



Algorithmen und Datenstrukturen

Effizienz und Funktionenklassen

Matthias Teschner
Graphische Datenverarbeitung
Institut für Informatik
Universität Freiburg

SS 09

Lernziele der Vorlesung



- Algorithmen
 - Sortieren, Suchen, Optimieren
- Datenstrukturen
 - Repräsentation von Daten
 - Listen, Stapel, Schlangen, Bäume
- Techniken zum Entwurf von Algorithmen
 - Algorithmenmuster
 - Greedy, Backtracking, Divide-and-Conquer
- **Analyse von Algorithmen**
 - Korrektheit, Effizienz

Analyse von Algorithmen



- **Korrektheit**
 - Ein korrekter Algorithmus stoppt (terminiert) für jede Eingabeinstanz mit der durch die Eingabe-Ausgabe-Relation definierten Ausgabe.
 - Ein inkorrekt Algorithmus stoppt nicht oder stoppt mit einer nicht durch die Eingabe-Ausgabe-Relation vorgegebenen Ausgabe.
- **Effizienz**
 - Bedarf an Speicherplatz und Rechenzeit
 - Wachstum (Wachstumsgrad, Wachstumsrate) der Rechenzeit bei steigender Anzahl der Eingabe-Elemente (Laufzeitkomplexität)

Überblick



- Kontext
- Einführung
- Funktionenklasse \mathcal{H}
 - Definition, Illustration, Beispiele
- weitere Funktionenklassen
 - Definition, Illustration, Beispiele
- Anwendung in der Laufzeitbeschreibung

Effizienz



- Bedeutung der Effizienz eines Algorithmus durch Begrenztheit von Ressourcen gegeben
 - Sortieralgorithmus A benötigt n^2 Schritte für n Elemente
 - Sortieralgorithmus B benötigt $n \log_2 n$ Schritte für n Elemente
 - Computer 1: 10^9 Schritte / s, Computer 2: 10^7 Schritte / s
 - Eingabegröße: 10^6 Elemente

Laufzeit:	Algorithmus A	Algorithmus B
Computer 1	17 min	0.02 s
Computer 2	2 d 8 h	2 s

Effizienz



- Laufzeitkomplexität dominiert die benötigte Berechnungszeit für große Eingaben
- durch Hard- und Software bedingte Unterschiede weniger entscheidend
- Anmerkung:
Speicherbedarf oder Kommunikationsbandbreite können auch bei der Effizienzanalyse relevant sein.
- Fokus auf Laufzeitkomplexität eines Algorithmus

Überblick



- Kontext
- Einführung
- Funktionenklasse \mathcal{H}
 - Definition, Illustration, Beispiele
- weitere Funktionenklassen
 - Definition, Illustration, Beispiele
- Anwendung in der Laufzeitbeschreibung

