Die hier definierten Komplexitätsfunktionen $t_A(n)$ und $s_A(n)$ erfassen den schlechtesten Fall, der bei der Länge n auftreten kann. Man bezeichnet diese $t_A(n)$ und $s_A(n)$ daher auch als worst case complexity.

Eine andere Möglichkeit, die Komplexität zu definieren, besteht in der Mittelung über die Komplexitätswerte für alle Eingaben der Länge n. Es sei E die Menge aller Eingaben und für jede natürliche Zahl i sei E_i die Menge der Eingaben der Länge i. Insbesondere gilt

$$E = E_0 \cup E_1 \cup E_2 \cup E_3 \cup E_4 \cup .... = \bigcup_{i=0}^{\infty} E_i .$$

Der mittlere Wert, die average case complexity, lautet dann:

$$\overline{t}_A(n) = \frac{1}{|E_n|} \sum_{w \in E_n} t_A(w) \qquad \text{und} \qquad \overline{s}_A(n) = \frac{1}{|E_n|} \sum_{w \in E_n} s_A(w) .$$

Es ist klar, dass der Aufwand an Zeit, an Platz usw., den ein Programm benötigt, besonders wichtig für die Informatik ist. Eine zentrale Aufgabe ist es daher, zu einer Funktion (bzw. zu einer Spezifikation) einen realisierenden Algorithmus (bzw. ein Programm) zu schreiben, dessen Komplexität möglichst gering ist.

Als Beispiel betrachten wir die Multiplikation zweier natürlicher Zahlen. Wir haben oben gesehen, dass es einen Algorithmus gibt, der die Multiplikation mit einer Zeitkomplexität von höchstens $8(n+1)^2$ realisiert, wobei n die Länge der eingegebenen Zahlen ist.

Gibt es einen schnelleren Algorithmus?

Oder kann man beweisen, dass es kein schnelleres Verfahren geben kann?

Vorbemerkung: Wenn man zwei natürliche Zahlen (ungleich Null), die jeweils k bzw. m Ziffern lang sind, miteinander multipliziert, so entsteht eine Zahl, die (k+m-1) oder (k+m) besitzt.

Da also das Ergebnis der Multiplikation nur so lang sein kann, wie die beiden Multiplikanten zusammen, könnte es eventuell sogar einen Algorithmus geben, der die Multiplikation proportional zu n durchführt.

Doch dies würde bedeuten, dass man die Multiplikation bis auf einen konstanten Faktor genau so schnell ausführen könnte wie die Addition.

Das glaubt eigentlich niemand. Dennoch ist es bisher keinem gelungen zu beweisen, dass die Multiplikation deutlich mehr Zeit als die Addition erfordert.

Nun wollen wir ein Verfahren vorstellen, welches deutlich schneller als mit der Zeitkomplexität $4(n+1)^2$ multipliziert.

Gegeben seien also zwei k-stellige natürliche Zahlen x und y; die Länge k sei eine gerade Zahl. Setze p=k/2. Dann lassen sich x und y schreiben in der Form:

$$\mathbf{x = A \cdot 2^p + B} \quad \text{und} \quad \mathbf{y = C \cdot 2^p + D},$$

wobei A, B, C, D mindestens 0 und echt kleiner als 2^p sind. Dann gilt

$$\mathbf{x \cdot y = (A \cdot 2^p + B) \cdot (C \cdot 2^p + D) = A \cdot B \cdot 2^{2p} + (A \cdot D + B \cdot C) \cdot 2^p + B \cdot D}.$$

Man kann also die Multiplikation zweier k-stelliger Zahlen durchführen, indem man sie auf 4 Multiplikationen von halber Länge k/2 zurückführt. Dies ergibt jedoch wiederum eine quadratische Zeitkomplexität. Kommt man vielleicht mit weniger als vier Multiplikationen halber Länge aus?

Es gilt: $(A{\cdot}D + B{\cdot}C) = (A-B){\cdot}(D-C) + A{\cdot}C + B{\cdot}D$,
wie man durch Ausrechnen leicht nachprüft.

