Gliederung der Grundvorlesung

1. Grundlagen der Programmierung

~~1.1 Algorithmen und Sprachen~~
1.2 Aussagen über Algorithmen
1.3 Daten und ihre Strukturierung
1.4 Grundbegriffe der Programmierung
1.5 Die Sprache Ada 95
1.6 Semantik von Programmen
1.7 Komplexität von Algorithmen und Programmen

Gliederung des Kapitels 1.2

1.2 Aussagen über Algorithmen

1.2.1 Charakteristika von Algorithmen
1.2.2 Grenzen der Algorithmen, Unentscheidbarkeit
1.2.3 O-Notation
1.2.4 Darstellung durch Gleichungen

1.2.1 Charakteristika von Algorithmen

Wie lauten notwendige Eigenschaften, die ein Algorithmus erfüllen muss, "damit er ein Algorithmus sein kann"?

Beispielsweise darf ein Algorithmus nicht in einem Schritt eine unendliche Menge von Möglichkeiten abprüfen oder unendlich viele Kopien von sich erzeugen können.

Auch darf ein Algorithmus nicht über unendlich viele verschiedene elementare Handlungen verfügen können; dies dürfen sogar nur beschränkt viele sein, da man anderenfalls keine mechanische Maschine zur Bearbeitung bauen könnte.

Wir listen einige Eigenschaften auf, die aus der Forderung an Algorithmen "realisierbar mit einer mechanischen Maschine" abgeleitet werden können.

Ein Algorithmus ist eine Vorschrift, die die Reihenfolge von durchzuführenden Handlungen (Operationen) auf Daten (Operanden) genau beschreibt. Hierbei muss gelten:

a) Die Daten sind "diskret" aufgebaut und es gibt beschränkt viele ("digitale") Zeichen $\{a_1, ...,a_k\}$, so dass jedes Datum eine endliche Folge dieser Zeichen ist.
b) Die Operationen sind "diskret" aufgebaut. Genauer: Es gibt beschränkt viele Zeichen $\{b_1, ...,b_m\}$, so dass jede Operation einschließlich ihrer Operanden hiermit beschrieben werden kann.
c) Die Vorschrift ist eine endliche Folge von Operationen. Die Vorschrift wird schrittweise abgearbeitet (diskrete Zeitskala).

Hinweis: Eine Menge heißt "diskret", wenn ihre Elemente gut unterscheidbar und mit endlicher Länge darstellbar sind. Erfolgt die Darstellung mit beschränkt vielen Zeichen, so spricht man von einer digitalen Darstellung.

d) Eine der Operationen ist als Startoperation ausgezeichnet.
e) Für jede Operation ist unmittelbar nach ihrer Ausführung bekannt, welches die möglichen (endlich vielen) Folgeoperationen sind oder ob der Algorithmus abbricht (terminiert).
f) Die Eingabe für die Vorschrift ist eine (eventuell unendliche oder auch leere) Folge von Daten (vgl. a).
g) In jedem Schritt (d.h. zu jedem Zeitpunkt) gilt: Die bis dahin bearbeitete oder betrachtete Menge an Daten und durchgeführten Operationen ist endlich.

Hinweis: Ein Algorithmus verfügt über eigene Speicherbereiche für die Daten und für die Vorschrift. Beide Bereiche kann der Algorithmus während seiner Abarbeitung verändern, aber in jedem Schritt nur einen endlichen Bereich. Beide Bereiche sind prinzipiell unendlich groß, auch wenn zu jedem Zeitpunkt nur ein endlicher Teil betrachtet worden sein kann.

Hinweis: Obige Ausführungen bilden keine richtige Definition, sondern nur eine Liste von umgangssprachlich formulierten Forderungen.

Turingmaschinen erfüllen diese Forderungen, aber auch andere "Rechenmaschinen" und die bereits vorgestellten Grammatiken, siehe Kapitel 1.1.5.

Auch Programme beschreiben Algorithmen. Prüfen Sie die Forderungen an einer Programmiersprache und ihren Programmen nach.

1.2.1.1.: Wünschenswerte Anforderungen an Algorithmen:

Determinismus, Nichtdeterminismus: Oft weiß man nicht, welches die nachfolgende Operation sein soll, und möchte eine Menge möglicher Folgeoperationen angeben. Solange diese Menge bei jeder Operation endlich ist, lässt sich das Problem durch einen normalen Algorithmus simulieren (siehe später: BFS-Verfahren mit BFS = Breadth-First-Search). Ist diese Menge jedoch unendlich, so liegt kein Algorithmus mehr vor.

