1 Asymptotische Notation

In diesem Kapitel wollen wir die Grundlagen der asymptotischen Notation vermitteln. Diese ist ein wichtiges Werkzeug zur Betrachtung der Effizienz eines Algorithmus.

1.1 Grundlagen und einleitende Beispiele

Eine grundlegende Aufgabenstellung in der Informatik ist die Analyse von Rechenverfahren. Die genaue Angabe beispielsweise der Laufzeit ist oft schwierig, da sie von der Programmiersprache, dem verwendeten Compiler und der zugrundegelegten Maschine abhängig ist. Wir benötigen daher Aussagen, die unabhängig von letzteren Faktoren sind, wie zum Beispiel

„Die Laufzeit nimmt höchstens linear zur Eingabelänge $n \in \mathbb{N}$ zu."

Wir unterscheiden dazu zwischen einem *Problem* (*allgemeine abstrakte Formulierung*) und der *Instanz* eines solchen.

Beispiel 1.1.
Problem:

Gegeben $a, b \in \mathbb{N}$. Berechne die Summe $s = a + b$ der Zahlen a und b.

Konkrete Instanz des Problems:

Gegeben 1356 und 6556, berechne $1356 + 6556$.

In unserem Beispiel werden in der Probleminstanz die Variablen der allgemeinen Problemformulierung mit konkreten Werten (hier z.B. $a = 1356$ und $b = 6556$) belegt, woraus sich eine konkrete zu lösende Aufgabenstellung ergibt. Der Grund dieser Trennung liegt darin, dass Lösungsverfahren im Allgemeinen für die abstrakten Probleme geschaffen werden, um sie auf alle Instanzen letzterer anwenden zu können. Ziel ist hierbei die Entwicklung von *mechanisch ausführbaren Rechenverfahren* zur Problemlösung.

Definition 1.1. Ein Verfahren, mit dem man quasi „mechanisch" beliebige Instanzen eines vorgegebenen Problems lösen kann, nennt man **Algorithmus**[1].

[1]Das Wort *Algorithmus* geht auf die lateinische Fassung des arabischen Namens *Al-Chwarizmi* zurück. Dieser Gelehrte schrieb seine mathematischen Werke im 9. Jahrhundert am Hofe des Kalifen Al-Ma'mun in Bagdad. Eines von ihnen hieß *Hisab al-gabr wal-muqabala*, d.h. Rechenverfahren durch Ergänzen und Ausgleichen. Das Wort *Algebra* stammt aus diesem Titel und ist offenbar auch historisch mit dem Lösen von Gleichungen verbunden. In Spanien tauchte der Name des Verfassers

Beispiel 1.2. Wir kennen für das Problem aus Beispiel 1.1 einen Lösungsalgorithmus aus der Schule (*schriftliches Addieren*). Wenden wir diesen auf die Instanz an, so erhalten wir

$$\begin{array}{cccccc} & 1 & 3 & 5 & 6 \\ + & 6 & 5 & 5 & 6 \\ & \textit{0} & \textit{1} & \textit{1} & \\ \hline & 7 & 9 & 1 & 2 \end{array}$$

wobei die kleinen kursiven Ziffern den Übertrag darstellen.

Um ein Problem mittels Algorithmen auf einem Rechner lösen zu können, müssen die Daten der realen Objekte des Problems mittels geeigneter Datenstrukturen (virtuelle Repräsentation (Objekte)) im Rechner gespeichert werden.

Beispiel 1.3. *Eine Klasse für natürliche Zahlen*
Nehmen wir an, dass die Zahlen a und b beliebig groß (lang) sein können und dass sie nicht notwendigerweise mit dem Datentyp `unsigned int` (32 Bit $[\leq 2^{32}-1]$) oder `unsigned long` (64 Bit $[\leq 2^{64}-1]$) repräsentiert werden können. Bei einer vorgegebenen Basis $B \in \mathbb{N}$, $B \geq 2$ lässt sich jede nichtnegative Zahl $a \in [0, B^n - 1]$, $n \in \mathbb{N}$ schreiben als

$$a = \sum_{i=0}^{n-1} a_i B^i =: (a_{n-1}, ..., a_0)$$

mit $0 \leq a_i < B$ für alle $i \in \{0, 1, ..., n-1\}$. Es bietet sich also an, für die Repräsentation beliebig großer Zahlen im Rechner $B = 2^{31}$ zu wählen (das letzte Bit wird für den Übertrag der Addition bei der Berechnung verwendet) und eine Zahl $a = (a_{n-1}, ..., a_0)$ als Vektor `vector<unsigned int>` von n `unsigned int` darzustellen. Zum Rechnen mit diesen Zahlen erstellen wir eine Klasse `Integer`, die eine für den Benutzer bequeme Verwendung garantiert. Lange natürliche Zahlen speichern wir als Objekte dieser Klasse. Wir stellen sie hier auszugsweise vor.

