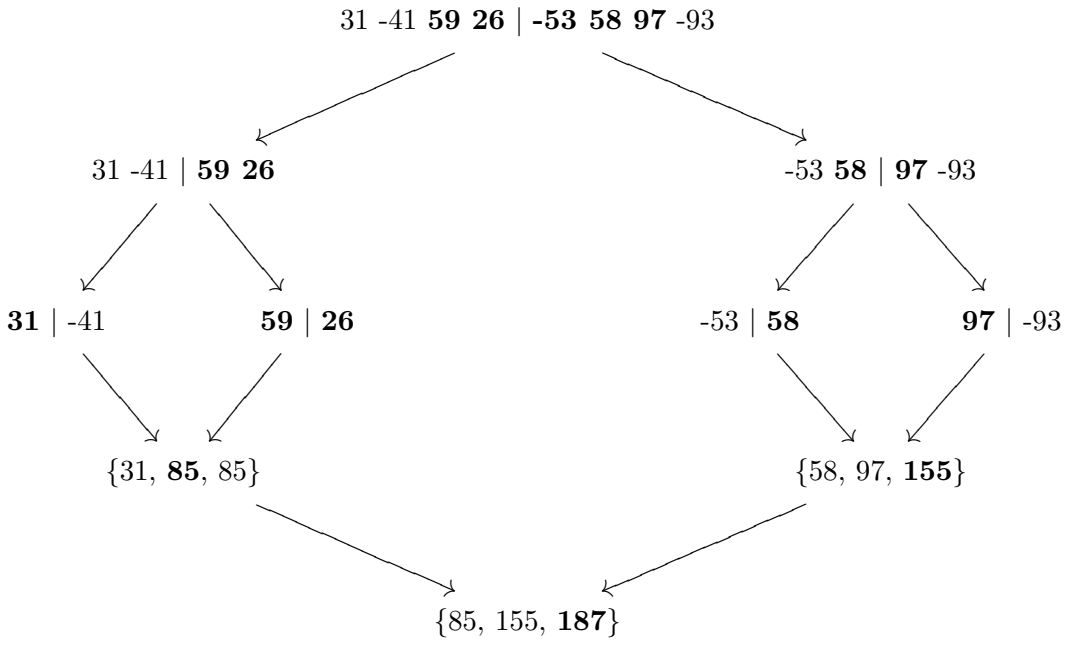

Abb. 1.1 Divide-and-Conquer-Algorithmus

Das linke Randmaximum (rechte Randmaximum) für P1 ist 0 (85) und für P2 ist es 58 (97). So ergeben sich erste Lösungen von 85 bzw. 155 für diese Teilprobleme.
P1 wird in P3 („`31 -41`“) und P4 („`59 26`“) zerlegt. Analog wird P2 in P5 („`-53 58`“) und P6 („`97 -93`“) zerlegt.
Diese Probleme sind trivial lösbar und es ergeben sich für P3, P4, P5 bzw. P6 die Lösungen 31, 85, 58 bzw. 97.
Die Lösung von P1 hat dann den Wert $\max\{31, 85, 85\} = 85$ und die von P2 hat den Wert $\max\{58, 97, 155\} = 155$.
Als optimale Lösung des Ausgangsproblems ergibt sich nun $\max\{85, 155, 187\} = 187$ mit der zugehörigen Teilfolge „59 26 -53 58 97“. □

Das folgende rekursive C-Programm implementiert die Divide-and-Conquer-Idee.

```
int maxSubarray2(
    int *A,
    int  n
)
// Loesung des Maximum Subarray Problems nach dem Divide-and-Conquer Prinzip
{
  int m,i,lmax,rmax,lsum,rsum,lbest,rbest,maxsum;
  if (n==1) {
      // Trivialer Fall
      if (A[0]>0) maxsum = A[0];
      else maxsum = 0;
  }
  else {
      // Loese die beiden Teilprobleme
      m = n/2;
      lbest = maxSubarray2(A,m);
      rbest = maxSubarray2(A+m,n-m);
      // Bestimme das linke Randmaximum
      lmax = lsum = 0;
      for (i=m-1;i>=0;i--) {
          lsum += A[i];
          if (lsum>lmax) lmax = lsum;
      }
      // Bestimme das rechte Randmaximum
      rmax = rsum = 0;
      for (i=m;i<n;i++) {
          rsum += A[i];
          if (rsum>rmax) rmax = rsum;
      }
      // Bestimme die beste Loesung
      maxsum = lmax + rmax;
      if (lbest>maxsum) maxsum = lbest;
      if (rbest>maxsum) maxsum = rbest;
  }
  return(maxsum);
}
```