In der Praxis verlangt man meist, dass Algorithmen deterministisch sein müssen, d.h., nach Abarbeitung jeder Operation steht eindeutig fest, welches die nachfolgende Operation ist oder ob die Berechnung hiermit beendet ist. Diese Forderung lässt sich aber bei vielen Anwendungen, insbesondere wenn Programme miteinander kommunizieren, nicht aufrecht halten.

(Vollständige) Terminierung = ein Algorithmus muss für alle Eingaben nach endlich vielen Schritten anhalten. Man möchte es also bei den realisierten Abbildungen nur mit der Menge der total rekursiven Funktionen $\Re$ zu tun haben, vgl. Definition 1.1.3.6.

In der Praxis wird dies häufig gefordert, insbesondere von Benutzern, die auf jede Eingabe eine Antwort erwarten, möglichst schon nach kurzer Zeit. Die Bedingung der vollständigen Terminierung ist für Algorithmen aber *nicht notwendig* und ist auch nicht für alle Algorithmen erwünscht. Z.B. sollte ein Betriebssystem prinzipiell unendlich lange arbeiten.

Algorithmen können Algorithmen als Daten einlesen. Dies überlegt man sich leicht wie in Abschnitt 1.1.7:

Ein Algorithmus lässt sich als Programm formulieren und ist dann eine Folge von Zeichen über dem Terminalalphabet der Programmiersprache, d.h., jeder Algorithmus lässt sich als ein Wort $w \in \Sigma^*$ auffassen. Es gibt jedoch Algorithmen, die solche Zeichenfolgen einlesen und verarbeiten können, folglich kann man Algorithmen als Eingabe für Algorithmen verwenden.

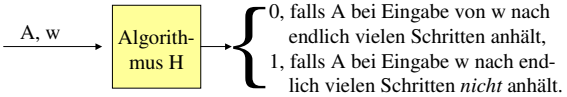
Dies ist nicht überraschend; denn ein Compiler ist ein Algorithmus, der Algorithmen einliest und sie in Programme der Maschinensprache übersetzt.

1.2.2 Grenzen der Algorithmen, Unentscheidbarkeit

Was kann man mit Algorithmen ***nicht*** beschreiben?

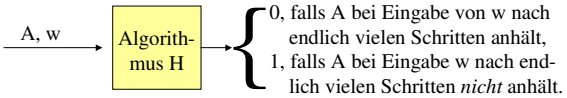
Betrachte folgende Aufgabe:
Konstruiere einen Algorithmus H, der beliebige Algorithmen und zugehörige Eingabedaten einlesen kann und der zu jedem beliebigen Algorithmus A und zu jeder Folge von Daten w in endlich vielen Schritten feststellt, ob der Algorithmus A für die Eingabedaten w nach endlich vielen Schritten anhält oder nicht.
Bezeichnung 1.2.2.1: Die Aufgabe, einen solchen immer terminierenden Algorithmus H zu finden, bezeichnet man als das Halteproblem für Algorithmen.

Gesucht wird also ein Algorithmus H

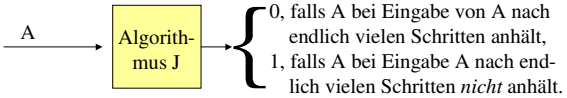


für alle Algorithmen A und für alle Eingabefolgen w.

Angenommen, es gibt solch einen Algorithmus H mit $\forall A\ \forall w$

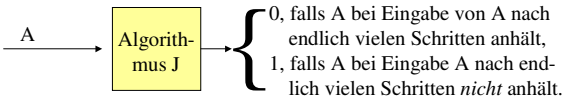


Speziell gibt es dann auch einen Algorithmus J mit $\forall A$

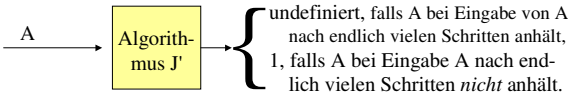


für alle Algorithmen A, indem man in H nur den Algorithmus A eingibt und dann H mit der Eingabe A und A (=w) ablaufen lässt.

