Ergänzungen zu: Komplexität von Algorithmen

Die Idee: ***Komplexität*** ist der Aufwand an Zeit $t_A(w)$ oder an Platz $s_A(w)$, den ein Algorithmus A bei der Eingabe w benötigt, um seine Berechnung durchzuführen. (t kommt von "time complexity" und s von "space complexity".)

Genauer: Sei A ein Algorithmus A. Eine Berechnung für die Eingabe w ist eine Folge von elementaren Anweisungen (leere Anweisung, Wertzuweisung) und Abfragen.

$t_A(w)$ = Anzahl der elementaren Anweisungen und Abfragen, die A bei Eingabe w durchläuft, bis er anhält,

$s_A(w)$ = Anzahl der Speicherplätze, auf die A bei Eingabe von w zugreift, bis er anhält.

In der Praxis wichtig: nicht die Eingabefolge w, sondern nur ihre Länge. Daher definiert man:

Etwas konkretere Idee:
Es sei A ein Algorithmus, w sind Eingabefolgen.
$t_A(n) =$ Max $\{t_A(w) \mid$ die Länge von w ist n$\}$,
$s_A(n) =$ Max $\{s_A(w) \mid$ die Länge von w ist n$\}$.

Diese Definition setzt voraus, dass der Algorithmus A für alle Eingaben anhält; denn falls der Algorithmus A für eine Eingabefolge w der Länge n *nicht* anhält, dann wären $t_A(n)$ [und eventuell auch $s_A(n)$] unendlich groß und somit für Anwendungen uninteressant.

Bezeichnung:

Es sei A ein Algorithmus, der für alle Eingaben w anhält.
Dann definieren wir:

$t_A(n)$ = Max $\{t_A(w) \mid$ die Länge von w ist n$\}$,

$s_A(n)$ = Max $\{s_A(w) \mid$ die Länge von w ist n$\}$.

t_A und s_A sind also Abbildungen von $\mathbb{N}_0$ nach $\mathbb{N}_0$.

Hinweis: Man muss eigentlich ein Maschinenmodell zugrunde legen, z.B. eine Grammatik, eine Turing- oder eine Registermaschine. Siehe später.

Beispiel: Multiplikation zweier Zahlen

```
x, y, z: Natural;
Get (x, y);
z:=0;
while y > 0 loop
     if (y mod 2 = 0) then
          y:=y div 2;
          x:=x+x;
     else
          y:=y-1;
          z:=z+x;
     end if;
end loop;
Put(z);
```

Dieses Programmstück berechnet die Multiplikation zweier natürlicher Zahlen. Wie groß ist der Aufwand?

```
Get (x, y);
z:=0;
while y > 0 loop
     if (y mod 2 = 0) then
          y:=y div 2;
          x:=x+x;
     else
          y:=y-1;
          z:=z+x;
     end if;
end loop;
Put(z);
```

2 Schritte

1 Schritt

1 Schritt

m+1 mal

1 Schritt

2 Schritte

oder:
2 Schritte

m mal

1 Schritt

Zusammen: 4m + 5 Schritte. Aber was ist m?

Die Schleife wird durchlaufen, bis y = 0 ist. Spätestens in jedem zweiten Schritt wird y halbiert. Also wird die Schleife mindestens log(y) mal und höchstens 2 log(y) mal durchlaufen. Also gilt $\log(y) \leq m \leq 2 \log(y)$.

Die Eingabewerte seien die natürlichen Zahlen a und b. Die Eingabelänge ist dann n = log(a) + log(b) + 2. (Das "+2" kommt daher, dass man ein Trennzeichen zwischen a und b und ein Zeichen für das Ende der Eingabe benötigt.)
m kann daher durch n nach oben abgeschätzt werden.

Für unseren Multiplikationsalgorithmus A gilt also:
$\mathbf{t_A(n)} = \text{Max}\ \{t_A(w) \mid \text{die Länge von w ist n}\} \leq 8n + 5.$

Man braucht drei Speicherplätze für x, y und z, folglich gilt:
$\mathbf{s_A(n)} = \text{Max}\ \{s_A(w) \mid \text{die Länge von w ist n}\} = 3.$

Aber: Ist dies eine "richtige" Aufwandsberechnung?
Warum geht zum Beispiel nur die Länge der zweiten Zahl b, die y zugeordnet wird, und nicht auch die der ersten Zahl a, die in x steht, in die Komplexität ein??

..... weil wir jede Wertzuweisung und jede Abfrage so behandeln, als ob sie **in einem Schritt** ausgeführt werden kann.