`Integer a`	deklariert und definiert ein Objekt a der Klasse `Integer`
`a.digits()`	gibt die Anzahl der Stellen n von a bzgl. der Basis B zurück
`a[i]`	gibt die i-te Stelle a_i von a als `unsigned int` zurück (0, falls $i \geq n$)

Offenbar genügt nach der Schulmethode die Addition von drei „einstelligen“ Zahlen (aktuelle Ziffer der ersten und der zweiten Zahl, sowie der Übertrag), um so iterativ das Ergebnis der Gesamtaddition zu berechnen. Diese ist für uns an dieser Stelle eine *primitive Operation* (hier: *primitive Addition*)

---

rund drei Jahrhunderte später in einer lateinischen Bearbeitung seiner Bücher auf. Diese beginnt mit den Worten:

Dixit Algoritmi... [Es sprach Algoritmi...]

Im Laufe der Zeit entwickelte sich so der Begriff *Algorithmus*, den man ganz allgemein mit mechanisch ausführbaren Rechenverfahren verband.

`unsigned int primitive_add(unsigned int i, unsigned int j, unsigned int& carry)`.

Das Resultat einer primitiven Addition kann höchstens zweistellig sein, wobei der Übertrag in der `unsigned int`-Variablen `carry` gespeichert wird. Damit ergibt sich der Additionsalgorithmus nach der Schulmethode in C++-Syntax zu

```
1  Integer Integer::operator+(const Integer& a)
2  {
3     int n = max(a.digits(),this->digits());
4
5     // Variable fuer die Summe
6     Integer sum(n+1);
7
8     // Variable fuer den Uebertrag
9     int carry = 0;
10
11    for(int i=0; i<n; i++)
12    {
13       // primitive Addition mit 2-stelligem Resultat
14       sum[i] = primitive_add(a[i],(*this)[i],carry);
15    }
16    sum[n] = carry;
17    return sum;
18 }
```

Offenbar benötigt der Algorithmus, um zwei n-stellige Zahlen zu addieren, n primitive Additionen, siehe `for`-Schleife. Die „Größenordnung“ der Laufzeit wird bestimmt durch (ist proportional zur) Zahl der primitiven Operationen. Bei der Rückgabe (`return sum`) findet eine Kopieroperation statt[2], ebenso sind primitive Operationen an der Berechnung des Maximums beteiligt, im Aufruf des Konstruktors von `Integer` (`Integer sum(n+1)`) verstecken sich weitere, ja prinzipiell müssen hier auch alle Zuweisungen (z.B. `sum[n] = carry`), Vergleiche (z.B. `i<n`) und sonstige Rechenoperationen (z.B. `i++`) genannt werden. An vielen Stellen ist die Zahl dieser Operationen konstant und unabhängig von der Problemgröße. Beispielsweise ist bei der Maximumsberechnung immer die selbe Zahl an primitiven Operationen beteiligt, unabhängig von den konkreten Werten `a.digits()` und `this->digits()`. Andere sind hier direkt abhängig von der Problemgröße. Es wird z.B. `i++` n mal ausgeführt, ebenso `i<n` und auch bei der Kopieroperation der Rückgabe wird der interne Vektor kopiert und somit $n+1$ `unsigned int` zugewiesen. Wir wollen hier die Sachlage nicht unnötig durch ein genaues Vorrechnen der exakten Anzahl der primitiven Operationen verkomplizieren. Wir halten aber fest, dass keine primitive Operation z.B. quadratisch in der Problemgröße n oft ausgeführt wird. Wir begnügen uns deshalb an dieser Stelle damit, dass eine Konstante $c \in \mathbb{R}$ existiert, so dass die

[2]Der Leser mache sich klar, dass hier tatsächlich `sum` kopiert wird.

Laufzeit durch $c \cdot n$ nach oben beschränkt wird. Genauer:

$$\text{Laufzeit von } \texttt{operator+} \leq c \cdot n.$$

Die Zahl der Stellen (Größe der Eingabe) n ist hier ein Maß für die Größe des Problems.