Zu diesem Algorithmus J mit $\forall A$

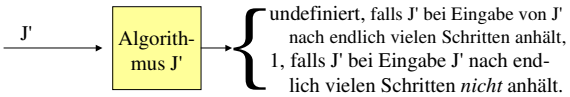


gibt es dann auch einen Algorithmus J' mit $\forall A$



Klar wegen: Modifiziere J so, dass der Algorithmus J anstelle der Ausgabe 0 in eine unendliche Schleife geht; so erhält man J'.

Was macht J' bei Eingabe von J'?



Fall 1: Bei Eingabe von J' hält J' an.
Dann liefert J' bei Eingabe von J' den Wert 1.
Nach Definition von J' hält dann J' bei Eingabe J' *nicht* an.
Widerspruch!

Fall 2: Bei Eingabe von J' hält J' *nicht* an.
Dann läuft J' bei Eingabe von J' in eine unendliche Schleife.
Nach Definition von J' hält dann J' bei Eingabe J' an und liefert 1.
Widerspruch!

Beide möglichen Fälle führen also auf einen Widerspruch.
Mehr Möglichkeiten gibt es nicht.
Folglich muss die Annahme, dass es den Algorithmus H gibt, falsch gewesen sein. Es gilt daher der

Satz 1.2.2.2 (Unlösbarkeit des Halteproblems)
Es gibt keinen Algorithmus H, der zu jedem beliebigen Algorithmus A und zu jeder Folge von Daten w in endlich vielen Schritten feststellt, ob der Algorithmus A für die Eingabedaten w nach endlich vielen Schritten anhält oder nicht.

Dieses Resultat formuliert man kurz in der Form:

Das Halteproblem ist algorithmisch nicht lösbar.

Anmerkung 1.2.2.3: Eine Menge $L \subseteq \Sigma^*$ heißt entscheidbar, wenn es einen Algorithmus mit einem immer terminierenden (vgl. 1.1.3.6) Programm π mit der Eingabemenge $E_\pi = \Sigma^*$ und der realisierten Funktion $f_\pi: \Sigma^* \to \{0,1\}$ gibt, so dass für alle $w \in \Sigma^*$ gilt:

$$f_\pi(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{falls } w \notin L \end{cases}$$

Anderenfalls heißt die Menge L unentscheidbar.

Der obige Satz besagt also:
Das Halteproblem für Algorithmen ist unentscheidbar.

Beachte: Das Halteproblem H lässt sich als Teilmenge $H \subseteq (\Sigma' \cup \{\eta\})^*$ für ein geeignetes Alphabet Σ' auffassen, in welchem sich alle Algorithmen und eingebbaren Daten beschreiben lassen, z.B. das Alphabet $\Sigma' = \hat{A}$. Sei $\eta \notin \Sigma'$. Dann gilt nämlich:
$H = \{u\eta v \mid u, v \in \Sigma'^*,$ u beschreibt einen Algorithmus und u hält für die Eingabe v an.$\} \subseteq (\Sigma' \cup \{\eta\})^*$.

Anmerkung 1.2.2.4: Für eine Menge $L \subseteq \Sigma^*$ heißt die Abbildung $\chi_L: \Sigma^* \to \{0,1\}$, für alle $w \in \Sigma^*$ definiert durch

$$\chi_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{falls } w \notin L \end{cases}$$

die charakteristische Funktion von L.

Die charakteristische Funktion verbindet die Begriffe "Teilmenge" und "Funktion" miteinander: Zu jeder Teilmenge gehört die charakteristische Funktion und zu jeder Funktion $g: \Sigma^* \to \{0,1\}$ gehört die Teilmenge $\{w \in \Sigma^* \mid g(w)=1\} \subseteq \Sigma^*$.

Satz 1.2.2.2 besagt also $\chi_H \notin \Re$ (und natürlich auch $\chi_H \notin \wp$).
Das heißt: Die charakteristische Funktion des Halteproblems $\chi_H: (\Sigma' \cup \{\eta\})^* \to \{0,1\}$ ist nicht rekursiv, vgl. 1.1.3.5.

Bezeichnung 1.2.2.5: Die Aufgabe, einen immer terminierenden Algorithmus H_ε zu finden, der zu jedem beliebigen Algorithmus A nach endlich vielen Schritten feststellt, ob der Algorithmus A für die leere Eingabe ε nach endlich vielen Schritten anhält oder nicht, bezeichnet man als das spezielle oder als das ε-Halteproblem für Algorithmen.