Somit erhalten wir aus obiger Gleichung:

$$x \cdot y = A{\cdot}C{\cdot}2^{2p} + (A{\cdot}D + B{\cdot}C){\cdot}2^p + B{\cdot}D$$
$$= A{\cdot}C{\cdot}2^{2p} + ((A-B){\cdot}(D-C) + A{\cdot}C + B{\cdot}D){\cdot}2^p + B{\cdot}D$$

wobei A, B, C, D mindestens 0 und echt kleiner als 2^p sind.

Nun haben wir es geschafft: Auf der rechten Seite stehen nur noch drei verschiedene Multiplikationen, nämlich:

$A{\cdot}C$, $B{\cdot}D$ und **$(A-B){\cdot}(D-C)$**

Beachte: Die Multiplikation mit 2^p bzw. 2^{2p} ist keine echte Multiplikation, sondern nur ein Anhängen von p bzw. 2p Nullen.

Somit haben wir die Multiplikation zweier k-stelliger Zahlen auf drei Multiplikationen von k/2-stelligen Zahlen zurückgeführt. Der Preis, den wir dafür bezahlen müssen, sind zwei zusätzliche Subtraktionen zweier k/2-stelliger Zahlen und zwei zusätzliche Additionen zweier k-stelliger Zahlen. Da aber die Zeit für die Addition und die Subtraktion proportional zur Länge der Zahlen ist, müssten wir dennoch schneller fertig werden. Wir prüfen dies nun nach.

Wenn t(k) die Anzahl der durchzuführenden Operationen für die Multiplikation zweier k-stelliger Zahlen nach diesem Verfahren ist, dann erhalten wir also folgende Gleichung:

$$t(k) = \underbrace{3{\cdot}t(k/2)}_{\text{Rekursion}} + \underbrace{4{\cdot}k}_{\text{4 Additionen k stelliger Zahlen}} + \underbrace{2{\cdot}(k/2)}_{\text{2 Subtraktionen k/2 stelliger Zahlen}} \quad \text{mit} \quad \underbrace{t(1)=1}_{\text{Multiplikation zweier Ziffern}}$$

Wie lautet die Lösung dieser Gleichung?

Probieren wir es aus. Ersetzen von t(k/2), t(k/4) usw. entsprechend der Rekursionsformel $t(k) = 3 \cdot t(k/2) + 5 \cdot k$ ergibt:

$$
\begin{aligned}
t(k) &= 3 \cdot t(k/2) + 5 \cdot k \\
&= 3 \cdot (3 \cdot t(k/4) + 5 \cdot k/2\ ) + 5 \cdot k \\
&= 3 \cdot 3 \cdot t(k/4) + 3 \cdot 5 \cdot k/2 + 5 \cdot k \\
&= 3 \cdot 3 \cdot t(k/4) + 5 \cdot k \cdot (1 + 3/2) \\
&= 3 \cdot 3 \cdot (3 \cdot t(k/8) + 5 \cdot k/4) + 5 \cdot k \cdot (1 + 3/2) \\
&= 3 \cdot 3 \cdot 3 \cdot t(k/8) + 5 \cdot k \cdot (1 + 3/2 + 9/4) \\
&= 3 \cdot 3 \cdot 3 \cdot (3 \cdot t(k/16) + 5 \cdot k/8) + 5 \cdot k \cdot (1 + 3/2 + 9/4) \\
&= 3 \cdot 3 \cdot 3 \cdot 3 \cdot t(k/16) + 5 \cdot k \cdot (1 + 3/2 + 9/4 + 27/8) \\
&= \ldots
\end{aligned}
$$

Die allgemeine Form nach i Schritten lautet offensichtlich:

$$
\begin{aligned}
t(k) &= 3^i \cdot t(k/2^i) + 5 \cdot k \cdot (1 + 3/2 + 9/4 + \ldots + 3^{i-1}/2^{i-1}) \\
&= 3^i \cdot t(k/2^i) + 10 \cdot k \cdot ((3/2)^i - 1)
\end{aligned}
$$

Beachte die geometrische Reihe

$$1 + a + a^2 + a^3 + \ldots + a^m = \frac{a^{m+1} - 1}{a - 1}$$

In unserem Fall ist $a = 3/2$.