Beispiel 1.4. *Erweiterung mit Multiplikation*
Wir erweitern die Klasse aus dem vorangegangenen Beispiel durch den aus der Schule bekannten Multiplikationsalgorithmus für natürliche Zahlen. Wir rufen uns diesen an nachfolgender Instanz des zum Additionsproblem analogen Multiplikationsproblems noch einmal in Erinnerung.

	8	1	$*$	1	1	1
			8	1		
+				8	1	
+					8	1
			8	9	9	1

Zur Implementierung dieses Algorithmus wollen wir voraussetzen, dass bereits folgende C++-Operatoren existieren (Operator-Overloading).

- `Integer operator+(const Integer&)` (*benötigt n primitive Additionen*),
- `Integer operator*(unsigned int)` (*benötigt $\leq n+1$ primitive Additionen*),
- `Integer operator<<(int)` (*benötigt keine primitiven Additionen*),
- `Integer operator=(const Integer&)` (*benötigt keine primitiven Additionen*),

Damit ergibt sich der Algorithmus in C++-Syntax zu

```
1  Integer Integer::operator*(const Integer& a)
2  {
3     int n = this->digits();
4     int m = a.digits();
5
6     // Variable fuer das Produkt
7     Integer p;
8
9     for(int i=0; i<m; i++)
10    {
11       // multipliziere (*this) mit der i-ten Ziffer von a
12       // und schiebe das Resultat um i nach links
13       p = p+(((*this)*a[i]) << i);
14    }
15    return p;
16 }
```

Wir analysieren die Anzahl der primitiven Additionen $pA(n)$ für den Fall, dass beide Faktoren gleich lang sind, d.h. $m = n$. Betrachten wir

`p+(((*this)*a[i]) << i)`.

Wie oben angegeben benötigt `(*this)*a[i]` höchstens $n+1$ primitive Additionen, das Ergebnis von `((*this)*a[i]) << i` hat $n+1+i$ Stellen, also benötigt der +-Operator gerade $n+1+i$ primitive Additionen. Damit ergibt sich

$$\begin{aligned} pA(n) &\leq \sum_{i=0}^{n-1}(n+1+n+1+i) = \sum_{i=0}^{n-1}(2n+2+i) \\ &= 2n^2 + 2n + \sum_{i=0}^{n-1} i = 2n^2 + 2n + \frac{1}{2}n(n-1) \\ &= \frac{5}{2}n^2 + \frac{3}{2}n \leq 5n^2. \end{aligned}$$

1.2 Random Access Machine (RAM)

Um die Ressourcen abzuschätzen, welche von einem Algorithmus benötigt werden, verwenden wir im Folgenden ein einfaches Rechnermodell.

Definition 1.2. Eine **Random Access Machine** (**RAM**) ist ein Rechner,

1. der über eine unendliche Folge von Registern oder Speicherzellen verfügt
2. der folgende Operationen (**primitive Operationen**) in konstanter Zeit durchführt:
   a) Lese- oder Schreibzugriffe auf jedes Register
   b) Additionen, Subtraktionen, Multiplikationen, Divisionen von einfachen Datentypen für Zahlen (`int`, `float`, u.s.w.)
   c) Deklarationen von einfachen Datentypen (`int`, `float`, `short`, `bool`, u.s.w.)
   d) Sprunganweisungen („goto“), z.B. in `if`-Anweisungen
   e) Vergleiche (==, !=, <, >, >=, <=) für einfache Datentypen
   f) boolsche Operationen (&&, ||, &, |, u.s.w.) angewendet auf einfache Datentypen

Die Idee ist also allgemein, die Laufzeit durch die Anzahl der benötigten primitiven Operationen abzuschätzen und jede dieser primitiven Operationen dann mit einer (konstanten, aber nicht notwendigerweise mit der gleichen) Zeiteinheit zu bewerten.

Beispiel 1.5. *Skalarprodukt von Vektoren*
Wir betrachten folgende Funktion, die das Skalarprodukt von zwei vorgegebenen Vektoren `a` und `b` der gleichen Länge (`a.size()` = `b.size()` = n) berechnet.

```
1  double skalprod(vector<double>& a, vector<double>& b)
2  {
3      int i;
4      int laenge = a.size();
5
6      // Variable fuer das Ergebnis
7      double ergebnis = 0.0;
8
9      // Laufe die einzelnen Vektoreintraege durch
10     for(i=0; i<laenge; i++)
11     {
12         ergebnis += a[i]*b[i];
13     }
14     return ergebnis;
15 }
```

Wir analysieren den Code und wollen uns keine Gedanken über die genauen konstanten Zeiteinheiten machen, deshalb führen wir Konstanten c_1 bis c_8 ein.

