

# Theorie der Algorithmenkonstruktion

## Aufwand und asymptotische Komplexität

Foliensatz von A. Weber

Überarbeitet und ergänzt von W. Küchlin zu Informatik II, Tübingen 2004

## *Schemata für den Entwurf*

**Bottom-up-Methode:** Wir konstruieren uns Objekte und Methoden immer höherer Abstraktionsstufe bis wir die Methode haben, die unser Problem löst.

**Top-down-Methode:** Wir betten unser Problem in eine Struktur (Objektklasse) ein. Die Methoden der Struktur verwenden andere Methoden auf niedrigerer Abstraktionsstufe, so lange bis wir nur noch bereits existierende Methoden benützen.

### *Schemata für den Entwurf*

**Beispiel 10.3.1.** Bottom-up-Methode:

**Problemspezifikation:** Es sind zwei Vektoren  $v_1$  und  $v_2$  über  $\mathbb{Q}^n$  gegeben. Gesucht ist die Summe  $v_1 + v_2 \in \mathbb{Q}^n$ .

Wir wollen einen Algorithmus konstruieren, welcher zwei Vektoren aus  $\mathbb{Q}^n$  addiert. Dafür implementieren wir Schritt für Schritt:

- Die Addition, Multiplikation und Division über den ganzen Zahlen  $\mathbb{Z}$  (Integer-Werten). Diese sind normalerweise in der Programmiersprache bereits vorhanden.
- Einen Datentyp `Rational` für die Elemente aus  $\mathbb{Q}$ , zum Beispiel als Paar von Integern (Zähler, Nenner).
- Die Addition auf dem Datentyp `Rational`. Dafür brauchen wir die Addition, Multiplikation und Division auf Integer-Werten.
- Einen Datentyp `Vektor` über `Rational`.
- Die Addition von zwei Vektoren aus  $\mathbb{Q}^n$ . Dafür benutzen wir die Addition auf `Rational`.

### Beispiel 10.3.2. Top-down-Methode:

**Problemspezifikation** Es sind zwei  $n \times n$ -Matrizen  $A, B$  über den komplexen Zahlen  $\mathbb{C}$  gegeben. Gesucht ist die Matrix  $S$ , welche als Summe der beiden Matrizen definiert ist:  $S := A + B$ .

Wir konstruieren einen Algorithmus, welcher zwei komplexe  $n \times n$ -Matrizen addiert. Dafür implementieren wir:

- Einen Datentyp `Matrix` über den komplexen Zahlen.
- Die Addition zweier Elemente aus `Matrix`.
- Einen Datentyp `Complex` für die Darstellung der komplexen Zahlen  $\mathbb{C}$ , zum Beispiel als Paar (Realteil, Imaginärteil).
- Die Addition von Elementen aus `Complex`. Dafür benötigen wir die Addition von `Float`-Werten. Diese ist normalerweise in der Programmiersprache schon vordefiniert.

## Schemata für den Entwurf

**Definition 10.3.3.** *Das Greedy-Schema verläuft in folgenden Schritten:*

1. *Behandle einfache und triviale Fälle.*
2. *Reduziere das Problem in einer Richtung.*
3. *Rekursiver Aufruf.*

**Definition 10.3.4.** *Das Divide-and-Conquer-Schema verläuft in folgenden Schritten:*

1. *Behandle einfache und triviale Fälle.*
2. *Teile: reduziere das Problem in zwei oder mehrere Teilprobleme.*
3. *Herrsche: löse die Teilprobleme (typischerweise rekursiv).*
4. *Kombiniere: setze die Teillösungen zur Gesamtlösung zusammen.*

- Beispiel: Exponentiation  $x^y$ 
  - Greedy

$$x^y = x \cdot x^{y-1}$$

```
power(x, y) ≡  
if (y = 0) then 1  
else x*power(x, y-1) fi .
```

- Divide & Conquer

$$x^y = x^{\frac{y}{2}} \cdot x^{\frac{y}{2}}$$

```
power(x, y) ≡  
if (y == 0) then 1  
else if even(y) then power(x, y/2) * power(x, y/2)  
else x * power(x, y-1) fi .
```

## Aufwand und Asymptotische Komplexität

- Beispiel: Exponentiation  $x^y$

- Greedy:  $y$  Multiplikationen erforderlich

```
power(x, y) ≡  
if (y = 0) then 1  
else x*power(x, y-1) fi .
```

- Divide & Conquer:

- im besten Fall: “ungefähr”  $\log_2(y)$  Multiplikationen
- im schlimmsten Fall. “ungefähr”  $2 \cdot \log_2(y)$  Multiplikationen

```
power(x, y) ≡  
if (y == 0) then 1  
else if even(y) then power(x, y/2) * power(x, y/2)  
else x * power(x, y-1) fi .
```

- Was ist besser?