Maschinenmodell



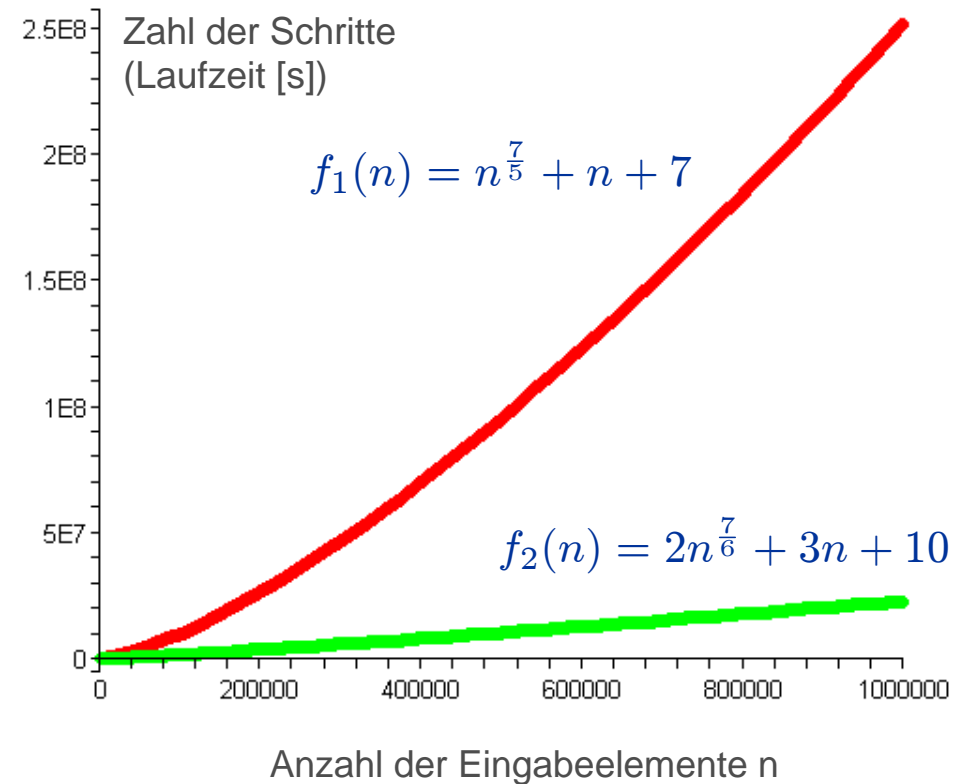
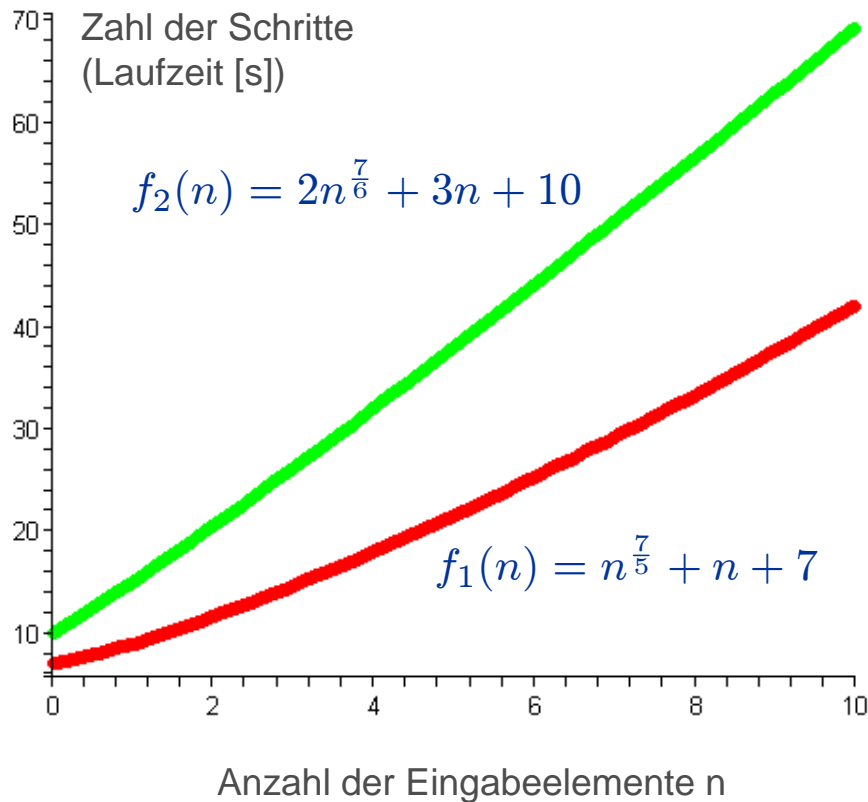
- Maschine
 - mit wahlfreiem Speicherzugriff (RAM)
 - mit einem Prozessor
 - Speicherhierarchie wird vernachlässigt (z. B. Cache)
 - kann Algorithmus als Programm ausführen
- Schritte eines Algorithmus sind zeitkonstante Anweisungen
 - werden in konstanter Zeit ausgeführt
 - Laufzeit unabhängig von der Eingabe
 - Laufzeitunterschiede einzelner Schritte werden vernachlässigt.
 - Addieren, Runden, Kopieren, bedingte Verzweigung, Aufruf eines Unterprogramms
 - Sortieren ist kein Schritt
(sinnvolle Definition von Schritten!)

Analyse der Laufzeiteffizienz



- Laufzeit hängt von der Komplexität der Eingabemenge ab
 - oft die Anzahl der Elemente n
- Laufzeit wird oft durch eine Funktion der Anzahl der Eingabeelemente $f(n)$ beschrieben
- Laufzeit von Algorithmus 1: $f_1(n) = n^{\frac{7}{5}} + n + 7$
- Laufzeit von Algorithmus 2: $f_2(n) = 2n^{\frac{7}{6}} + 3n + 10$
- Wie vergleicht man die Effizienz der Algorithmen?

Analyse der Laufzeiteffizienz



- entscheidend ist das Verhalten für große Eingaben

Wachstumsrate



- Üblicherweise interessiert nur der Term höchster Ordnung.
- Terme niedriger Ordnung werden ignoriert.
- Konstante Faktoren werden ignoriert.

$$\begin{array}{l} f_1(n) = n^{\frac{7}{5}} + n + 7 \\ f_2(n) = 2n^{\frac{7}{6}} + 3n + 10 \end{array} \Rightarrow \begin{array}{l} n^{\frac{7}{5}} \\ n^{\frac{7}{6}} \end{array}$$

- **Wachstumsrate** $n^{\frac{7}{6}} \leq n^{\frac{7}{5}}$
- Algorithmus 2 ist effizienter als Algorithmus 1.
- \Rightarrow asymptotisches Verhalten (Wie verhält sich die Funktion für Eingabegrößen gegen unendlich?)