Code	**Kosten**
`int i;`	c_1
`int laenge = a.size();`	c_2
`double ergebnis = 0.0;`	c_3
`i=0;`	c_4
`i<laenge;`	$c_5 \cdot n$ (Schleife)
`i++;`	$c_6 \cdot n$ (Schleife)
`ergebnis += a[i]*b[i];`	$c_7 \cdot n$ (Schleife)
`return ergebnis;`	c_8

Wir erhalten damit eine Laufzeit von

$$c_1 + c_2 + c_3 + c_4 + c_5 n + c_6 n + c_7 n + c_8 = (c_5 + c_6 + c_7)n + c_1 + c_2 + c_4 + c_4 + c_8.$$

Mit

$$c := 2 \max\{c_5 + c_6 + c_7, c_1 + c_2 + c_4 + c_4 + c_8\}$$

erhalten wir

$$\text{Laufzeit von } \texttt{skalprod} \leq c \cdot n.$$

Oft kann man nur eine obere Schranke für die Laufzeit angeben (Worst-Case-Analyse), jedoch sollte diese so nahe an der tatsächlichen Laufzeit liegen wie möglich. Unser Interesse gilt dem Laufzeitverhalten für große Eingaben n, genauer dem Verhalten für $n \to \infty$, um zu entscheiden, welcher Algorithmus von zwei gegebenen schneller bzw. effizienter ist. Wie wir in den vorangegangenen Beispielen schon gesehen haben, sind wir meist an der „Größenordnung“ der Laufzeit interessiert. Die Laufzeit selbst wird als Funktion in der Eingabegröße (Problemgröße) angegeben. Die asymptotische Notation dient nun zur Klassifizierung und zum Vergleich von Funktionen nach ihren Zuwachsraten.

1.3 Klassifizierung nach Zuwachsraten

Im Folgenden bezeichnen wir mit

$$\mathcal{F} := \{f : \mathbb{N} \to \mathbb{R}_0^+\}$$

die Menge aller Funktionen von $\mathbb{N}$ nach $\mathbb{R}_0^+$ und klassifizieren die darin enthaltenen Funktionen nach ihrem Wachstumsverhalten.

1.3.1 Die Funktionsklassen

Definition 1.3. Für $g \in \mathcal{F}$ ist die Funktionsklasse $\Theta(g)$ (lies: „Theta von g“) definiert als

$$\Theta(g) := \{f \in \mathcal{F} \mid \exists c_1 > 0 \exists c_2 > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)\}.$$

Intuitiv ist damit $f \in \Theta(g)$, wenn f in der gleichen „Größenordnung“ wie g ist, d.h. f wird durch g ab einer Stelle n_0 nach oben bzw. nach unten (durch Multiplikation mit Konstanten c_1 und c_2) beschränkt. Algorithmen mit den Laufzeiten $f(n)$ und $g(n)$ sind also gleich oder ähnlich effizient.

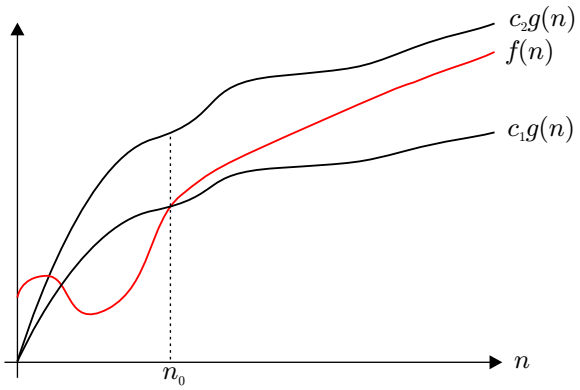

Abbildung 1.1: $f(n) \in \Theta(g(n))$

Analog definieren wir die „einseitigen Beschränkungen nach oben und unten“.