Diese Ersetzungen kann man vornehmen, bis $k = 2^i$ geworden ist, also bis $i = \log(k)$. Dann ist $t(k/2^{\log(k)}) = t(1) = 1$. Wir setzen dies ein und erhalten:

$$
\begin{aligned}
\mathbf{t(k)} &= 3^{\log(k)} \cdot t(k/2^{\log(k)}) + 10 \cdot k \cdot ((3/2)^{\log(k)} - 1) \\
&= k^{\log(3)} + 10 \cdot k \cdot (k^{\log(1,5)} - 1) \\
&= 11 \cdot k^{\log(3)} - 10 \cdot k \approx \mathbf{11 \cdot k^{1,585} - 10 \cdot k} \in \mathbf{O(k^{1,585})}
\end{aligned}
$$

Beachte hierbei die Formeln:
log ist hier der Logarithmus zur Basis 2,

$a^{\log(b)} = b^{\log(a)}$ und

$k \cdot k^{\log(1,5)} = k^{1+\log(1,5)} = k^{\log(2)+\log(1,5)} = k^{\log(2 \cdot 1,5)} = k^{\log(3)}$

Satz 1.2:

Die Multiplikation zweier k-stelliger Zahlen lässt sich in proportional zu $k^{1,585}$ Schritten durchführen.

(Die Schulmethode benötigt k^2 Schritte, siehe nächste Folie.)

Geht es noch schneller? Ja. Der beste derzeit bekannte Algorithmus, der Algorithmus nach Schönhage und Strassen (1971), der sich allerdings nur für riesige Zahlen eignet, benötigt

$O(k \cdot \log(k) \cdot \log(\log(k)))$ Schritte.

Dies ist schon relativ dicht an der Größenordnung $O(k)$, so dass man vielleicht eines Tages doch einen Algorithmus finden wird, der die Multiplikation in $O(k)$ Schritten - und damit ungefähr so schnell wie eine Addition - durchführt?

Anmerkung: Ab wann lohnt sich dieses rekursive Verfahren? Hierfür müssen wir wissen, wie aufwändig eine normale Multiplikation in Wahrheit ist. Der Algorithmus aus Beispiel 1.5 benötigt höchstens $8 \cdot (k+1)^2$ Schritte (vgl. Folie 71; setzen Sie dort log(a)=log(b)=k und rechnen nochmals nach). Doch bei einer genaueren Untersuchung erkennt man, dass jener Algorithmus im Wesentlichen in k^2 Schritten arbeitet; denn nur die Anweisung z:=z+x kostet "richtig" Zeit (x:=x+x bedeutet: hänge eine 0 an x an, bei y:=y div 2 wird nur die letzte 0 gestrichen usw.).

Dann läuft die Frage, ab wann sich die rekursive Methode lohnt, auf die Frage hinaus, ab welchem k gilt: $11 \cdot k^{1,585} - 10 \cdot k < k^2$? Dies trifft leider erst für Werte von k zu, die größer als 290 sind. Es müssen also schon spezielle Probleme sein, bei denen es sich lohnt, diese Methode einzusetzen.
Dies war aber nur eine überschlägige Berechnung. Eine genaue Analyse müsste *alle* auftretenden Operationen einbeziehen, z.B. auch die Funktionsaufrufe und die Speicherung von aktuellen Parametern.

Auf den letzten Folien trat das Symbol O(...) auf, das Sie aus der "Einführung in die Informatik I" bereits kennen. Dies ist das so genannte *Landau-Symbol* (nach dem deutschen Mathematiker Edmund Landau, 1877-1938). Es beschreibt die Größenordnung einer Funktion. Hierbei werden multiplikative und additive Konstanten vernachlässigt und nur der Term in Abhängigkeit von k, der für $k \to \infty$ alles andere überwiegt, berücksichtigt.

