

# Asymptotik und Laufzeitanalyse

Vorkurs Informatik  
SoSe14

09. April 2014

# Laufzeitanalyse

Algorithmen = Rechenvorschriften

Wir fragen uns:

- Ist der Algorithmus effizient?
- welcher Algorithmus löst das Problem schneller?
- wie lange braucht der Algorithmus noch?

# Ziel

## Ziel

Die Laufzeit von Algorithmen verlässlich voraussagen.

Die Laufzeit hängt ab von:

- Eingabe
  - Größe
  - Struktur (z.B. vorsortierte Liste)
- Hardware
  - Architektur
  - Taktfrequenz
- Software
  - Betriebssystem
  - Programmiersprache
  - Interpreter/Compiler/Assembler

# Laufzeit

- Angabe der Laufzeit als Funktion  $f : \mathbb{N} \rightarrow \mathbb{R}^+$  der Eingabegröße  $n$ .

## Beispiel

$$T_A(n) = 2n + 1$$

$$T_B(n) = \frac{1}{2}n^2 + 5$$

- unabhängig von Hardware und Software
- Laufzeiten nach Wachstumsverhalten klassifizieren.  
⇒ auf das Wesentliche beschränken

# Asymptotik

# Was ist das?

## Definition (Asymptotische Analyse)

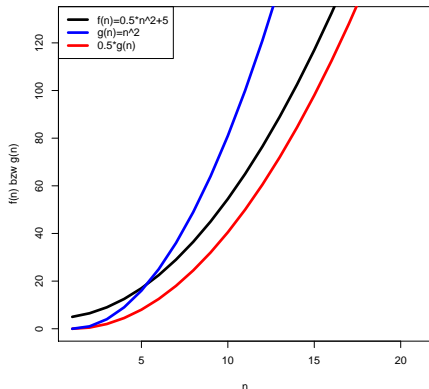
*Methode um das Grenzverhalten von Funktionen zu klassifizieren, indem man nur den wesentlichen Trend des Grenzverhaltens beschreibt.*

- wir ordnen Funktionen in „Klassen“
- mit Hilfe der Betrachtung des Wesentlichen

Was ist das Wesentliche?

# Asymptotisch gleiches Wachstum

$\Theta(g) := \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \text{es gibt Konstanten } c_1 > 0 \text{ und } c_2 > 0 \text{ und } n_0 \in \mathbb{N}, \text{ so dass für alle } n \geq n_0 \text{ gilt: } c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}.$



$f = \Theta(g)$ , da für  $c_1 = 0.5$ ,  $c_2 = 1$  und  $n_0 = 5$  gilt:  
 $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$

# Asymptotisches Wachstum

## Definition (asymptotisch gleiches Wachstum: $\Theta$ )

Seien  $f$  und  $g$  Funktionen  $\mathbb{N} \rightarrow \mathbb{R}^+$ , und der Grenzwert der Folge  $\frac{f(n)}{g(n)}$  möge existieren. Dann ist:

$$f = \Theta(g) :\Leftrightarrow 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

- wir betrachten das Wachstum für **große** Werte von  $n$   
⇒ Konstanten werden uninteressant



# $\mathcal{O}$ -Notation

## Definition (Groß-O)

Seien  $f$  und  $g$  Funktionen  $\mathbb{N} \rightarrow \mathbb{R}^+$ ,

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

in Worten:

- $\mathcal{O}(g)$  umfasst alle Funktionen  $f$  für die gilt: es existiert eine positive Konstante  $c$  und eine natürliche Zahl  $n_0$ , so dass  $f(n) \leq c \cdot g(n)$ , für alle  $n \geq n_0$  gilt.

oder:

- die Funktionswerte von  $f$  sind ab einem gewissen  $n_0 \leq$  **einem Vielfachen** von  $g$ .

$\mathcal{O}(g)$  ist eine Menge/Klasse von Funktionen  
wir schreiben trotzdem:  $f = \mathcal{O}(g)$

# Asymptotisches Wachstum

Definition (asymptotisch langsamer/schnelleres Wachstum „ $\prec$ “)

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

„ $f$  wächst asymptotisch langsamer als  $g$ “

Transitivität: „ $g$  wächst asymptotisch schneller als  $f$ “

**Man schreibt auch:**  $f = o(g)$  (sprich: „klein-o“)

# Grenzwerte

## Definition ( $\mathcal{O}$ -Notation über Grenzwerte)

Möge der Grenzwert der Folge  $\frac{f(n)}{g(n)}$  existieren dann ist:

- $f = \mathcal{O}(g) : \Leftrightarrow 0 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$ ;  
*f wächst höchstens so schnell wie g*
- $f = \Theta(g) : \Leftrightarrow 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$ ;  
*f wächst genau so schnell wie g*
- $f = o(g) : \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ ;  
*f wächst langsamer als g*

$$f = \Theta(g) \Leftrightarrow f = \mathcal{O}(g) \text{ und } g = \mathcal{O}(f)$$

# asymptotische Hackordnung

für beliebige Konstanten  $0 < \epsilon < 1 < c$  gilt:

$$\underbrace{1} \prec \underbrace{\log \log n \prec \log n} \prec \underbrace{n^\epsilon \prec n \prec n^c} \prec \underbrace{n^{\log n} \prec c^n \prec n! \prec n^n \prec c^{c^n}}$$

konstant

logarithmisch

polynomiell

exponentiell

- hilfreiche innerliche Grundhaltung:

## Denke im Großen!

# Beispiele

- $1000n = \mathcal{O}(n)$
- $n^2 + 500n = \mathcal{O}(n^2)$
- $\log_a n = \mathcal{O}(\log_b n)$

## Lehrsätze:

Die  $\mathcal{O}$ -Notation,...