Definition 1.4. Für $g \in \mathcal{F}$ ist die Funktionsklasse $\mathcal{O}(g)$ (lies: „(groß) O von g“) definiert durch

$$\mathcal{O}(g) := \{f \in \mathcal{F} \mid \exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) \leq cg(n)\}$$

und die Funktionsklasse $\Omega(g)$ (lies: „groß Omega von g“) durch

$$\Omega(g) := \{f \in \mathcal{F} \mid \exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) \geq cg(n)\}.$$

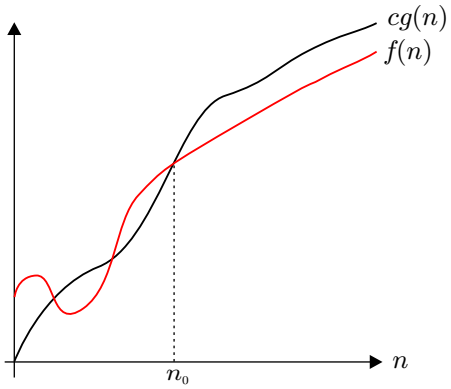

Abbildung 1.2: $f(n) \in \mathcal{O}(g(n))$

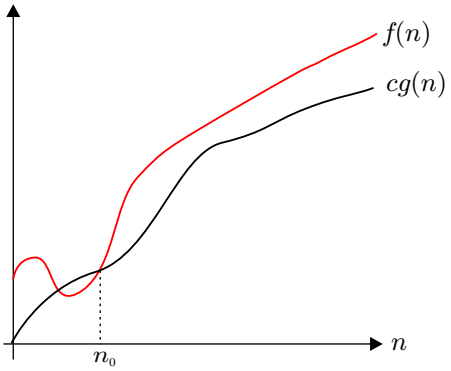

Abbildung 1.3: $f(n) \in \Omega(g(n))$

Wir hätten $\Theta(g)$ auch über den offensichtlichen Zusammenhang mit $\mathcal{O}(g)$ und $\Omega(g)$ definieren können, wie dies in der Fachliteratur häufig getan wird, denn es gilt

$$\Theta(g) = \mathcal{O}(g) \cap \Omega(g).$$

Abschließend führen wir noch zwei seltener gebrauchte Schreibweisen ein.

Definition 1.5. Für $g \in \mathcal{F}$ ist die Funktionsklasse $o(g)$ (lies: „klein o von g“) definiert durch

$$o(g) := \{f \in \mathcal{F} \mid \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) < cg(n)\}$$

und die Funktionsklasse $\omega(g)$ (lies: „klein omega von g“) durch

$$\omega(g) := \{f \in \mathcal{F} \mid \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) > cg(n)\}.$$

Der Unterschied zu $\mathcal{O}(n)$ bzw. $\Omega(n)$ besteht nur in der Ersetzung von ’$\exists c > 0$’ durch ’$\forall c > 0$’, d.h. die Funktionen f und g müssen sich beliebig weit (für n genügend groß) unterscheiden.

Beispiel 1.6. Für $g(n) := n^2$ ist $f(n) = 3n^2 + 7n + 12 \in \mathcal{O}(g(n))$, denn es gilt

$$f(n) = 3n^2 + 7n + 12 \underset{n \in \mathbb{N}}{\leq} 3n^2 + 7n^2 + 12n^2 = \underbrace{22}_{=:c} n^2 = cg(n).$$

Analog ist $f(n) \in \Omega(g(n))$, denn es ist

$$f(n) = 3n^2 + 7n + 12 \underset{n \in \mathbb{N}}{\geq} n^2 = g(n) = 1 \cdot g(n).$$

Also ist insgesamt $f(n) \in \Theta(g(n))$.

Anmerkung 1.1. An vorstehendem Beispiel 1.6 sehen wir, dass für Polynome

$$p(n) = \sum_{i=0}^{k} c_i n^i, \quad c_i \in \mathbb{R},\ c_k \neq 0$$

gilt

$$p(n) \in \mathcal{O}(n^k),$$

denn

$$p(n) = \sum_{i=0}^{k} c_i n^i \underset{n\in\mathbb{N}}{\leq} \sum_{i=0}^{k} |c_i| n^k = n^k \underbrace{\sum_{i=0}^{k} |c_i|}_{=:c} = cn^k.$$

Anmerkung 1.2. In der Praxis definieren wir nicht immer eine „Vergleichsfunktion“ g zur Angabe der Klassen $\mathcal{O}$, Ω, Θ, o und ω, sondern schreiben beispielsweise für

$$\mathcal{O}(g(n)) \quad \text{mit} \quad g(n) := n^k$$

einfach

$$\mathcal{O}(n^k),$$

wie wir es in Amnerkung 1.1 schon getan haben.