Satz 1.2.2.6: Das ε-Halteproblem ist unentscheidbar.

Beweis durch Rückführung auf das Halteproblem. Wir konstruieren zu jedem Algorithmus A und zu jeder Eingabe w einen Algorithmus A_w, so dass gilt: A hält bei Eingabe von w genau dann an, wenn A_w mit der leeren Eingabe anhält. Wäre das ε-Halteproblem also entscheidbar, dann wäre auch das Halteproblem entscheidbar, im Widerspruch zu Satz 1.2.2.2. Folglich muss das ε-Halteproblem unentscheidbar sein.

Es sei A ein Algorithmus und $w \in \Sigma^*$ eine Eingabefolge. Dann wandele A in folgenden Algorithmus A_w um: A_w ist wie A aufgebaut, besitzt jedoch eine weitere Zeichenketten-Variable X und an den Anfang des Anweisungsteils wird die Wertzuweisung X := "w"; gesetzt. Die read-Befehle werden jetzt dadurch ersetzt, dass immer die nächsten Zeichen von X (anstelle: "von der Eingabe") der jeweiligen Variablen zugewiesen werden. Der so erhaltene Algorithmus A_w liest also nichts ein und arbeitet genau so, wie A bei der Eingabe w arbeitet. Folglich gilt:

A hält bei Eingabe von w $\Leftrightarrow$ A_w hält bei Eingabe von ε.

Damit ist Satz 1.2.2.6 bewiesen.

■

1.2.3 O-Notation

Wer Programme einsetzt, fragt meist nach der Zeitspanne, innerhalb welcher die Ergebnisse ausgegeben werden. Man spricht von der "Zeitkomplexität" des Programms.

Diese "Rechendauer" ist abhängig von der Eingabe. In der Regel definiert man den Zeitaufwand $t_\pi: \mathbb{N}_0 \to \mathbb{N}_0$ als Funktion der Länge der Eingabe des Programms π:

$t_\pi(n)$ = maximale Zeit, die das Programm π für irgendeine Eingabe w der Länge n benötigt.

t_π ist natürlich nur dann eine totale Funktion, wenn das Programm π immer terminiert. Meist verlangt man dies, wenn man die Zeitkomplexität berechnen will.
Wir betrachten zunächst ein Beispiel.

Beispiel

```
program was is
var  A, B: natural;
begin read (A);
      B := 1;
      while A > 1 do A := A div 2; B := B+1 od;
      write (B)
end;
```

Mit Ablaufprotokollen ermittelt man einige Werte der realisierten Funktion $g: \mathbb{N}_0 \to \mathbb{N}_0$:

a	g(a)	a	g(a)	a	g(a)
0	1	4	3	8	4
1	1	5	3	16	5
2	2	6	3	80	7
3	2	7	3	1024	11


```
program was is
var  A, B: natural;
begin read (A);
      B := 1;
      while A > 1 do A := A div 2; B := B+1 od;
      write (B)
end;
```

Man vermutet, dass die realisierte Funktion $g: \mathbb{N}_0 \to \mathbb{N}_0$ lautet: g(a) = Länge der binären Darstellung der Zahl a.

Dies trifft zu, weil in jedem Schritt die Länge der Zahl A durch die Wertzuweisung A := A div 2 genau um eins verringert wird, bis eine Darstellung der Länge 1 erreicht ist. Die Schleife wird einmal weniger, als die Länge angibt, durchlaufen; da B anfangs auf 1 gesetzt wurde, wird daher genau die Länge der Binärdarstellung der Eingabe berechnet.


```
program was is
var  A, B: natural;
begin read (A);
      B := 1;
      while A > 1 do A := A div 2; B := B+1 od;
      write (B)
end;
```

Wie lange dauert nun die Berechnung? Wir nehmen an, die Auswertung aller Ausdrücke und die Durchführung einer Wertzuweisung dauern gleich lange, so erhält man (es sei n die Länge der Binärdarstellung der Eingabezahl a):

$1 + 1 + (n-1) * (1 + 1 + 1) + 1 + 1 = 3n + 1$

Die Größenordnung ist also proportional zu n (man sagt, sie sei O(n)), und meist interessiert nur diese Abschätzung, welche multiplikative und additive Konstanten ignoriert. ■

Um die Größenordnung einer reellwertigen oder ganzzahligen Funktion zu beschreiben, verwenden wir die so genannten *Landau-Symbole* (nach dem deutschen Mathematiker Edmund Landau, 1877-1938). Hierbei werden multiplikative und additive Konstanten vernachlässigt und nur der Term in Abhängigkeit von n, der für $n \to \infty$ 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.