Formal gesehen handelt es sich bei O(f) um die Definition einer Funktionenklasse in Abhängigkeit von einer Funktion f. In O(f) sind alle Funktionen über den reellen Zahlen enthalten, die "schließlich von f dominiert" werden.

Um die Größenordnung von Funktionen zu beschreiben, verwendet man folgende fünf Klassen O, o, Ω, ω und Θ:

Definition: "groß O", "klein O", "groß Omega", klein Omega", "Theta"

Es sei f: $\mathbb{R}^+ \to \mathbb{R}^+$ eine Funktion über den positiven reellen Zahlen $\mathbb{R}^+ \subset \mathbb{R}$ (oft wird diese Definition auf die natürlichen Zahlen eingeschränkt, also auf Funktionen f: $\mathbb{N} \to \mathbb{N}$; unten muss dann nur g: $\mathbb{R}^+ \to \mathbb{R}^+$ durch g: $\mathbb{N} \to \mathbb{N}$ ersetzt werden).

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

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

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

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

Θ(f) $:= \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0:$
$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$.

Erläuterungen: Es sei f: $\mathbb{R}^+ \to \mathbb{R}^+$

g liegt in **O(f)**, wenn g höchstens so stark wächst wie f, wobei Konstanten nicht zählen. Statt *g ist höchstens von der Größenordnung f*, sagen wir, *g ist groß-O von f,* und meinen damit, dass $g \in O(f)$ ist.

Wenn eine Funktion g zusätzlich echt kleiner für gleiche Werte von n wächst als f, wenn also zusätzlich gilt:

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

dann sagen wir, *g ist von echt kleinerer Größenordnung als f* oder *g ist klein-o von f,* und meinen damit, dass $g \in$ **o(f)** ist.

Erläuterungen (Fortsetzung): Es sei f: $\mathbb{R}^+ \rightarrow \mathbb{R}^+$

Die Klassen Ω und ω (groß Omega und klein Omega) bilden die "Umkehrungen" der Klasse O und o. Eine Funktion g liegt genau dann in **Ω(f)** bzw. in **ω(f)**, wenn f in O(g) bzw. in o(g) liegt.

In Ω(f) liegen also die Funktionen, die mindestens so stark wachsen wie f, und in ω(f) liegen die Funktionen, die zusätzlich bzgl. n echt stärker wachsen.

In der Klasse **Θ(f)** liegen die Funktionen, die sich bis auf Konstanten im Wachstum wie f verhalten. Wenn $g \in \Theta(f)$ ist, dann sagen wir, *g ist von der gleichen Größenordnung wie f* oder *g ist Theta von f*. Da in diesem Fall g in O(f) und f in O(g) liegen müssen, folgt unmittelbar die Gleichheit $\Theta(f) = O(f) \cap \Omega(f)$.

Schreibweisen: Es sei f: $\mathbb{R}^+ \rightarrow \mathbb{R}^+$
Statt $g \in O(f)$ schreibt man manchmal auch $g=O(f)$, um auszudrücken, dass g höchstens von der Größenordnung f ist. Das gleiche gilt für die anderen vier Klassen.

Anstelle der Funktionen gibt man meist nur deren formelmäßige Darstellung an. Beispiel: Statt

O(f) für die Funktion f mit $f(n) = n^2$ für alle $n \in \mathbb{N}$

schreibt man einfach $O(n^2)$.

Man schreibt auch "Ordnungs-Gleichungen", die aber nur von links nach rechts gelesen werden dürfen, z.B.:

$3 \cdot n^3 + 12 \cdot n^2 + 8 \cdot n \cdot \log(n) + 6/n = 3 \cdot n^3 + O(n^2) = O(n^3)$.
Korrekt müsste man hierfür beispielsweise schreiben:

$3 \cdot n^3 + 12 \cdot n^2 + 8 \cdot n \cdot \log(n) + 6/n \in O(3 \cdot n^3) \cup O(n^2+n \cdot \log(n)) = O(n^3)$.