Asymptotische Analyse



- Methode zur Einschätzung des Grenzverhaltens einer Funktion (bei uns Laufzeitverhalten für wachsende Eingabegrößen)
- Fokus auf den wesentlichen Trend des Grenzverhaltens
- ermöglicht die Einordnung von Funktionen in **Funktionsklassen**
- Beispielsweise können die **Landau-Symbole** Θ , O , Ω verwendet werden, um Klassen von Funktionen zu beschreiben.
- Funktionsklassen werden zur Beschreibung von Laufzeit bzw. Komplexität von Algorithmen verwendet (**asymptotische Effizienz eines Algorithmus**).

Überblick



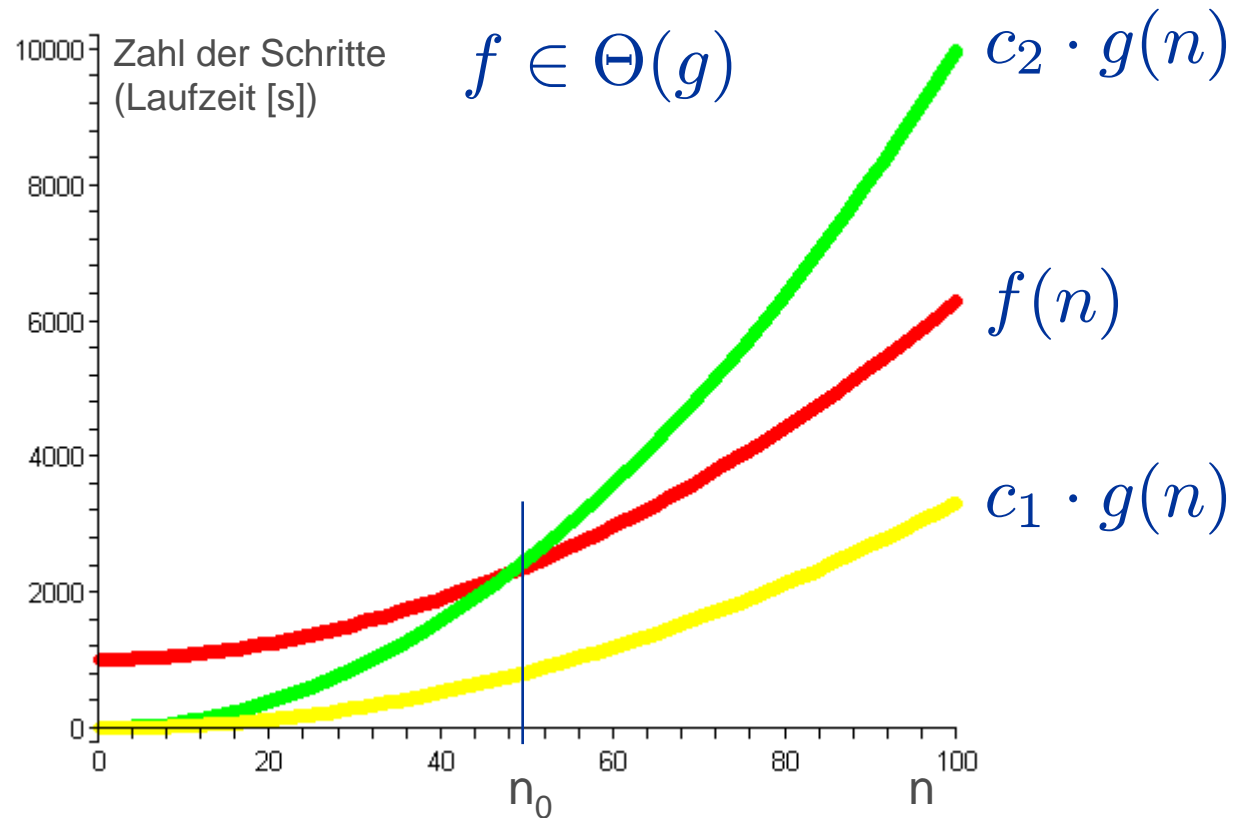
- Kontext
- Einführung
- Funktionenklasse \mathcal{H}
 - Definition, Illustration, Beispiele
- weitere Funktionenklassen
 - Definition, Illustration, Beispiele
- Anwendung in der Laufzeitbeschreibung

Landau-Symbol Θ



- $f \in \Theta(g)$:
 $\exists c_1 > 0 \quad \exists c_2 > 0 \quad \exists n_0 \quad \forall n > n_0 :$
 $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$
- f und g sind Funktionen, die Laufzeiten beschreiben:
 $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$
- f ist Element der Menge Theta von g, wenn zwei positive Konstanten c_1 und c_2 existieren, sodass der Funktionswert $f(n)$ ab einem bestimmten $n=n_0$ immer zwischen $c_1 \cdot g(n)$ und $c_2 \cdot g(n)$ liegt.
- Wie sich f und g für $n < n_0$ verhalten, spielt keine Rolle.
- Anschaulich: f wächst genauso schnell wie g.