Anmerkung 1.3. In Anlehnung an die Mengenlehre sind auch Schreibweisen der Form

$$g_1(n) + \mathcal{O}(g_2(n)) = \{g \in \mathcal{F} \mid g(n) = g_1(n) + f(n),\ f(n) \in \mathcal{O}(g_2(n))\}$$

für Funktionen g_1 und g_2 gebräuchlich. Die Summe ist einzeln für jede Funktion in $\mathcal{O}(g_2(n))$ zu interpretieren. Analog sind

$$\mathcal{O}(g_1(n))+\mathcal{O}(g_2(n)) = \{g \in \mathcal{F} \mid g(n) = f_1(n)+f_2(n),\ f_1(n) \in \mathcal{O}(g_1(n)),\ f_2(n) \in \mathcal{O}(g_2(n))\}$$

und

$$\mathcal{O}(g_1(n))\cdot\mathcal{O}(g_2(n)) = \{g \in \mathcal{F} \mid g(n) = f_1(n)\cdot f_2(n),\ f_1(n) \in \mathcal{O}(g_1(n)),\ f_2(n) \in \mathcal{O}(g_2(n))\}$$

sowie

$$c \cdot \mathcal{O}(g_1(n)) = \{g \in \mathcal{F} \mid g(n) = c \cdot f(n),\ f(n) \in \mathcal{O}(g_1(n))\}$$

mit $c \in \mathbb{R}_0^+$ zu verstehen.

Anmerkung 1.4. In Analogie zum Vergleich zweier reeller Zahlen $a, b \in \mathbb{R}$ können wir das asymptotische Wachstumsverhalten der verschiedenen Funktionsklassen wie folgt interpretieren:

$$\begin{array}{lll} f \in \mathcal{O}(g) & \text{entspricht} & a \leq b \\ f \in \Omega(g) & \text{entspricht} & a \geq b \\ f \in \Theta(g) & \text{entspricht} & a = b \\ f \in o(g) & \text{entspricht} & a < b \\ f \in \omega(g) & \text{entspricht} & a > b \end{array}$$

Der Vergleich hat jedoch seine Grenzen. Zwei reelle Zahlen a und b sind immer in dieser Weise miteinander vergleichbar, d.h. es gilt immer *genau einer* der drei Fälle

$$a < b \qquad \text{oder} \qquad a = b \qquad \text{oder} \qquad a > b.$$

Funktionen f und g können jedoch auch *nicht* im Sinne der asymptotischen Notation miteinander vergleichbar sein, wie z.B.

$$f(n) := n^4 \qquad \text{und} \qquad g(n) := \begin{cases} n^3 & \text{falls } n \text{ ungerade} \\ n^5 & \text{falls } n \text{ gerade} \end{cases}.$$

Da es für jedes $c > 0$ und für jedes $n_0 \in \mathbb{N}$ ein $n \geq n_0$, n ungerade, gibt mit $n^4 > cn^3$ ($n^3 < cn^4$) ist also

$$f(n) \notin \mathcal{O}(g(n)) \qquad (g(n) \notin \Omega(f(n))).$$

Analog gibt es für jedes $c > 0$ und für jedes $n_0 \in \mathbb{N}$ ein $n \geq n_0$, n gerade, mit $n^4 < cn^5$ ($n^5 > cn^4$), also ist auch

$$f(n) \notin \Omega(g(n)) \qquad (g(n) \notin \mathcal{O}(f(n))).$$

Somit ist folglich

$$f(n) \notin \Theta(g(n)) \quad \text{und auch} \quad g(n) \notin \Theta(f(n)).$$

Die Funktion g „springt“ hier beliebig weit (für große n) um f „herum“, deshalb ist ein asymptotischer Vergleich nicht möglich (die Funktionen haben kein miteinander vergleichbares asymptotisches Verhalten). In nachfolgender Abbildung ist dies im Zusammenhang mit den Funktionen x^3, x^4, x^5, $x \in \mathbb{R}$, noch einmal verdeutlicht.