- Bsp:  $g(x, 3) \rightarrow 3$  Multipl.;  $dc(x, 3) \rightarrow 4$  Mult.

## Aufwand und asymptotische Komplexität

### • Beispiel

**Beispiel 10.4.3.** Diese nicht-rekursive Methode berechnet die Potenz  $m^n$  nach dem Divide-and-Conquer-Schema. Wir wollen bestimmen, wieviele Multiplikationen (in Abhängigkeit von  $n$ ) dazu nötig sind.

```
int power(int m, int n)
{
    int res=1; // Initialize
    while(n > 0){
        if(n % 2 == 1){
            res = res * m;
            n = n-1;
        }
        else {
            m = m * m;
            n = n/2;
        }
    }
    return res;
}
```

$n$	$M(n)$
1	1
2	2
3	3
4	3
5	4
6	4
7	5
8	4
9	5
10	5

$n$	$M(n)$
11	6
12	5
13	6
14	6
15	7
16	5
17	6
18	6
19	7
20	6



## Aufwand und asymptotische Komplexität

- Analyse

- die Anzahl der Multiplikationen ist  $(S-1)+E$ , wobei  $S$  die Anzahl der Stellen und  $E$  die Anzahl der Einsen in der Dualzahl  $n$  ist.

Am wenigsten Multiplikationen werden gebraucht, wenn  $n$  eine Zweierpotenz ( $n = 2^k$ ) ist, da dann nur eine einzige Eins gelöscht wird und somit die wenigsten Schleifendurchgänge ausgeführt werden müssen. Die Dualzahl  $n$  hat dann  $k$  Nullen und eine Eins und somit gilt

$$M_{\text{best}}(n) = M(2^k) = k + 1 = \lfloor \log_2(n) \rfloor + 1 \quad .$$

Der schlimmste Fall tritt ein, wenn  $n$  um eins kleiner als eine Zweierpotenz ist ( $n = 2^{k+1} - 1$ ), denn dann kommt zum besten Fall die Löschung von  $k$  Einsen hinzu:

$$\begin{aligned} M_{\text{worst}}(n) &= M(2^{k+1} - 1) = 1 + M(2^{k+1} - 2) = 2 + M(2^k - 1) \\ &= 3 + M(2^k - 2) = 4 + M(2^{k-1} - 1) \\ &= 5 + M(2^{k-1} - 2) = \dots = 2k + M(1) \\ &= 2k + 1 = 2\lfloor \log_2(n) \rfloor + 1 \quad . \end{aligned}$$

## Aufwand und asymptotische Komplexität

- **Weiteres Beispiel**

**Beispiel 10.4.2.** Wir betrachten folgende als Programmfragment gegebene Methode `f1`, die  $1! \cdot 2! \cdot \dots \cdot (n-3)! \cdot (n-2)!$  berechnet.

- Exakte Bestimmung der Komplexität

```
int f1 (int n) {
    int res = 1; // Init
    for(int j=1; j<n; j++)
        for(int i=1; i<j; i++)
            res = res * i;

    return res;
}
```

$n$	$Z(n)$	$V(n)$	$M(n)$	$I(n)$
1	2	0	0	0
2	3	1	0	1
3	5	3	1	3
4	8	6	3	6
5	12	10	6	10
6	17	15	10	15
7	23	21	15	21
8	30	28	21	28
9	38	36	28	36
10	47	45	36	45

# Aufwand und asymptotische Komplexität

$$\begin{aligned}M(n) &= (n-2) + M(n-1) = (n-2) + (n-3) + M(n-4) \\ &= \sum_{k=1}^{n-2} k = \frac{(n-1)(n-2)}{2} .\end{aligned}$$

Für die Anzahl der Inkrementierungen gilt  $I(n) = (n-2) + 1 + I(n-1)$ , woraus folgt:

$$I(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} .$$

Die Anzahl der Vergleiche ist gleich der Anzahl der Inkrementierungen:

$$V(n) = I(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} .$$

Die Anzahl benötigter Zuweisungen  $Z(n)$  ist gleich

$$Z(n) = 1 + 1 + I(n) = 2 + \frac{n(n-1)}{2} .$$

## Aufwand und asymptotische Komplexität

**Definition 10.4.4.** Sei  $f : \mathbb{N} \rightarrow \mathbb{R}^*$ . Die *Ordnung* von  $f$  (the order of  $f$ ) ist die Menge