Zur Charakterisierung der asymptotischen Laufzeit nehmen wir vereinfachend $n = 2^k$ an, damit die Aufteilung in der Mitte immer möglich ist.
Die Berechnung der Randmaxima erfolgt in der Zeit $\Theta(n)$ und für die Laufzeit $T(n)$ ergibt sich somit die Rekursionsformel

$$T(n) = \begin{cases} \Theta(1), & \text{falls } n = 1, \\ 2 \cdot T(\frac{n}{2}) + \Theta(n), & n > 1. \end{cases}$$

Wir werden im nächsten Abschnitt auf allgemeine Lösungsansätze für solche Formeln eingehen. Hier genügt eine einfache Betrachtung.
Nach der l-ten Aufteilung ergeben sich 2^l Teilprobleme. Der Gesamtaufwand zur Bestimmung der Randmaxima sowie des Maximums aus drei Werten ist $\Theta(n)$, da jedes Element in genau einem Teilproblem enthalten ist.
Das Problem wird $\log n$-mal aufgeteilt, der Gesamtaufwand ist also

$$(\log n + 1) \cdot \Theta(n) = \Theta(n \log n).$$

Dieses Verfahren ist folglich wesentlich schneller als der naive Ansatz. Es gibt aber noch eine bessere Lösung.

Algorithmus 3: Scan-Line-Verfahren

Die Divide-and-Conquer-Argumentation gilt für jede beliebige Teilung des Feldes. Wir verschieben nun den Trennstrich von links mit einem Element beginnend jeweils um ein Element nach rechts. Die maximale Teilfolge ist irgendein rechtes Randmaximum.
Zu jeder Position $l = 0, 1, \ldots, n-1$ aktualisieren wir nun jeweils

- die maximale Summe `maxsum` einer Teilfolge von $A[0], \ldots, A[l-1]$,
- das rechte Randmaximum `scanmax` von $A[0], \ldots, A[l-1]$.

Sei `maxsum` die maximale Summe einer Teilfolge von $A[0] \ldots A[l-1]$ und `scanmax` das aktuelle rechte Randmaximum.

(i) $\texttt{scanmax} + A[l] > 0$
Hier kann noch ein relevantes Randmaximum entstehen, es wird $\texttt{scanmax} = \texttt{scanmax} + A[l]$ gesetzt.

(ii) $\texttt{scanmax} + A[l] \leq 0$
Die Teilfolge ist jetzt uninteressant, es wird neu begonnen mit $\texttt{scanmax} = 0$.

(iii) Es wird getestet, ob $\texttt{scanmax} > \texttt{maxsum}$. Falls ja wird `maxsum` aktualisiert.

Die Initialisierung für $l = 0$ ist klar. Wir geben wieder eine `C`-Implementierung an.

```
int maxSubarray3(
    int *A,
    int  n
)
// Loesung des Maximum Subarray Problems nach dem Scan-Line Prinzip
{
  int l,scanmax,maxsum;

  scanmax = maxsum = 0;
  for (l=0;l<n;l++) {
      // Inspiziere Position l

      // Update fuer rechtes Randmaximum
      if (scanmax+A[l] > 0) scanmax += A[l];
      else scanmax = 0;

      // Bessere Loesung gefunden?
      if (scanmax>maxsum)
          maxsum = scanmax;
  }
  return(maxsum);
}
```

Die Laufzeit ist offenbar linear $\Theta(n)$. Da jedes Element von A betrachtet werden muss, kann es kein asymptotisch schnelleres Verfahren geben. Der Scan-Line-Algorithnus ist (asymptotisch) **optimal**.
Optimalität von Algorithmen kann allerdings nur in seltenen Fällen bewiesen werden.

Beispiel Ein Rechenzeitvergleich der 3 Implementierungen ergab

n	(1)	(2)	(3)
1000	21.3 s	0.0 s	0.0 s
10000	6 h	0.1 s	0.0 s
1000000	–	6.1 s	0.3 s

□

1.6 Lösungen von Rekursionsgleichungen

Die Laufzeit von Algorithmen lässt sich häufig über eine Rekursionsformel charakterisieren. Die Auflösung solcher Formeln ist nicht immer klar und erfordert etwas Erfahrung.