- ... verschluckt in einem Produkt konstante Faktoren
- ... verschluckt in einer Summe mit konstant vielen Summanden, alle Summanden, außer den mit dem größten asymptotischen Wachstum
- ... unterscheidet nicht zwischen verschiedenen Basen des Logarithmus

# Laufzeitanalyse

## Warum?

Annahme: Ein einfacher Befehl benötigt  $10^{-9}$ sec

$n$	$n^2$	$n^3$	$n^{10}$	$2^n$	$n!$
16	256	4.096	$\geq 10^{12}$	65536	$\geq 10^{13}$
32	1.024	32.768	$\geq 10^{15}$	$\geq 4 \cdot 10^9$	$\geq 10^{31}$
64	4.096	262.144	$\geq 10^{18}$	$\geq 6 \cdot 10^{19}$	mehr als $10^{14}$ Jahre
128	16.384	2.097.152	mehr als	mehr als	
256	65.536	16.777.216	10 Jahre	600 Jahre	
512	262.144	134.217.728			
1024	1.048.576	$\geq 10^9$			
$10^6$	$\geq 10^{12}$	$\geq 10^{18}$			
	mehr als 15 Minuten	mehr als 10 Jahre			

# Beispiel 1

## Algorithmus (zaehle( $n$ ))

```
1 function zaehle( $n$ ){  
2   for(int  $i=1$ ;  $i \leq n$ ;  $i++$ ){  
3     print  $i$ ;  
4   }  
5 }
```

Laufzeit: Zähle die Anzahl der Befehle.

- $s_{3,3}$ : Print-Befehl  $\rightarrow 1$
- $s_{2,4}$ : for-Schleife  $\rightarrow \sum_{i=1}^n s_{3,3} = \sum_{i=1}^n 1 = n$
- $s_{1,5}$ : nichts  $\rightarrow s_{2,4} = n = O(n)$



## Beispiel 2

### Algorithmus ( $\text{maximum}(A[0..n])$ )

```
1 function maximum(A[0..n]){
2   int max = A[0];
3   for(i = 1; i ≤ n; i++) do{
4     if A[i] ≥ max{
5       max = A[i];
6     }
7   }
8   return max;
9 }
```

**Laufzeit:**  $O(n)$  Lineare Suche

## Beispiel 3

Algorithmus (insertionsort( $A[0..n]$ ))

```
1 function insert( $A[0..n]$ ){
2   for( $i = 1; i \leq n; i++$ ) do{
3     int  $x = A[i]$ ;
4     int  $j = i$ ;
5     while  $j > 0$  and  $A[j - 1] > x$  do{
6        $A[j] = A[j - 1]$ ;
7        $j = j - 1$ ;
8     }
9      $A[j] = x$ ;
10  }
11 }
```

## Beispiel 3

**Neu:** Die Laufzeit hängt davon ab, wie häufig  $s_{6,8}$  durchlaufen wird.

- Best-case Laufzeit: Die while-Schleife wird nie durchlaufen.  $A[j - 1] > x$  ist nie erfüllt;  $A[0..n]$  ist bereits sortiert
- Worst-case Laufzeit: Die while-Schleife wird immer komplett durchlaufen.  $A[j - 1] > x$  ist immer erfüllt;  $A[0..n]$  ist absteigend sortiert

# Beispiele häufiger Laufzeiten (1)

$f = O(1)$  :  $f$  wird **nie** größer als ein konstanter Wert;  
z.B. Zugriff auf das  $i$ -te Element eines Arrays.

$f = O(\log n)$  :  $f$  wächst ungefähr um einen konstanten Betrag, wenn sich die Eingabelänge verdoppelt.  
Teile-und-Herrsche-Prinzip; z.B. Binärsuche

$f = O(n)$  :  $f$  wächst ungefähr auf das Doppelte, wenn sich die Eingabelänge verdoppelt.  
jede Eingabestelle sehen; z.B. Lineare Suche

$f = O(n^2)$  :  $f$  wächst ungefähr auf das vierfache, wenn sich die Eingabelänge verdoppelt.  
z.B. einfache Sortieralgorithmen wie Selection Sort

## Beispiele häufiger Laufzeiten (2)

$f = O(2^n)$  :  $f$  wächst ungefähr auf das Doppelte, wenn sich die Eingabelänge um eins erhöht.  
z.B. Untersuchung aller Teilmengen

$f = O(n!)$  :  $f$  wächst ungefähr auf das  $(n + 1)$ -fache, wenn sich die Eingabelänge um eins erhöht.  
z.B. Untersuchung aller Permutationen

**Summen** : Hintereinander ausführen von Schleifen

**Produkt** : geschachtelte Schleifen