Illustration



- f und g haben die gleiche Größenordnung.

Beispiel



- $f_1(n) = \frac{1}{2}n^2 + 3n$
 $f_1 \in \Theta(n^2)$?
- Existieren c_1 , c_2 , und n_0 , sodass für $n > n_0$ gilt:
 $c_1n^2 \leq \frac{1}{2}n^2 + 3n \leq c_2n^2$?

Beispiel



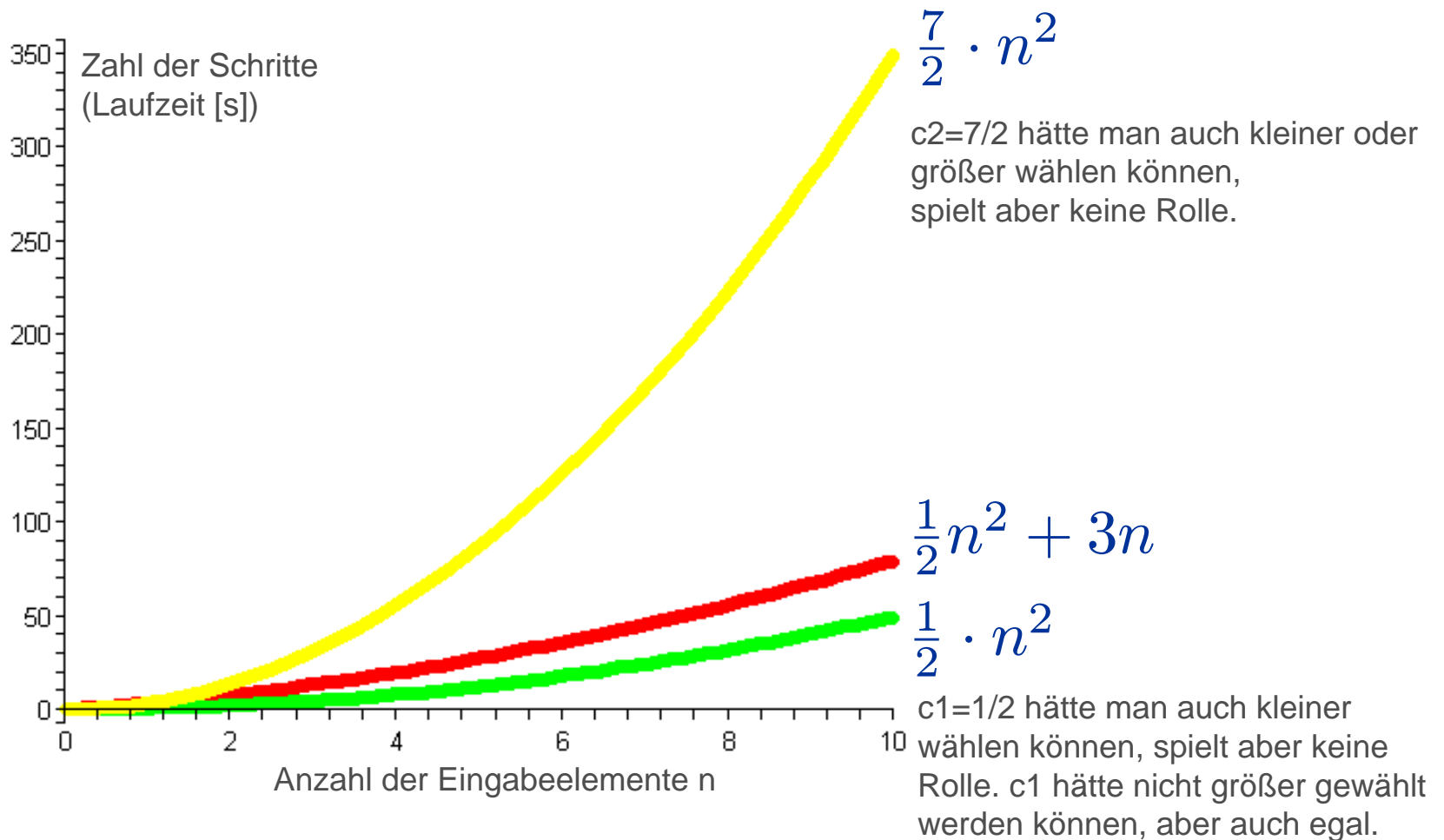
- $c_1 \leq \frac{1}{2} + \frac{3}{n} \leq c_2$
- Für $n \rightarrow \infty$: $\frac{1}{2} + \frac{3}{n} \rightarrow \frac{1}{2}$ und $\frac{1}{2} + \frac{3}{n} \geq \frac{1}{2} = c_1$
- Für $n = 1$: $\frac{1}{2} + \frac{3}{1} = \frac{7}{2} = c_2$