Insgesamt verwendet man folgende 5 Klassen O, o, Ω, ω und Θ, wobei von uns am häufigsten " O " eingesetzt wird.

Definition 1.2.3.1: "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).

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

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

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

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

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

Landau-Symbole

Erläuterungen 1.2.3.2: 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 schwächer als f wächst, 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) 1.2.3.2: 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 1.2.3.3: 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:

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

1.2.3.4: 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 kennen Sie den Logarithmus log(n). Noch wesentlich schwächer wächst der "iterierte Logarithmus":

$\log^*(n) = 0$, für n=0 und 1,

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

Aufgabe für Sie: Untersuchen Sie diese Funktion log*.

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

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

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:

Der Logarithmus wächst bekanntlich schwächer als jede noch so kleine positive Potenz, d.h.: Für jedes $m \geq 1$ gilt ab einem hinreichend großen n: $\log(n) < n^{\frac{1}{m}}$ und folglich $\log^m(n) < n$.

$\Omega(n)$ = Klasse der mindestens linear wachsenden Funktionen:
$\Omega(n)$:= $\{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0: n \leq c \cdot g(n)\}$.
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:
$\Theta(n)$:= $\{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0: c_1 \cdot n \leq g(n) \leq c_2 \cdot n\}$.
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.
$O(n^2)$:= $\{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0: g(n) \leq c \cdot n^2\}$.
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 meist folgende Funktionsklassen:

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.

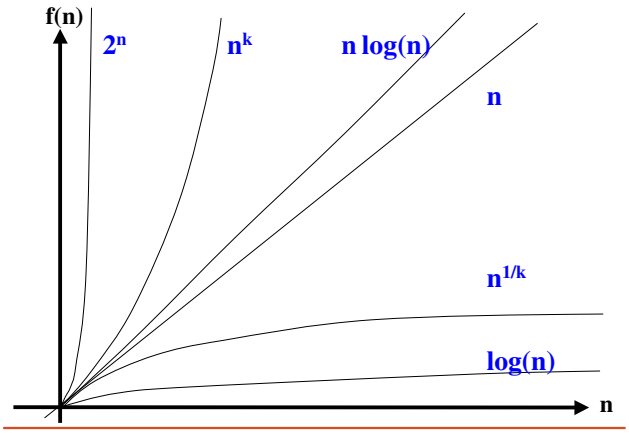
O(n): lineare Funktionen.

O(n·log(n)): Das sind Funktionen, die "ein wenig" stärker als linear wachsen.

$O(n^2)$: höchstens quadratisch wachsende Funktionen.

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

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



Hilfssatz 1.2.3.5:
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 $\Theta(f+g) = \Theta(f) = \Theta(g)$.

Zur Übung: Untersuchen Sie, 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 nämlich nur die Größenordnung der Komplexität und nicht der genaue Wert von Konstanten.

1.2.4 Darstellung durch Gleichungen

In Abschnitt 1.2.3 wurde die Zeitkomplexität an einem Beispiel veranschaulicht und es wurde die O-Notation zur Beschreibung von Größenordnungen vorgestellt.

Neben der *Komplexität*, also dem Aufwand an Ressourcen wie Zeit, Speicherplatz, Programmlänge, Übertragungsmedien usw., spielt die *Korrektheit* eine zentrale Rolle. Hierunter versteht man den Nachweis, dass ein Algorithmus oder ein Programm genau das tut, was es tun soll.

Wie notiert man exakt, *was ein Programm tun soll?* Dies kann die Angabe der gewünschten realisierten Abbildung sein, man kann aber auch andere Beschreibungen verwenden. Wir illustrieren hier kurz "Gleichungen".

Beispiel 1.2.4.1: In Kapitel 1.1. (vgl. 1.1.2.4) wurde der größte gemeinsame Teiler ggT zweier natürlicher Zahlen durch folgende Gleichungen dargestellt:

(1) $ggT(a,b) = ggT(b,a)$ für alle $a,b \in \mathbb{N}_0$,
(2) $ggT(a,0) = ggT(a,a) = a$ für alle $a \in \mathbb{N}_0$ mit $a>0$,
(3) $ggT(a,b) = ggT(a-b,b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$,
$ggT(a,b) = ggT(a+b,b)$ für alle $a,b \in \mathbb{N}_0$,
(4) $ggT(a,b) = ggT(b, a \bmod b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$.