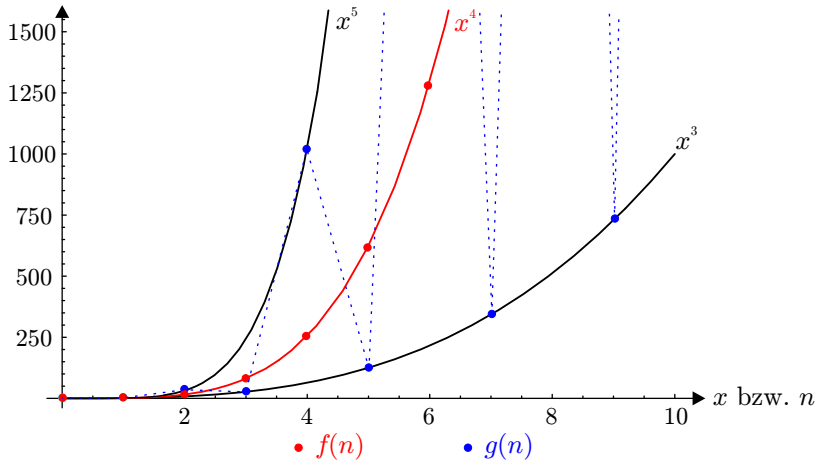

Abbildung 1.4: Asymptotischer Vergleich bzgl. f und g nicht möglich

Achtung: $f \leq g$, $f \geq g$, $f = g$, $f < g$ und $f > g$ für Funktionen f und g sind existierende Schreibweisen in der Mathematik und haben *keinen* Zusammenhang zur asymptotischen Notation. Beispielsweise bedeutet für Funktionen $f, g : \mathbb{N} \to \mathbb{R}$ die Schreibweise $f \leq g$

$$f(n) \leq g(n) \quad \text{für alle } n \in \mathbb{N}.$$

Analog sind die anderen Schreibweisen zu verstehen. Um der Verwechselungsgefahr auch im Sinne des Lesers nicht ausgesetzt zu sein, verwenden wir oben die Analogie zu zwei reellen Zahlen a und b.

Anmerkung 1.5. Oft findet man in der Literatur bei asymptotischen Abschätzungen anstelle eines '$\in$' oder gar '$\subset$' ein '$=$', was auf „einseitige Gleichungen“ führt und streng genommen sogar falsch ist. Beispielsweise schreibt man oft

$$3n^2 + 7n + 12 = 3n^2 + \Theta(n) = \Theta(n^2)$$

anstatt

$$3n^2 + 7n + 12 \in 3n^2 + \Theta(n) \subset \Theta(n^2).$$

Offensichtlich nimmt die Genauigkeit der Abschätzung von links nach rechts ab, d.h. wir dürfen diese Art „Gleichungen“ nur von links nach rechts lesen. Auch wir werden uns zuweilen dieser Schreibweise bedienen.

Hinter der Benutzung der asymptotischen Notation steht die implizite Annahme, dass von zwei Algorithmen immer der mit der kleineren asymptotischen Zuwachsrate der Laufzeit (bzw. des Speicherplatzbedarfs) als besser zu werten ist. In der Praxis ist dabei aufgrund der Konstanten c bzw. c_1 und c_2 jedoch Vorsicht geboten.

Beispiel 1.7. Algorithmus 1 habe in Abhängigkeit der Eingabelänge n eine Laufzeitfunktion

$$f_1(n) = 10^{100}n \in \mathcal{O}(n).$$

Wir sagen dann, „*Algorithmus 1 hat die Laufzeit (Zeitkomplexität)* $\mathcal{O}(n)$“. Algorithmus 2 löse das gleiche Problem und habe eine Laufzeitfunktion

$$f_2(n) = n^2 \in \mathcal{O}(n^2).$$

Offensichtlich gilt für alle $n < 10^{100}$

$$f_1(n) = 10^{100}n > n^2 = f_2(n).$$

Nach asymptotischer Abschätzung ist Algorithmus 1 (Laufzeit $\mathcal{O}(n)$) besser als Algorithmus 2 (Laufzeit $\mathcal{O}(n^2)$) anzusehen. Für alle realistischen Eingabegrößen ($n < 10^{100}$) hat jedoch Algorithmus 2 die bessere Laufzeit.

Wir sehen also:

Anmerkung 1.6. Vergleiche von Ressourcenanforderungen (Laufzeit und Speicherplatz) mittels asymptotischen Größenordnungen ($\mathcal{O}$-Notation) sind nur unter bestimmten Voraussetzungen sinnvoll. Im Allgemeinen sind das Voraussetzungen über die implizit vorhandenen Konstanten bei exakten Laufzeiten.

1.3.2 Eigenschaften

Nachfolgend zeigen wir einige Eigenschaften der asymptotischen Notation auf.

Satz 1.1. *Seien* $f, g, h \in \mathcal{F}$*, dann gilt*

(a) $f(n) \in \mathcal{O}(g(n))$ *und* $g(n) \in \mathcal{O}(h(n)) \quad \Longrightarrow \quad f(n) \in \mathcal{O}(h(n))$ *(Transitivität)*

(b) $f(n) \in \mathcal{O}(f(n))$ *(Reflexivität)*

Beweis.