$$\frac{1}{2}n^2 + 3n \in \Theta(n^2),$$

da fuer $n \geq 1$ gilt:

$$\frac{1}{2} \cdot n^2 \leq \frac{1}{2}n^2 + 3n \leq \frac{7}{2} \cdot n^2$$

Beispiel



Beispiel



- $f_1(n) = n^3$
 $f_1 \in \Theta(n^2)$?
- Existieren c_1 , c_2 , und n_0 , sodass für $n > n_0$ gilt:
 $c_1 n^2 \leq n^3 \leq c_2 n^2$?
- $c_1 \leq n \leq c_2$
- Für jedes c_2 existiert ein n mit $n > c_2$.
- $n^3 \notin \Theta(n^2)$

Noch mehr Beispiele



- Es interessiert nur der Term höchster Ordnung, der am schnellsten wachsende Summand
- Terme niedriger Ordnung und Konstanten werden ignoriert.

$$2 \cdot n^2 + 7 \cdot n - 20 \in \Theta(n^2)$$

$$2 \cdot n^2 + 7 \cdot n \log n - 20$$

$$7 \cdot n \log n - 20$$

5

$$2 \cdot n^2 + 7 \cdot n \log n + n^3$$

- Laufzeit eines Algorithmus wird über Funktionenklasse beschrieben.

Typische Funktionenklassen zur Effizienzbeschreibung



$f \in \Theta(1)$	zeitkonstant, f ist beschränkt
$f \in \Theta(\log n) = \Theta(\log_2 n)$	logarithmisch wachsende Funktionen
$f \in \Theta(n)$	linear wachsende Funktionen
$f \in \Theta(n \log n)$	n-log-n wachsend, superlinear
$f \in \Theta(n^2)$	quadratisch wachsende Funktionen
$f \in \Theta(n^3)$	kubisch wachsende Funktionen
$f \in \Theta(n^k)$	polynomiell wachsende Funktionen
$f \in \Theta(2^n)$	exponentiell wachsende Funktionen

Praktische Relevanz der Funktionenklassen



- 1 Rechenschritt benötigt 0.001s

Maximale Problemgröße bei gegebener Rechenzeit und Laufzeitkomplexität			
Laufzeit / Komplexität	1 Sekunde	1 Minute	1 Stunde
n	1000	60000	3600000
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

Praktische Relevanz der Funktionenklassen



- Rechner B 10fach schneller als Rechner A

Maximale Problemgröße auf Rechner B bei gegebener Problemgröße p auf Rechner A

Laufzeit / Komplexität	Problemgröße auf Rechner B
n	$10 p$
n^2	$3.16 p$
n^3	$2.15 p$
2^n	$p + 3.32 = p + \log_2 10$