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

## Aufwand und asymptotische Komplexität

**Definition 10.4.4.** Sei  $f : \mathbb{N} \rightarrow \mathbb{R}^*$ . Die *Ordnung* von  $f$  (the order of  $f$ ) ist die Menge

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

**Definition 10.4.11 („Omega“).** Für eine Funktion  $f : \mathbb{N} \rightarrow \mathbb{R}^*$  ist die Menge  $\Omega$  wie folgt definiert:

$$\Omega(f(n)) = \{t : \mathbb{N} \rightarrow \mathbb{R}^* \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : t(n) \geq c \cdot f(n)\}$$

## Aufwand und asymptotische Komplexität

**Definition 10.4.4.** Sei  $f : \mathbb{N} \rightarrow \mathbb{R}^*$ . Die *Ordnung* von  $f$  (the order of  $f$ ) ist die Menge

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

**Definition 10.4.11** („Omega“). Für eine Funktion  $f : \mathbb{N} \rightarrow \mathbb{R}^*$  ist die Menge  $\Omega$  wie folgt definiert:

$$\Omega(f(n)) = \{t : \mathbb{N} \rightarrow \mathbb{R}^* \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : t(n) \geq c \cdot f(n)\}$$

**Definition 10.4.12** („Theta“). Die exakte Ordnung  $\Theta$  von  $f(n)$  ist definiert als:

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

## Aufwand und asymptotische Komplexität

- Beispiel für Wachstum von Funktionen

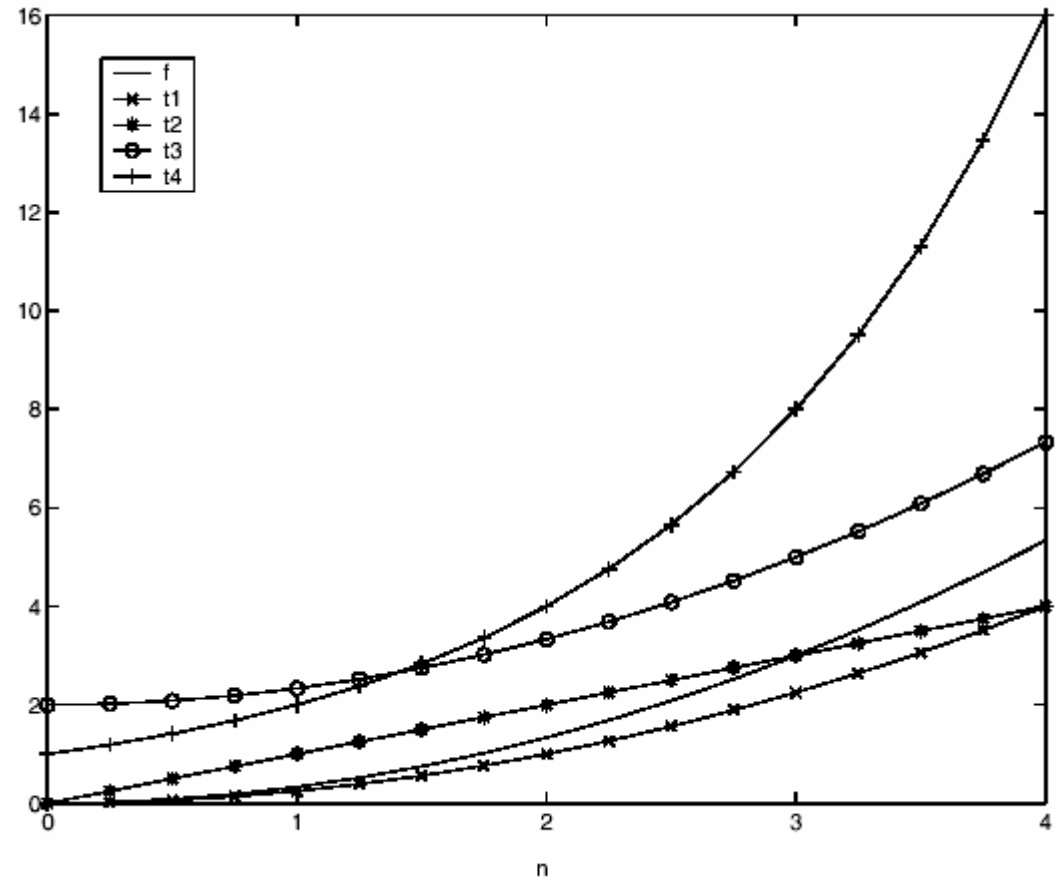
Wir wählen  $f(n) = \frac{1}{3}n^2$ . Es sei