In der Praxis würde die Anweisung x:=x+x nicht einen Schritt, sondern so viele Schritte benötigen, wie der Wert von x lang ist (ungefähr log(x)); denn so lange dauert die ziffernweise Addition, wenn man sie wie in der Schule erlernt durchführt.

Falls wir nichts über die Zahldarstellung wissen, müssten wir wie folgt abschätzen:

```
Get (x, y);
z:=0;
while y > 0 loop
     if (y mod 2 = 0) then
          y:=y div 2;
          x:=x+x;
     else
          y:=y-1;
          z:=z+x;
     end if;
end loop;
Put(z);
```

log(a) + log(b) = n Schritte

1 Schritt

≤ log(b) Schritte ≤ 2·log(b)+1 mal

≤ log(b) Schritte

≤ log(b) Schritte

log(a)+log(b) Schritte

log(b) Schritte

log(a)+log(b) Schritte

bis zu 2·log(b) mal

log(a) + log(b) = n Schritte

Zusammen: ≤ (2 log(b)+1) (4 log(b)+log(a))+2n+1 ≤ ca. $8(n+1)^2$ Schritte.

Stellt man die Zahlen zur Basis 2 dar, dann kann man die Komplexität des Algorithmus verringern. Die Operationen benötigen dann nur noch folgenden Aufwand:

y mod 2 = 0	1 Einheit: Prüfe, ob letzte Ziffer von y 0 ist.
y := y div 2	1 Einheit: Streiche die letzte 0 von y.
x := x + x	1 Einheit: Füge eine '0' an x an.
y := y -1	Da y hier ungerade ist, muss nur die letzte Ziffer (die 1 sein muss) gestrichen werden.
z := z + x	Dies kostet bis zu log(a)+log(b) ≈ n Schritte.

Was ist mit der Abfrage $y > 0$?

y > 0 Kann bis zu log(b) Schritte kosten.

Wirklich? Nicht, wenn man z.B. etwas über die Länge des aktuellen Werts von y weiß.

Man setzt eine Variable L beim Einlesen von b auf den Wert L=0, falls die Eingabe nur die Ziffer 0 war, und anderenfalls auf L = 111...110, wobei L genau so viele Einsen erhält, wie b Stellen hat. Die Bedingung y>0 ersetzt man dann durch (Erste Ziffer von L ist 1) **or else** (letzte Ziffer von y ist 1). In der Schleife wird nach der Anweisung y:=y div 2 die Anweisung L:="L ohne die erste Ziffer" eingefügt.

Es sei nun b1 = (Erste Ziffer von L ist 1)
und b2 = (letzte Ziffer von y ist 1)

```
Get (x, y);
Setze L wie angegeben;
z:=0;
while b1 else or b2 loop
     if (y mod 2 = 0) then
        y:=y div 2;
        L:=L ohne erste Ziffer;
        x:=x+x;
     else
        y:=y-1;
        z:=z+x;
     end if;
end loop;
Put(z);
```

n Schritte für die Eingabe

$\leq \log(b)+1$ Schritte

1 Schritt

$\leq 2 \cdot \log(b)+2$ Schritte

$\leq 2 \cdot \log(b)$ Schritte

$\leq \log(b)$ Schritte

$\leq \log(b)$ Schritte

$\leq \log(b)$ Schritte

$\leq \log(b) \cdot (\log(a)+\log(b))$

$(\approx n \cdot \log(b)$ Schritte)

$\log(a) + \log(b) \approx n$ Schritte

Zusammen: $\leq 9 \cdot \log(b) + n \cdot (\log(b)+2) + 2n+4 \leq n^2 + 11 \cdot n$ Schritte.

Bezeichnung:
Berechnet man die Komplexität $t_A(n)$ bzw. $s_A(n)$ so, dass jede elementare Anweisung und jede Abfrage genau einen Schritt dauern, bzw. dass jede Zahl genau einen Speicherplatz belegt, so spricht man von der **uniformen Komplexität**.
Berechnet man die Komplexitäten so, dass jede Operation so viel Zeit kostet, wie die zeichenweise Ausführung erfordert, bzw. dass Werte so viel Platz belegen, wie sie Zeichen haben, so spricht man von der **logarithmischen Komplexität**.

In der Praxis berechnet man stets zunächst die uniforme Komplexität: Sie gibt meist eine hinreichend genaue Orientierung über den zu erwartenden Aufwand. Erst für eine genaue Abschätzung bzw. bei der Programmierung betrachtet man die logarithmische Komplexität, die den tatsächlichen Aufwand wesentlich genauer wiedergibt.