$$10 \cdot 2^p = 2^{\log_2 10} \cdot 2^p = 2^{p+\log_2 10}$$

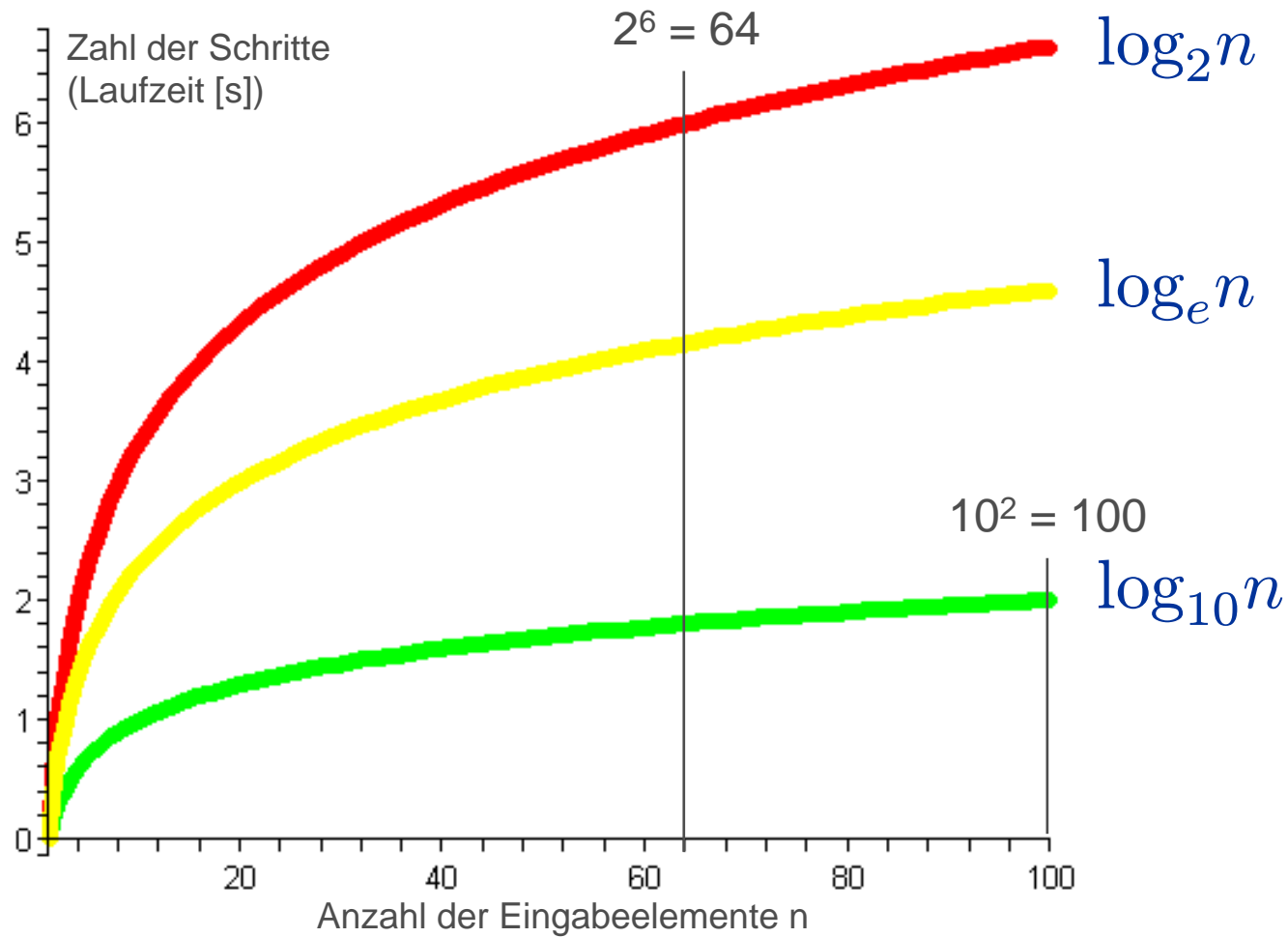
- Polynomielle Laufzeiten erlauben die Bearbeitung einer um einen Faktor größeren Problemgröße bei schnellerer Hard- / Software. Exponentielle Laufzeiten nicht.

Kurz zum Logarithmus



- Logarithmen lösen Gleichungen der Form $a = b^n$ bei gegebenen a und b : $n = \log_b a$
- Bezeichnungen:
 $\log_e a = \ln a$ $\log_{10} a = \lg a$ $\log_2 a = \lg a$
- bei $\log a$ ist die Basis aus dem Kontext klar, bei uns oft 2

Illustration



Einige Rechenregeln



- Umkehrfunktionen

$$b^{\log_b n} = n \quad \log_b (b^n) = n \quad \log_2 (256) = \log_2 (2^8) = 8$$

- Produktregel

$$\log_b (m \cdot n) = \log_b m + \log_b n$$

- Potenzregel

$$\log_b (n^r) = r \cdot \log_b n$$

- Basisumrechnung

$$\log_a n = \underbrace{\log_a b}_{\text{konstant}} \log_b n$$

- daher $\Theta(\log n) = \Theta(\log_b n)$ für beliebige Basis b und wir sind wieder beim Thema.

Zusammenfassung zu Θ



- $\Theta(g)$ beschreibt eine Klasse von Funktionen
- $f \in \Theta(g)$, wenn f bis auf einen konstanten Faktor gleich schnell wächst wie g
- g ist eine asymptotisch scharfe obere und untere Schranke von f
- f wird einer Funktionenklasse g zugeordnet, indem
 - nur der Term höchster Ordnung betrachtet wird (der am schnellsten wachsende Summand)
 - Terme niedriger Ordnung und Konstanten ignoriert werden

■ Beispiele:

$$2 \cdot n^2 + 7 \cdot n - 20 \in \Theta(n^2)$$

$$7 \cdot n \log n - 20 \in \Theta(n \log n)$$

Überblick



- Kontext
- Einführung
- Funktionenklasse \mathcal{H}
 - Definition, Illustration, Beispiele
- **weitere Funktionenklassen**
 - Definition, Illustration, Beispiele
- Anwendung in der Laufzeitbeschreibung