$$t_1(n) = \frac{1}{4}n^2,$$

$$t_2(n) = n,$$

$$t_3(n) = \frac{1}{3}n^2 + 2,$$

$$t_4(n) = 2^n.$$



## Aufwand und asymptotische Komplexität

### Beispiel 10.4.9.



- Wir wollen untersuchen, ob  $n^2 \in O(n^3)$ . Dafür müssen wir zeigen, daß es ein  $c \in \mathbb{R}^+$  und ein  $n_0 \in \mathbb{N}$  gibt, so daß für alle  $n \geq n_0$  gilt:  $n^2 \leq c \cdot n^3$ . Dies gilt nur, wenn es ein  $c \in \mathbb{R}^+$  und ein  $n_0 \in \mathbb{N}$  gibt, so daß für alle  $n \geq n_0$  gilt:  $1 \leq c \cdot n$ . Solche Werte  $c$  und  $n$  gibt es: Zum Beispiel erfüllen  $c = 1$  und  $n_0 = 1$  diese Bedingung.
- Wir wollen untersuchen, ob  $n^3 \in O(n^2)$  gilt. Dies ist richtig, falls es ein  $c \in \mathbb{R}^+$  und ein  $n_0 \in \mathbb{N}$  gibt, so daß für alle  $n \geq n_0$  gilt, daß  $n^3 \leq cn^2$ . Dies gilt nur, wenn es ein  $c \in \mathbb{R}^+$  und ein  $n_0 \in \mathbb{N}$  gibt, so daß für alle  $n \geq n_0$  gilt, daß  $n \leq c$ . Nach dem Satz des Archimedes (siehe Satz 10.4.5) kann es kein solches  $c$  geben! Die Annahme  $n^3 \in O(n^2)$  führt also zu einem Widerspruch, d. h. es gilt  $n^3 \notin O(n^2)$ .





**Lemma 10.4.1.** *Für beliebige Funktionen  $f$  und  $g$  gilt:*

■ 
$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$



## Folien zu Kap. 10: Theorie der Algorithmenkonstruktion

---

**Lemma 10.4.1.** *Für beliebige Funktionen  $f$  und  $g$  gilt:*

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

**Beweis:** Wir wollen beweisen, daß die beiden Mengen gleich sind. Dafür müssen wir zeigen, daß jede der beiden Mengen jeweils in der anderen Menge enthalten ist.

## Folien zu Kap. 10: Theorie der Algorithmenkonstruktion

**Lemma 10.4.1.** Für beliebige Funktionen  $f$  und  $g$  gilt:

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

**Beweis:** Wir wollen beweisen, daß die beiden Mengen gleich sind. Dafür müssen wir zeigen, daß jede der beiden Mengen jeweils in der anderen Menge enthalten ist.

„ $\subseteq$ “: Sei  $t(n) \in O(f(n) + g(n))$ . Wir zeigen, daß  $t(n)$  auch in der Menge  $O(\max\{f(n), g(n)\})$  ist. Damit  $t(n) \in O(f(n) + g(n))$  gilt, muß es ein  $c \in \mathbb{R}^+$  und ein  $n_0 \in \mathbb{N}$  geben, so daß für alle  $n \geq n_0$  gilt

$$t(n) \leq c \cdot (f(n) + g(n)) \quad .$$

## Folien zu Kap. 10: Theorie der Algorithmenkonstruktion

**Lemma 10.4.1.** Für beliebige Funktionen  $f$  und  $g$  gilt:

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

**Beweis:** Wir wollen beweisen, daß die beiden Mengen gleich sind. Dafür müssen wir zeigen, daß jede der beiden Mengen jeweils in der anderen Menge enthalten ist.