Induktive Einsetzungsmethode: „Rate die Lösung und bestätige sie durch Induktion."

Beispiel Für die Fibonacci-Zahlen

$$f(n) = \begin{cases} f(n-1) + f(n-2), & n \geq 3, \\ 1, & n \in \{1, 2\}, \end{cases}$$

vermuten wir exponentielles Wachstum $f(n) \geq a \cdot c^n$, für Konstanten a und c. Einsetzen liefert

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ &\geq a \cdot c^{n-1} + a \cdot c^{n-2} \\ &= a \cdot c^n \cdot \Big(\frac{1}{c} + \frac{1}{c^2}\Big). \end{aligned}$$

Die Behauptung gilt, wenn $\frac{1}{c} + \frac{1}{c^2} \geq 1$, d. h. wenn $0 \geq c^2 - c - 1$. Man kann c also als positive Lösung der Gleichung $c^2 - c - 1 = 0$ wählen und erhält $c = \frac{1}{2}(\sqrt{5} + 1) \approx 1.618$.
Die Zahl a lässt sich dann setzen als $a = \frac{1}{c^2}$. Es folgt $f(1) = 1 \geq a \cdot c^1 = \frac{1}{c} \approx 0.62$ und $f(2) = 1 \geq a \cdot c^2 = 1$ und auch der Induktionsanfang ist gegeben.
Umgekehrt kann man auch eine exponentielle obere Schranke $b \cdot d^n$ beweisen, denn es gilt

$$\begin{aligned} f(n) &\leq b \cdot d^{n-1} + b \cdot d^{n-2} \\ &= b \cdot d^n \cdot (\frac{1}{d} + \frac{1}{d^2}). \end{aligned}$$

Man kann $d = \frac{1}{2}(\sqrt{5}+1)$ und $b = \frac{1}{d}$ wählen. Es ist also $f(n) = \Theta(c^n)$ und es gilt die Abschätzung $0.38 \cdot 1.62^n \leq f(n) \leq 0.62 \cdot 1.62^n$. □

Iterationsmethode: „Setze wiederholt die Formel ein, bis ein geschlossener Ausdruck entsteht."

Beispiel Gegeben sei die Rekursion $T(n) = n + 3 \cdot T\big(\frac{n}{4}\big)$ (n sei 4er-Potenz).

$$\begin{aligned} T(n) = n + 3 \cdot T(\frac{n}{4}) &= n + 3 \cdot \Big(\frac{n}{4} + 3 \cdot T(\frac{n}{16})\Big) \\ &= n + \frac{3}{4} \cdot n + 3^2 \cdot T\big(\frac{n}{4^2}\big) + 3 \cdot T\big(\frac{n}{4^3}\big)\Big) \\ &= n + \frac{3}{4} \cdot n + \big(\frac{3}{4}\big)^2 \cdot n + 3^3 \cdot T\big(\frac{n}{4^3}\big) \\ &\;\;\vdots \\ &= n + \frac{3}{4} \cdot n + \ldots + \big(\frac{3}{4}\big)^{\log_4 n - 1} \cdot n + 3^{\log_4 n} \cdot T(1) \\ &\leq n \cdot \sum_{i=0}^{\infty} \big(\frac{3}{4}\big)^i + 3^{\log_4 n} \cdot T(1) \\ &= n \cdot \frac{1}{1 - \frac{3}{4}} + o(n) = 4 \cdot n + o(n) = O(n). \end{aligned}$$

Da $T(n) \geq n$ gilt, folgt sogar $T(n) = \Theta(n)$. □

Abschließend folgt ein nützlicher Satz zur Analyse von Divide-and-Conquer-Algorithmen.

Satz 1.10 (Master-Theorem) *Seien $a \geq 1$ und $b > 1$ Konstanten und $f : \mathbf{N_0} \to \mathbf{N_0}$ eine Funktion. Für die Funktion $T : \mathbf{N_0} \to \mathbf{N_0}$ gelte die Rekursion $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, wobei $\frac{n}{b}$ entweder $\lfloor \frac{n}{b} \rfloor$ oder $\lceil \frac{n}{b} \rceil$ bedeutet.*
Dann kann T asymptotisch wie folgt charakterisiert werden:

(i) Falls
$$f(n) = O(n^{\log_b a - \varepsilon})$$
für eine Konstante $\varepsilon > 0$, dann gilt $T(n) = \Theta(n^{\log_b a})$.