Einige Klassen:

O(1) ist die Klasse der Funktionen, die höchstens wie ein Vielfaches der konstanten Funktion *f(n) = 1 für alle* $n \in \mathbb{N}$ wachsen. Somit gehören alle konstanten Funktionen, aber auch Funktionen wie sin(n), cos(n), 1/n, $1/n^2$ oder 1/log(n) zu O(1).

Gibt es eine Klasse von Funktionen, die nur sehr schwach wachsen, also deutlich langsamer als f(n)=n? Aus der Schule und dem ersten Semester kennen Sie den Logarithmus. Noch wesentlich schwächer wächst der "iterierte Logarithmus":

log*(n) = 0, für n=0 und 1,

$$\log^*(n) = \mathrm{Min}\{k \mid \underbrace{\log(\log(\log(\ldots \log(n) \ldots)))}_{\text{k ineinander geschachtelte Logarithmen}} < 2\} \text{ für } n>1.$$

Aufgabe für Sie: Untersuchen Sie diese Funktion log*.

O(n) = Klasse der höchstens linear wachsenden Funktionen:

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

Man beachte, dass hierin auch alle Funktionen der Form $g(n) = c_1 \cdot n + c_2$ (für zwei positive Konstanten c_1 und c_2) enthalten sind, weil für $n \geq 1$ gilt: $g(n) = c_1 \cdot n + c_2 \leq (c_1 + c_2) \cdot n$.

Wenn g in O(n) liegt, so sagt man auch, g sei *höchstens linear*.

Zu den höchstens linear wachsenden Funktionen gehört (wegen log(x) < x für alle x>0) auch der Logarithmus. Es gilt daher: $\log(n) \in O(n)$. Aber auch für die Potenzen des Logarithmus gilt $\log^m(n) \in O(n)$ für alle natürlichen Zahlen m. Hierfür beachte:

Für jedes $m \geq 1$ gilt ab einem hinreichend großen n:

$\log(n) < n^{\frac{1}{m}}$ wegen $n < 2^{\sqrt[m]{n}}$; folglich $\log^m(n) < n$.

$\Omega(n)$ = Klasse der mindestens linear wachsenden Funktionen:
$\Omega(n)$:= $\{g\colon \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+\ \exists\, n_0 \in \mathbb{N}\ \forall n \geq n_0\colon n \leq c \cdot g(n)\}$.
Auch hierin sind alle Funktionen der Form $g(n) = c_1 \cdot n + c_2$ (für zwei positive Konstanten c_1 und c_2) enthalten, weil für $n \geq 1$ gilt:
$n \leq (1/c_1) \cdot g(n) = n + (c_2/c_1)$.
Wenn g in $\Omega(n)$ liegt, so sagt man auch, g sei *mindestens linear*.

$\Theta(n)$ = Klasse der linear wachsenden Funktionen:
$\Theta(n)$:= $\{g\colon \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+\ \exists\, n_0 \in \mathbb{N}\ \forall n \geq n_0\colon c_1 \cdot n \leq g(n) \leq c_2 \cdot n\}$.
Wegen $\Theta(f) = O(f) \cap \Omega(f)$ für alle Funktionen f gehören zu $\Theta(n)$ insbesondere alle Funktionen der Form $g(n) = c_1 \cdot n + c_2$ (für zwei positive Konstanten c_1 und c_2), aber auch Funktionen wie
$g(n) = n + \log^m(n) + 1/n \in \Theta(n)$ usw.

$O(n^2)$ = Klasse der höchstens quadratisch wachsenden Funktionen.
$O(n^2)$:= $\{g\colon \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+\ \exists\, n_0 \in \mathbb{N}\ \forall n \geq n_0\colon g(n) \leq c \cdot n^2\}$.
Hierin sind alle Funktionen der Form $g(n) = c_1 \cdot n^2 + c_2 \cdot n + c_3$ (für Konstanten c_1, c_2 und c_3) enthalten, weil ab einem gewissen $n \geq 1$ dann gilt: $g(n) = c_1 \cdot n^2 + c_2 \cdot n + c_3 \leq (c_1+1) \cdot n^2$.
Wenn g in O(n) liegt, so sagt man, g wächst *höchstens quadratisch*.
$\Omega(n^2)$ ist die Klasse der mindestens quadratisch wachsenden Funktionen.