Es genügen bereits drei Gleichungen: für alle $a,b \in \mathbb{N}_0$:

$ggT(a,b) = ggT(b,a)$
$ggT(a,0) = a$ für $a > 0$
$ggT(a+b,b) = ggT(a,b)$

(Weisen Sie nach, dass die anderen Gleichungen hieraus folgen.)

Wir verlangen daher, dass die realisierte Abbildung g unseres zu schreibenden Programms für alle $a,b \in \mathbb{N}_0$ folgende Gleichungen erfüllen muss:

$g(a,b) = g(b,a)$
$g(a,0) = a$ für $a > 0$
$g(a+b,b) = g(a,b)$ [hieraus folgt $g(a,b) = g(a-b,b)$]

Das Programm "euklid" (siehe Beispiel 1.1.2.3, nochmals abgedruckt auf der nächsten Folie) erfüllt dies:

if $A < B$ *then* $H := A; A := B; B := H$ *fi;*
erfüllt $g(A,B) = g(B,A)$

while $B \neq 0$ *do* $H := A \bmod B; A := B; B := H$ *od*
erfüllt $g(A,B) = g(A-B,B) = g(B,A-B)$

Der Abbruch *"B=0"* mit *"write(A)"* erfüllt $g(A,0) = A$.

Erinnerung an Beispiel 1.1.2.3:

```
program euklid1 is
var A, B, H: natural;
begin read (A); read (B);
  if (A > 0) or (B > 0) then
     if A < B then H := A; A := B; B := H fi;
     while B ≠ 0 do
        H := A mod B; A := B; B:= H od
  fi;
  write (A)
end
```


Zugleich kann man aus den Gleichungen

$g(a,b) = g(b,a)$
$g(a,0) = a$ für $a > 0$
$g(a+b,b) = g(a,b)$ [hieraus folgt $g(a,b) = g(a-b,b)$]

auch den Algorithmus ablesen:

Ersetze immer die linke Seite durch die rechte Seite, also

{ (A,B) ersetzen durch (A-B,B),
falls A-B < B, dann (A,B) ersetzen durch (B,A) }

diese beiden Anweisungen iterieren, bis B=0 ist.

Die Ersetzung (A,B) durch (B, A mod B) kürzt diesen Vorgang nur ab.

Fazit: Die Beschreibung durch Gleichungen liefert Anforderungen an das Programm. Zugleich kann sie Hinweise geben, wie ein Lösungsalgorithmus zu finden ist. ■

Beispiel 1.2.4.2. Quersumme einer Zahl z zur Basis $b \geq 2$:

Sei $z = z_{n-1} z_{n-2} \ldots z_1 z_0$ die Zahldarstellung zur Basis $b \geq 2$ mit $z_i \in \{0, 1, \ldots, b-1\}$, so ist die Quersumme $quer_b(z) = z_{n-1} + z_{n-2} + \ldots + z_1 + z_0$. Gleichungen hierzu:

$quer_b(z) = z$ für $z < b$,
$quer_b(z) = quer_b(z \text{ div } b) + quer_b(z \bmod b)$ für alle $z \geq 0$.

Wegen $z \bmod b < b$ und $quer_b(0) = 0$ liefert dies sofort:

```
program quersumme is
var Z, B, Q: natural;
begin read (Z); read (B); Q := 0;
   if B ≥ 2 then
      while Z > 0 do Q := Q + (Z mod B); Z := Z div B od;
      write (Q)     else write ("Basis zu klein.") fi
end
```

■

Beispiel 1.2.4.3. Teilworterkennung

Beim Editieren sucht man im bereits bestehenden Text u manchmal ein Teilwort v. Formale Beschreibung:
Sei $u = u_1 u_2 \ldots u_{n-1} u_n$ ein Wort der Länge $n \geq 0$ mit $u_i \in \Sigma$ für ein Alphabet $\Sigma = \{a_1, a_2, \ldots, a_{r-1}, a_r\}$.
Sei $v = v_1 v_2 \ldots v_{m-1} v_m$ ein weiteres Wort ($m \geq n \geq 0$, $v_i \in \Sigma$).
Entscheide, ob das Wort v als Teilwort in u enthalten ist.
(*Definition:*
v ist Teilwort von u $\Leftrightarrow$ es gibt Wörter x und y mit $u = x\,v\,y$.)