Landau-Symbol O

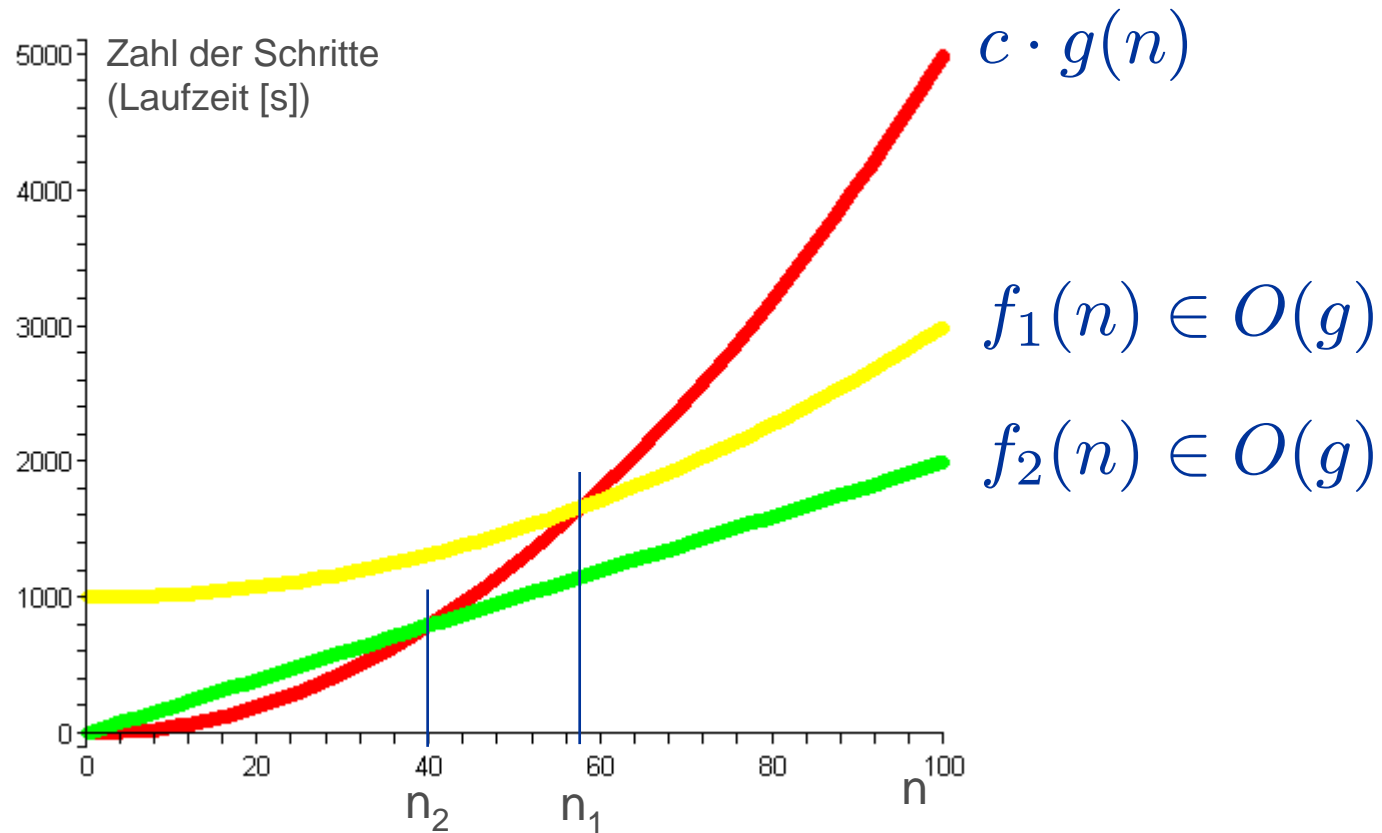


- Wenn eine Funktion nicht von oben und von unten beschränkt wird, sondern nur von oben (**obere asymptotische Schranke**), dann wird O verwendet.

$$f \in O(g) : \exists c > 0 \quad \exists n_0 \quad \forall n > n_0 : f(n) \leq c \cdot g(n)$$

- f ist Element der Menge Gross- O von g , wenn eine positive Konstante c existiert, sodass der Funktionswert $f(n)$ ab einem bestimmten $n=n_0$ immer kleiner oder gleich $c \cdot g(n)$ ist.
- Wie sich f und g für $n < n_0$ verhalten, spielt keine Rolle.
- Anschaulich: $f(n)$ wächst höchstens so schnell wie $g(n)$ bzw. $c \cdot g(n)$.

Illustration



- f ist höchstens von der Größenordnung von g .