Für jede natürliche Zahl k ist **$O(n^k)$** die Klasse der höchstens wie n^k wachsenden Funktionen; $O(n^k)$ umfasst insbesondere alle Polynome vom Grad k.

Für jede natürliche Zahl k ist **$o(n^k)$** die Klasse der echt schwächer als n^k wachsenden Funktionen, z.B. Funktionen wie n^{k-1} oder $n^k/\log(n)$ oder n^{k-d} für jede reelle Zahl d>0.

In der Praxis betrachtet man in der Regel folgende Funktionen:

$O(1)$: konstante Funktionen.

$O(\log n)$: höchstens logarithmisch wachsende Funktionen; wenn die Länge einer Darstellung wichtig ist, kommt oft der Logarithmus ins Spiel.

$O(n^{1/k})$: höchstens mit einer k-ten Wurzel wachsende Funktionen.

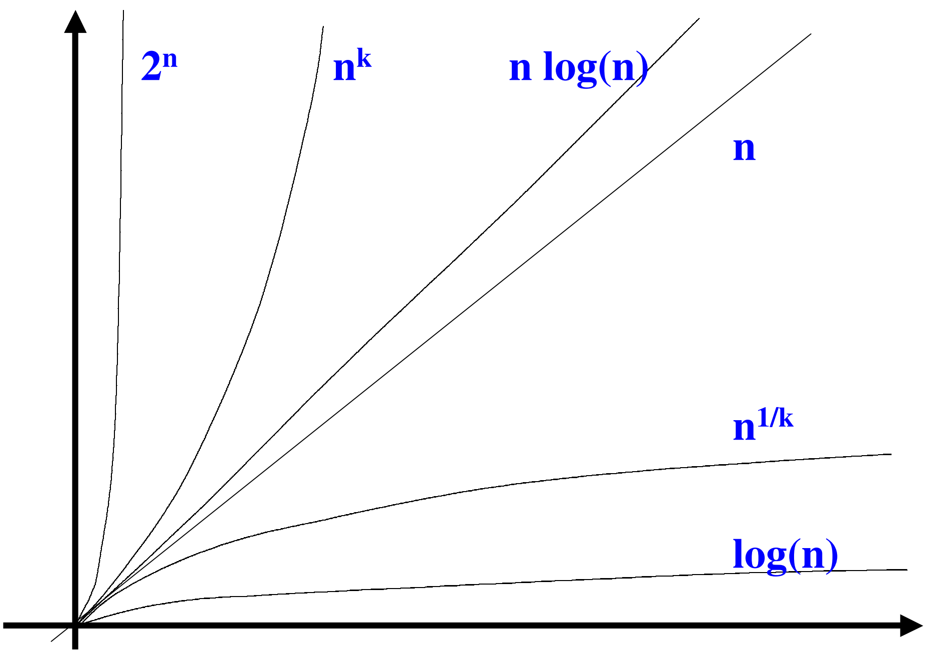
$O(n)$: lineare Funktionen.

$O(n \log(n))$: Das sind Funktionen, die "fast unmerklich" stärker als linear wachsen.

$O(n^2)$: höchstens quadratische Funktionen.

$O(n^k)$: höchstens polynomiell vom Grad k wachsende Funktionen.

$O(2^n)$: höchstens exponentiell (zur Basis 2) wachsende Funktionen.



Hilfssatz: Es gelten folgende Aussagen (der Beweis ist einfach):

$o(f) \subset O(f)$, $\omega(f) \subset \Omega(f)$, $\Theta(f) = O(f) \cap \Omega(f)$.

Für alle $g \in O(f)$ gelten $O(f+g) = O(f)$ und $o(f+g) = o(f)$.