„ $\subseteq$ “: Sei  $t(n) \in O(f(n) + g(n))$ . Wir zeigen, daß  $t(n)$  auch in der Menge  $O(\max\{f(n), g(n)\})$  ist. Damit  $t(n) \in O(f(n) + g(n))$  gilt, muß es ein  $c \in \mathbb{R}^+$  und ein  $n_0 \in \mathbb{N}$  geben, so daß für alle  $n \geq n_0$  gilt

$$t(n) \leq c \cdot (f(n) + g(n)) \quad .$$

Dies können wir wie folgt abschätzen:

$$c \cdot (f(n) + g(n)) \leq c \cdot 2 \cdot \max\{f(n), g(n)\}$$

## Folien zu Kap. 10: Theorie der Algorithmenkonstruktion

**Lemma 10.4.1.** Für beliebige Funktionen  $f$  und  $g$  gilt:

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

**Beweis:** Wir wollen beweisen, daß die beiden Mengen gleich sind. Dafür müssen wir zeigen, daß jede der beiden Mengen jeweils in der anderen Menge enthalten ist.

„ $\subseteq$ “: Sei  $t(n) \in O(f(n) + g(n))$ . Wir zeigen, daß  $t(n)$  auch in der Menge  $O(\max\{f(n), g(n)\})$  ist. Damit  $t(n) \in O(f(n) + g(n))$  gilt, muß es ein  $c \in \mathbb{R}^+$  und ein  $n_0 \in \mathbb{N}$  geben, so daß für alle  $n \geq n_0$  gilt

$$t(n) \leq c \cdot (f(n) + g(n)) \quad .$$

Dies können wir wie folgt abschätzen:

$$c \cdot (f(n) + g(n)) \leq c \cdot 2 \cdot \max\{f(n), g(n)\}$$

Mit  $\bar{c} = 2 \cdot c$  gilt für alle  $n \geq n_0$ , daß

$$t(n) \leq \bar{c} \cdot \max\{f(n), g(n)\} \quad .$$

Also ist  $t(n) \in O(\max\{f(n), g(n)\})$ .

## Folien zu Kap. 10: Theorie der Algorithmenkonstruktion

**Lemma 10.4.1.** Für beliebige Funktionen  $f$  und  $g$  gilt:

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

**Beweis:** Wir wollen beweisen, daß die beiden Mengen gleich sind. Dafür müssen wir zeigen, daß jede der beiden Mengen jeweils in der anderen Menge enthalten ist.

„ $\subseteq$ “: Sei  $t(n) \in O(f(n) + g(n))$ . Wir zeigen, daß  $t(n)$  auch in der Menge  $O(\max\{f(n), g(n)\})$  ist. Damit  $t(n) \in O(f(n) + g(n))$  gilt, muß es ein  $c \in \mathbb{R}^+$  und ein  $n_0 \in \mathbb{N}$  geben, so daß für alle  $n \geq n_0$  gilt

$$t(n) \leq c \cdot (f(n) + g(n)) \quad .$$

„ $\supseteq$ “: Sei  $t(n) \in O(\max\{f(n), g(n)\})$ . Also existiert ein  $c \in \mathbb{R}^+$  und ein  $n_0 \in \mathbb{N}$ , so daß für alle  $n \geq n_0$  gilt:  $t(n) \leq c \cdot \max\{f(n), g(n)\}$ . Es gilt aber  $\max\{f(n), g(n)\} \leq f(n) + g(n)$ , also ist  $t(n)$  auch Element von  $O(f(n) + g(n))$ .

- $O(n^4 + n^2) = O(n^4)$
- $O(n^5 + 2^n) = O(2^n)$

**Lemma 10.4.2.** *Es gelten die folgenden Aussagen:*

1.  $O(f(n)) \subseteq O(g(n))$  genau dann, wenn  $f(n) \in O(g(n))$ .
2.  $O(f(n)) = O(g(n))$  genau dann, wenn  $f(n) \in O(g(n))$  und  $g(n) \in O(f(n))$ .
3.  $O(f(n)) \subset O(g(n))$  genau dann, wenn  $f(n) \in O(g(n))$  und  $g(n) \notin O(f(n))$ .

**Beweis:**

1. „ $\Rightarrow$ “ Da  $f(n) \in O(f(n))$  und  $O(f(n)) \subseteq O(g(n))$ , ist  $f(n)$  auch in  $O(g(n))$ .  
„ $\Leftarrow$ “ Wenn  $f(n) \in O(g(n))$  ist, dann gibt es ein  $c \in \mathbb{R}^+$  und ein  $n_0 \in \mathbb{N}$ , so daß für alle  $n \geq n_0$  gilt  $f(n) \leq c \cdot g(n)$ .