(a) Da $f(n) \in \mathcal{O}(g(n))$ existieren $c_f > 0$ und $n_f > 0$ so dass für alle $n \geq n_f$ gilt

$$f(n) \leq c_f(g(n)).$$

Da $g(n) \in \mathcal{O}(h(n))$ existieren $c_g > 0$ und $n_g > 0$ so dass für alle $n \geq n_g$ gilt

$$g(n) \leq c_g(h(n)).$$

Mit $c := c_f \cdot c_g$ und $n_0 := \max\{n_f, n_g\}$ gilt dann für alle $n \geq n_0$

$$f(n) \leq c_f \cdot g(n) \leq c_f \cdot c_g \cdot h(n) = c \cdot h(n),$$

also ist $f(n) \in \mathcal{O}(h(n))$.

(b) Wegen $f(n) \leq 1 \cdot f(n)$ für alle $n \in \mathbb{N}$ ist die Aussage trivial.

■

Anmerkung 1.7. Satz 1.1 gilt auch wenn man „$\mathcal{O}$“ durch „Ω“ oder „Θ“ ersetzt.
Für Θ gilt darüber hinaus noch eine weitere Eigenschaft.

Satz 1.2. *(Symmetrieeigenschaft von Θ)*
Seien $f, g \in \mathcal{F}$, dann gilt

$$f(n) \in \Theta(g(n) \quad \Longleftrightarrow \quad g(n) \in \Theta(f(n)).$$

Beweis. Übung. ■

Anmerkung 1.8.
Die analoge Aussage zu Satz 1.2 bzgl. „$\mathcal{O}$“ und „Ω“ an Stelle von „Θ“ gilt *nicht*.

Von mathematischem Standpunkt ist folgende Aussage interessant, die wir jedoch in diesem Rahmen nur erwähnen wollen und nicht weiter beachten werden.

Anmerkung 1.9.
Auf $\mathcal{F}$ wird aufgrund der Transitivität, der Reflexivität und der Symmetrie durch $\sim$ mit

$$f \sim g \quad :\Longleftrightarrow \quad f \in \Theta(g)$$

für $f, g \in \mathcal{F}$ eine *Äquivalenzrelation* definiert.

Lemma 1.1. *Seien $g, h \in \mathcal{F}$ und $c > 0$, dann gilt*

(a) $\mathcal{O}(g(n)) + \mathcal{O}(h(n)) \subseteq \mathcal{O}(\max\{g(n), h(n)\})$

(b) $c \cdot \mathcal{O}(g(n)) \subseteq \mathcal{O}(g(n))$

(c) $\mathcal{O}(g(n)) \cdot \mathcal{O}(h(n)) \subseteq \mathcal{O}(g(n) \cdot h(n))$

Beweis.

(a) Seien $f_g \in \mathcal{O}(g)$ und $f_h \in \mathcal{O}(h)$ beliebig. Zu f_g existieren nach Definition 1.4 $c_{f_g} > 0$ und $n_{f_g} > 0$ so dass für alle $n \geq n_{f_g}$

$$f_g(n) \leq c_{f_g} g(n)$$

gilt. Ebenso existieren zu f_h analog $c_{f_h} > 0$ und $n_{f_h} > 0$ so dass für alle $n \geq n_{f_h}$

$$f_h(n) \leq c_{f_h} h(n)$$

gilt. Damit erhalten wir für alle $n \geq \max\{n_{f_g}, n_{f_h}\}$

$$\begin{aligned} f_g(n) + f_h(n) &\leq c_{f_g} g(n) + c_{f_h} h(n) \\ &\leq \max\{c_{f_g}, c_{f_h}\} \cdot (g(n) + h(n)) \\ &\leq \max\{c_{f_g}, c_{f_h}\} \cdot (\max\{g(n), h(n)\} + \max\{g(n), h(n)\}) \\ &= 2 \max\{c_{f_g}, c_{f_h}\} \cdot \max\{g(n), h(n)\} \end{aligned}$$

also $f_g(n) + f_h(n) \in \mathcal{O}(\max\{g(n), h(n)\})$.

(b) Sei $f(n) \in c \cdot \mathcal{O}(g(n))$ beliebig, d.h. es existiert $f_g(n) \in \mathcal{O}(g(n))$ mit $f(n) = c \cdot f_g(n)$. Dann existieren zu f_g Zahlen $c_{f_g} > 0$ und $n_{f_g} > 0$ so dass für alle $n \geq n_f$

$$f_g(n) \leq c_{f_g} \cdot g(n)$$

gilt, also

$$f(n) = c \cdot f_g(n) \leq c c_{f_g} \cdot g(n)$$

und somit ist $f(n) \in \mathcal{O}(g(n))$.

(c) Übung.

■