(ii) Falls
$$f(n) = \Theta(n^{\log_b a}),$$
dann gilt $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.

(iii) Falls
$$f(n) = \Omega(n^{\log_b a + \varepsilon})$$
für eine Konstante $\varepsilon > 0$ und falls $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ für eine Konstante $c < 1$ und n genügend groß, dann gilt $T(n) = \Theta(f(n))$.

□

Generell werden bei der Algorithmenanalyse einige Vereinfachung vorgenommen (die auch theoretisch abgesichert werden können).

- **Nichtganzzahligkeit**

  Die Laufzeitfunktion hat eigentlich ganzzahlige Argumente. Wir schreiben aber z. B. einfach $T(n) = 2 \cdot T(\frac{n}{2}) + n.$, da sich diese Änderung asymptotisch nicht auswirkt.

- **Ignorieren der Anfangsbedingung**

  Oft ist nur die Gestalt der Lösungsfunktion interessant und nicht die konkreten konstanten Faktoren, die in der O-Notation sowieso ignoriert werden.

- **Abschätzungen nur für große n**

  Wir schreiben einfach $5 + \log(n + 7) \leq n$, obwohl dies für kleine n nicht gilt.

- **Asymptotische Notation bereits in den Gleichungen**

  Man schreibt z. B. bereits $T(n) = \Theta(n) + T\left(\frac{n}{2}\right)$, um die Analyse zu vereinfachen.

- **Obere und untere Schranken**

  Manchmal ist es einfacher, Schranken für die Lösung einer Rekursionsgleichung zu finden, als sie exakt zu lösen. Oft ist die Rekursionsgleichung auch eigentlich eine Ungleichung.

1.7 Amortisationsanalyse

Man findet oft die Situation vor, dass nach Eintreten des Worst-Case für eine Operation dieser schlechteste Fall erst nach einer gewissen Anzahl weiterer Operationen wieder eintreten kann oder dass bei einer bestimmten Abfolge unterschiedlicher Operationen der Worst-Case nur selten eintreten kann.
Berücksichtigt man diese Effekte kommt man unter Umständen zu einer besseren Abschätzung für den Zeitbedarf einer Folge von Operationen als nur durch Summation der einzelnen Worst-Case-Zeiten. Eine solche amortisierte Komplexitätsangabe ist aber keine Average-Case-Angabe.

Beispiel Ein Binärzählers soll von 0 bis n zählen. Der Zähler benötigt also $\lceil \log(n-1) \rceil$ Stellen. Eine simple Worst-Case-Abschätzung für die Laufzeit liefert $T(n) = O(n \log n)$, da ein Hochzählen einen Übertrag über $\log n$ Stellen verursachen kann.
Analysiert man aber die zu ändernden Bits pro Zähloperation, ergibt sich folgendes Bild beim Hochzählen von 0 auf 16 (in Klammer die Anzahl der geänderten Bits).

```
 0:  0 0 0 0 0
 1:  0 0 0 0 1     (1)
 2:  0 0 0 1 0     (2)
 3:  0 0 0 1 1     (1)
 4:  0 0 1 0 0     (3)
 5:  0 0 1 0 1     (1)
 6:  0 0 1 1 0     (2)
 7:  0 0 1 1 1     (1)
 8:  0 1 0 0 0     (4)
 9:  0 1 0 0 1     (1)
10:  0 1 0 1 0     (2)
11:  0 1 0 1 1     (1)
12:  0 1 1 0 0     (3)
13:  0 1 1 0 1     (1)
14:  0 1 1 1 0     (2)
15.  0 1 1 1 1     (1)
16:  1 0 0 0 0     (5)
```

Eine genauere Abschätzung für die Laufzeit ist dann

$$\begin{aligned} T(n) &\leq n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots \\ &\leq 2 \cdot n. \end{aligned}$$

Die Worst-Case-Laufzeit für ein Inkrementieren des Zählers beträgt $\log n$, die amortisierte Komplexität aber nur $T(n)/n = 2$. □