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:

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

Im Allgemeinen ist $E = \Sigma^*$ für ein geeignetes Alphabet Σ, und die E_i sind die Teilmengen aus Wörtern der Länge i. Dann lauten die definierenden Formeln für die average case complexity:

$$\bar{t}_A(n) = \frac{1}{|\Sigma|^n} \sum_{\substack{w \in \Sigma^* \\ |w| = n}} t_A(w) \qquad \text{und} \qquad \bar{s}_A(n) = \frac{1}{|\Sigma|^n} \sum_{\substack{w \in \Sigma^* \\ |w| = n}} s_A(w).$$

Eine "bessere" Definition der Komplexität, wäre daher:

Es sei A ein Algorithmus, der für alle Eingaben anhält, und w eine Eingabe. Eine Berechnung $ß_{w,1}, ß_{w,2}, ß_{w,3}, ..., ß_{w,m}$ von A bei Eingabe w ist die Folge aus m Bedingungen und elementaren Anweisungen, die A bei Eingabe von w bis zum Anhalten durchläuft.

Jede Bedingung bzw. elementare Anweisung ß möge in genau $\zeta(ß)$ Zeiteinheiten ausgeführt werden. Dann definiere für alle w

$$t'_A(w) = \sum_{i=1}^{m} \zeta(ß_{w,i})$$

als den Zeitaufwand von A für Eingabewörter w.

Hinweis: In ζ ist die Maschine enthalten, die den Algorithmus abarbeitet.

Die worst case Zeitkomplexität $\mathbf{t_A} : \mathbb{N}_0 \to \mathbb{N}_0$ von A ist dann:

$\mathbf{t_A(n)}$ = Max $\{t'_A(w) \mid$ die Länge von w ist n$\}$

und die average case Zeitkomplexität lautet:

$$\bar{t}_A(n) = \frac{1}{|\Sigma|^n} \sum_{\substack{w \in \Sigma^* \\ |w| = n}} t'_A(w)$$

Auf die Platzkomplexität, die etwas schwieriger zu behandeln ist, kommen wir später zurück, siehe handschriftliche Folien in Kap. 1 und Kap. 7.

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 $n^2 + 11 \cdot n$ realisiert, mit n = Länge der eingegebenen Zahlen.

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 $O(n^2)$ 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

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

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:

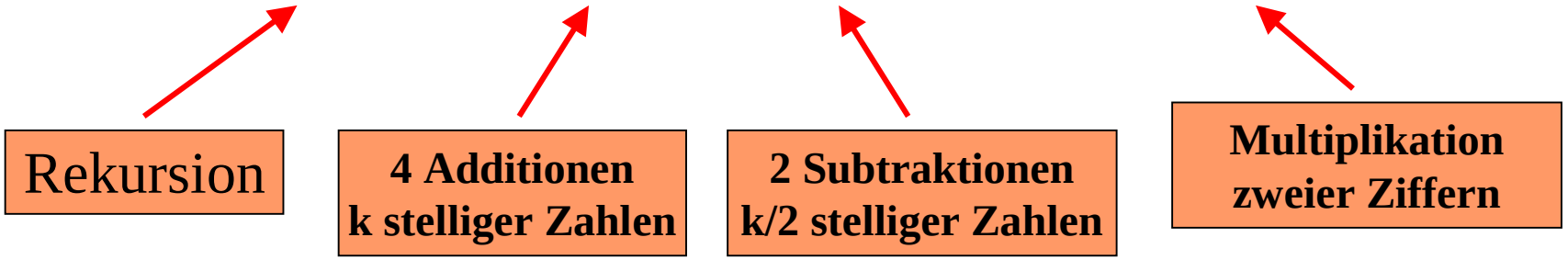
$A \cdot C$, **$B \cdot D$** und **$(A-B) \cdot (D-C)$**

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:

$$t(k) = 3 \cdot t(k/2) + 4 \cdot k + 2 \cdot (k/2) \quad \text{mit} \quad t(1)=1$$



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·t(k/2) + 5·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:

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

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:

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

(Unser Verfahren, aber auch die Schulmethode benötigen k^2 Schritte.)

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?
Der Algorithmus aus unserem Beispiel benötigte höchstens $k^2 + 11 \cdot k$ Schritte.

Dann läuft die Frage, ab wann sich die rekursive Methode lohnt, auf die Frage hinaus, ab welchem k gilt: $11 \cdot k^{1,585} - 10 \cdot k < k^2 + 11 \cdot k$? Dies trifft leider erst für Werte von k zu, die größer als 200 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).