Für alle $g \in \Omega(f)$ gelten $\Omega(f+g) = \Omega(g)$ und $\omega(f+g) = \omega(g)$.

Für alle $g \in \Theta(f)$ gilt $\Theta(f+g) = \Theta(f) = \Theta(g)$.

Aufgabe: Untersuchen Sie, ob folgende Formeln gelten:

$\omega(f) \cup O(f) = \Omega(f)$?

$O(f) - o(f) = \Theta(f)$?

$o(f) \cap \omega(f) = \emptyset$?

In der Regel sind Funktionen durch diese Klassen nicht vergleichbar. Zum Beispiel gilt für die Funktionen

$$f(n) = \begin{cases} 1, \text{ für gerades } n \\ n, \text{ für ungerades } n \end{cases} \qquad g(n) = \begin{cases} n, \text{ für gerades } n \\ 1, \text{ für ungerades } n \end{cases}$$

weder $f \in O(g)$ noch $g \in O(f)$.

Wir werden vor allem mit den Klassen O und Θ bei der Untersuchung des Zeit- und Platzaufwands rechnen. Oft interessiert uns nämlich nur die Größenordnung der Komplexität und nicht der genaue Wert von Konstanten.

Historischer Hinweis:

Seit es (mathematische) Verfahren gibt, möchte man diese so schnell wie möglich ausführen. Beispiele sind die Lösung algebraischer (etwa: quadratischer) Gleichungen und die Berechnung von Wurzeln sowie die Lösung linearer Gleichungssysteme (Gaußsches Eliminationsverfahren). Bei vielen Verfahren kannte man ziemlich exakt die Größenordnung des Zeitaufwands in Abhängigkeit von den Parametern. Mit der Entwicklung von Großrechenanlagen in den 1960er Jahren analysierte man sehr genau die Komplexität von Algorithmen (abhängig vom jeweils benutzten Maschinenmodell). Hieraus entstand etwa Mitte der 1960er Jahre die Komplexitätstheorie.

Heute ist es üblich, mit jedem Algorithmus auch seine Komplexität zu berechnen, meist in Abhängigkeit zur Länge der Eingabe oder zur Zahl der bearbeiteten Objekte. Das Problem sind hierbei vor allem die *unteren* Schranken, also der Nachweis, dass ein Problem zu seiner Lösung mindestens einen gewissen Aufwand erfordert. Es gibt nur wenige Probleme, für die man nicht-triviale untere Schranken kennt. *Obere* Schranken findet man dagegen meist leicht, da jeder Algorithmus eine solche obere Schranke für das von ihm realisierte Problem liefert.

1.5 Gegenläufigkeiten

Wenn es für ein Problem eine algorithmische Lösung gibt, so gibt es grundsätzlich auch unendlich viele Lösungsalgorithmen. Unter dieser Vielzahl möchte man einen möglichst guten finden. Dabei ist die "Zielfunktion", mit der die Algorithmen bewertet werden, wichtig. Sucht man beispielsweise einen Algorithmus, der möglichst wenig Speicherplatz verbraucht, so wird man einen anderen Algorithmus erhalten, als wenn man ein möglichst schnell arbeitendes Verfahren haben möchte.

Unterschiedliche Ziele sind in der Regel nicht gleichzeitig zu optimieren. Man spricht dann von Gegenläufigkeiten (engl.: Trade-Offs). Diese treten oft als "Zeit gegen Platz" (time versus space) auf.

Typische Situation: Ein Problem kann wesentlich einfacher gelöst werden, wenn man zuvor Hilfsberechnungen (z.B. eine Entfernungstabelle für gewisse "markante" Stellen bei der Berechnung kürzester Wege) durchführt und deren Ergebnis in arrays speichert; hat man keinen zusätzlichen Speicherplatz zur Verfügung, so muss man andererseits länger rechnen.

Wir geben hierfür keine konkreten Fragestellungen an, da wir bei Such- und Sortierverfahren mehrere solcher Beispiele kennen lernen werden. Es sollte hier nur auf dieses ständig wiederkehrende Problem hingewiesen werden.