Sei  $t(n)$  in  $O(f(n))$ . Dann gibt es ein  $c' \in \mathbb{R}^+$  und ein  $n'_0 \in \mathbb{N}$ , so daß für alle  $n \geq n'_0$  gilt  $t(n) \leq c' \cdot f(n)$ . Wir wählen  $n''_0 := \max\{n'_0, n_0\}$  und  $c'' := c \cdot c'$ . Dann gilt für alle  $n \geq n''_0$ :

$$t(n) \leq c'' \cdot g(n).$$

Also ist  $t(n) \in O(g(n))$ , d. h.  $O(f(n)) \subseteq O(g(n))$ .

2. Folgt direkt aus 1.

- 3. Folgt direkt aus 1 und 2.



- $O(n^2) = \{n^2, 2n^2 - 6, 3n^2 + 5, 0.5n^2 + 1, \dots, n, 2n, 3n, \dots, n^{1/2}, \dots, \log(n), \dots, 1/x, \dots\}$
- Damit:  $(3n^2 + 5) \in O(n^2)$
- Damit:  $O(3n^2 + 5) \subseteq O(n^2)$
- $O(3n^2 + 5) = \{n^2, 2n^2 - 6, 3n^2 + 5, 0.5n^2 + 1, \dots, n, 2n, 3n, \dots, n^{1/2}, \dots, \log(n), \dots, 1/x, \dots\}$
- Damit:  $(n^2) \in O(3n^2 + 5)$
- Damit:  $O(n^2) \subseteq O(3n^2 + 5)$
- Damit:  $O(n^2) = O(3n^2 + 5)$

## Aufwand und asymptotische Komplexität

**Lemma 10.4.3.** Falls  $f(n) \in O(g(n))$  und  $g(n) \in O(h(n))$ , dann ist auch  $f(n) \in O(h(n))$ .

**Beweis:** Falls  $f(n) \in O(g(n))$  und  $g(n) \in O(h(n))$ , dann gibt es Konstanten  $c', c'' \in \mathbb{R}^+$  und  $n'_0, n''_0 \in \mathbb{N}$ , so daß für alle  $n \geq n_0 := \max\{n'_0, n''_0\}$  gilt:

$$f(n) \leq c' \cdot g(n) \leq c' \cdot c'' \cdot h(n).$$

Also gilt  $f(n) \in O(h(n))$ . ■

Es gibt positive Funktionen  $f$  und  $g$ , so daß  $f(n) \notin O(g(n))$  und auch  $g(n) \notin O(f(n))$ .  
Wähle zum Beispiel  $f(n) = \sin(n) + 1$  und  $g(n) = \cos(n) + 1 + 2^{(-n)}$   
 $+ 2^{(-n)}$

- $O(n^2) = O(3n^2) = O(0.5n^2)$
- $O(n^2) \mu O(3n^2) \mu O(0.5n^2)$
  
- $O(n^2) \mu O(n^{2,5}) \mu O(n^3)$
- $O(n^2) \frac{1}{2} O(n^{2,5}) \frac{1}{2} O(n^3)$

## Aufwand und asymptotische Komplexität

**Lemma 10.4.4.** Für alle  $m \in \mathbb{N}$  gilt  $O(n^m) \subseteq O(n^{m+1})$ .

**Beweis:** Der Beweis erfolgt durch Induktion nach  $m$ . Der Induktionsschritt erfolgt analog zu der Rechnung in Bsp. 10.4.9; die Details stellen wir als Übung. ■

## Aufwand und asymptotische Komplexität

**Satz 10.4.10.** Sei  $A(n) := a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$ , wobei  $a_i \in \mathbb{R}^*$  für  $0 \leq i \leq m$ . Dann gilt  $A(n) \in O(n^m)$ .

Wir sagen, ein Algorithmus  $A$  mit Komplexität  $f(n)$  braucht höchstens **polynomielle Rechenzeit** (*polynomial time*), falls es ein Polynom  $p(n)$  gibt, so daß  $f(n) \in O(p(n))$ .  $A$  braucht höchstens **exponentielle Rechenzeit** (*exponential time*), falls es eine Konstante  $a \in \mathbb{R}^+$  gibt, so daß  $f(n) \in O(a^n)$ .

- Allgemein:  $O(A(n)) \simeq O(a^n)$
- Beispiel:  $O(2n^9 + 3n^7 + 5n^4) \simeq O(2^{n+1})$