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

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

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

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

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

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

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:

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

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: $\mathbb{R}^+ \to \mathbb{R}^+$

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: $\mathbb{R}^+ \to \mathbb{R}^+$

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:

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

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.

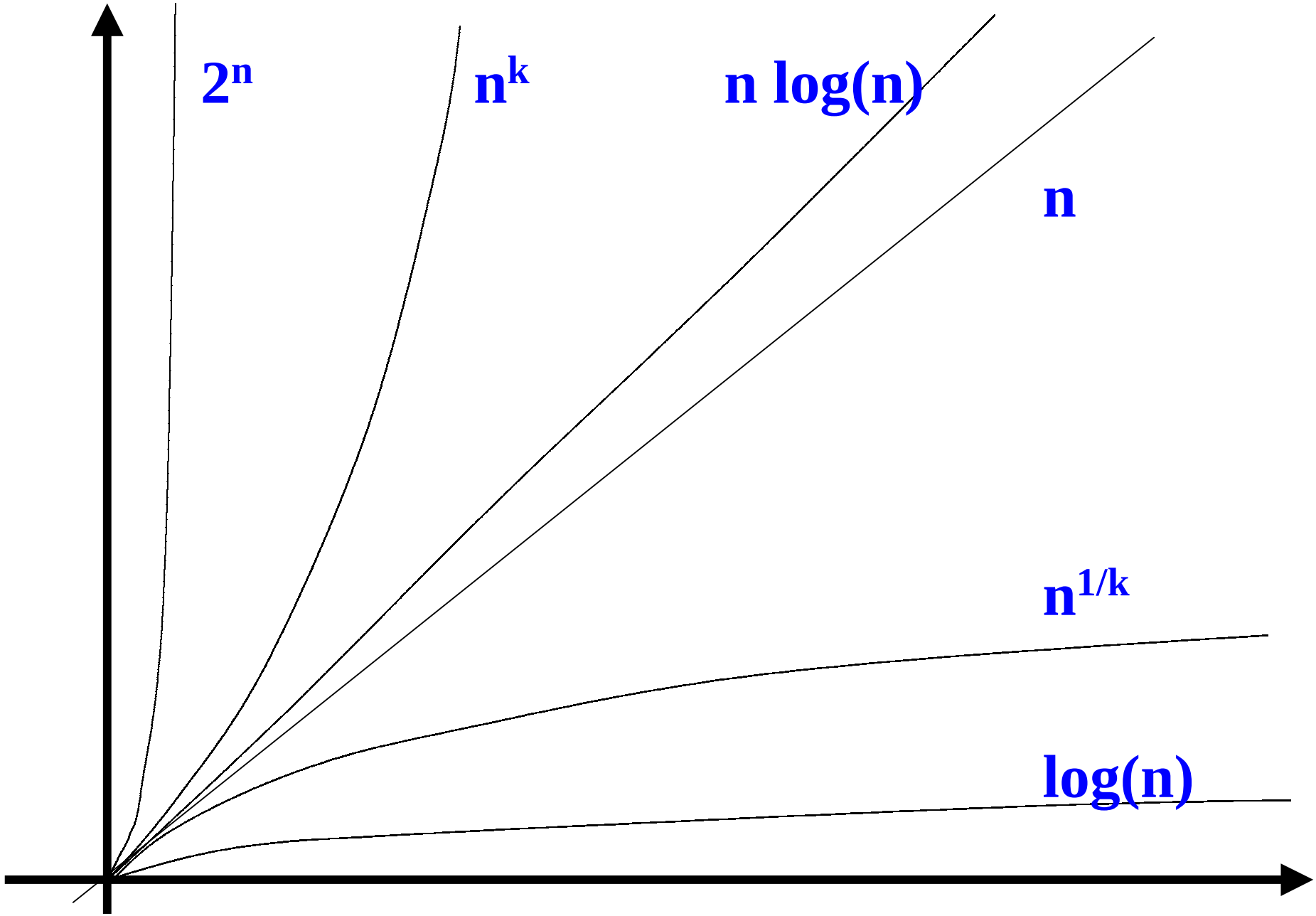
O(n): lineare Funktionen.

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.


2^n
n^k
n log(n)
n
$n^{1/k}$
log(n)

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 $\Theta(f+g) = \Theta(f) = \Theta(g)$.

Prüfen Sie selbst nach, ob folgende Formeln gelten:

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

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

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

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.