Für die Entscheidungsfunktion teil: $\Sigma^* \times \Sigma^* \to \{\text{false}, \text{true}\}$ mit:
teil(u,v) = true $\Leftrightarrow$ v ist Teilwort von u
$\Leftrightarrow$ es gibt Wörter x und y mit $u = x\,v\,y$
wollen wir nun einige Gleichungen angeben.

Für teil: $\Sigma^* \times \Sigma^* \to \{\text{false}, \text{true}\}$ gelten z.B. die folgenden Gleichungen:

teil(u,v) = false für alle Wörter $u, v \in \Sigma^*$ mit $|v| > |u|$,
teil(u,v) = false für alle Wörter $u, v \in \Sigma^*$ mit $|u| = |v|$ und $u \neq v$,
teil(u,u) = true für alle Wörter $u \in \Sigma^*$,
teil(u,ε) = true für alle Wörter $u \in \Sigma^*$,
teil(a,b) = false für alle Zeichen $a, b \in \Sigma$ mit $a \neq b$,
teil(ax,by) = teil(x,by) für alle $x, y \in \Sigma^*$ und $a, b \in \Sigma$ mit $a \neq b$,
teil(ax,ay) = teil(x,y) or teil(x,ay) für alle $x, y \in \Sigma^*$ und $a \in \Sigma$,
teil(wx,by) = teil(x,by) für alle $x, y, w \in \Sigma^*$ und $b \in \Sigma$ und das Zeichen b kommt im Wort w nicht vor,
teil(ax,y) = teil(x,y) für alle $x, y \in \Sigma^*$ und $a, b \in \Sigma$, falls y nicht Anfangswort von ax ist (d.h., $ax \neq yz$ für jedes z).

Können Sie hieraus einen Algorithmus konstruieren, der zu zwei einzulesenden Wörtern u und v (stellen Sie diese beiden Wörter durch Felder array[1..n] of character dar) feststellt, ob v Teilwort von u ist?
Die obigen Gleichungen sind "von links nach rechts" aufgebaut, d.h., sie skizzieren, wie man das Wort v von links nach rechts über das Wort u verschiebt und dabei mittels
teil(ax,ay) = teil(x,y) or teil(x,ay)
den Test, ob das zweite Wort v ab dieser Position im Wort u auftritt, schrittweise durchführt.
Versuchen Sie, einen einfachen Algorithmus zur Lösung des Problems zu schreiben.

■

1.2.5 Historische Anmerkung

Im 19. Jahrhundert hoffte man, dass man alle Probleme mit Algorithmen lösen könne, die sich durch Gleichungen über den natürlichen Zahlen formal darstellen lassen. 1931 bewies der österreichische Logiker K. Gödel (1906-1978), dass dies jedoch nicht möglich ist. Wir haben hier einen allgemeinen Widerspruchsbeweis für Algorithmen verwendet, der wie die Cantorsche Diagonalisierung zum Beweis, dass reelle Zahlen überabzählbar sind, aufgebaut ist.
Die Darstellung durch Funktionen-Gleichungen ist in der Mathematik lange bekannt und wurde 1936 von A. Church für die Definition der Berechenbarkeit eingeschränkt. Diese Ideen finden in der Sprache LISP (und Nachfolgesprachen wie SCHEME, MIRANDA, HASKELL usw.) seinen Niederschlag.

Aus dem Unentscheidbarkeitssatz 1.2.2.2 lässt sich eine Fülle von weiteren Problemen ableiten, die alle algorithmisch nicht lösbar sind, z.B.
- entscheide, ob Programme für *alle* Eingaben anhalten,
- entscheide, ob Programme für *alle* Eingaben nicht anhalten,
- entscheide, ob zwei beliebige Programme dasselbe berechnen, also die gleiche realisierte Abbildung besitzen,
- entscheide, ob eine kontextfreie Grammatik alle Wörter erzeugt usw.

Weitere Aussagen hierzu finden Sie in Theorie-Vorlesungen und in Büchern über Grundlagen der Informatik oder der Philosophie.

1.2.6 Übungsaufgaben

- noch Fehlanzeige, solange siehe Übungen zur Vorlesung -