Beispiele



- Es interessiert nur der Term höchster Ordnung, der am schnellsten wachsende Summand
- Terme niedriger Ordnung und Konstanten werden ignoriert.
- $f(n)$ muss nur von oben durch $c \cdot g(n)$ beschränkt sein.

$$2 \cdot n^2 + 7 \cdot n - 20 \in O(n^2)$$

$$2 \cdot n^2 + 7 \cdot n \log n - 20$$

$$7 \cdot n \log n - 20$$

5

$$2 \cdot n^2 + 7 \cdot n \log n + n^3$$

Landau-Symbol Ω

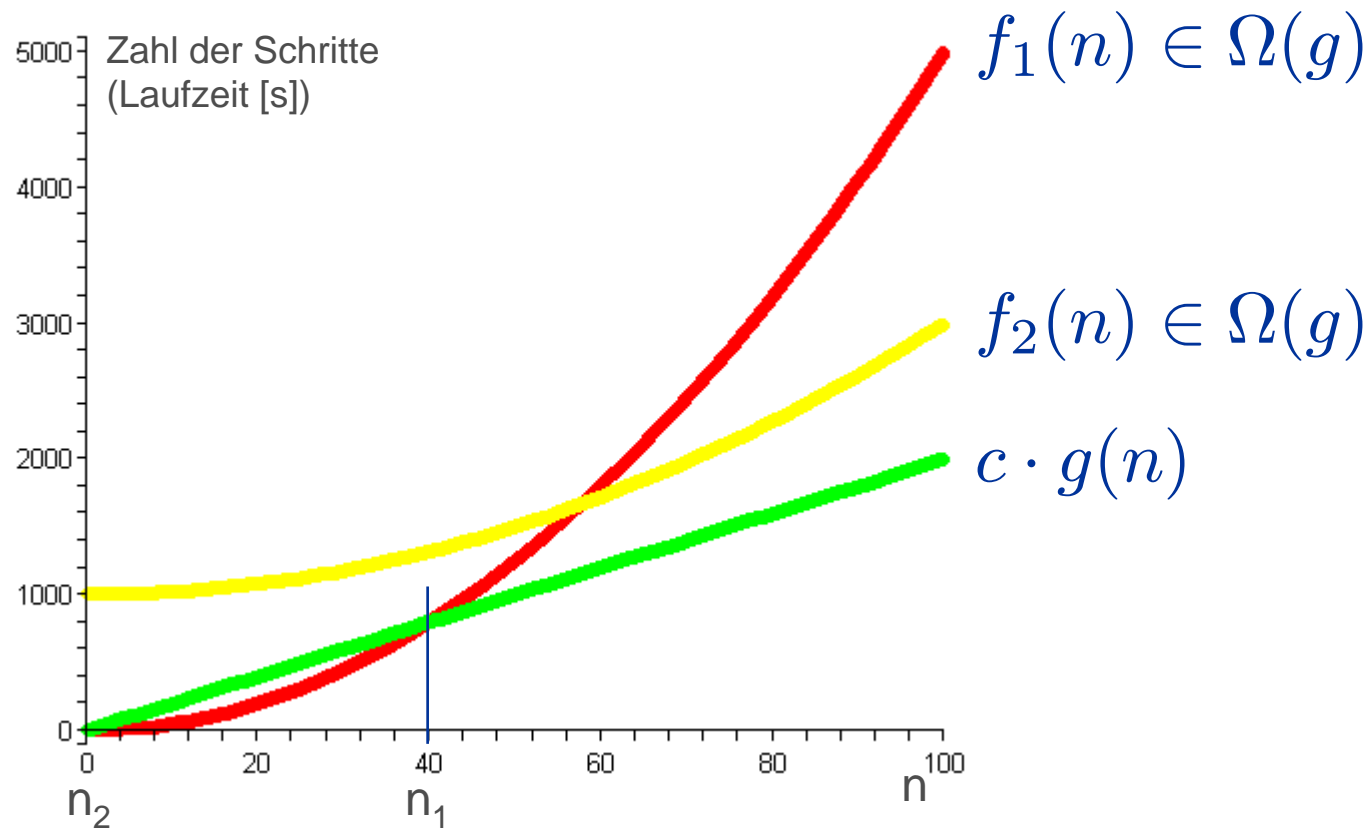


- Wenn eine Funktion nicht von oben und von unten beschränkt wird, sondern nur von unten (**untere asymptotische Schranke**), dann wird Ω verwendet.

$$f \in \Omega(g) : \exists c > 0 \quad \exists n_0 \quad \forall n > n_0 : f(n) \geq c \cdot g(n)$$

- f ist Element der Menge Gross- Ω von g , wenn eine positive Konstante c existiert, sodass der Funktionswert $f(n)$ ab einem bestimmten $n=n_0$ immer größer oder gleich $c \cdot g(n)$ ist.
- Wie sich f und g für $n < n_0$ verhalten, spielt keine Rolle.
- Anschaulich: $f(n)$ wächst mindestens so schnell wie $g(n)$ bzw. $c \cdot g(n)$.

Illustration



- f ist mindestens von der Größenordnung von g .

Beispiele



- Es interessiert nur der Term höchster Ordnung, der am schnellsten wachsende Summand
- Terme niedriger Ordnung und Konstanten werden ignoriert.
- $f(n)$ muss nur von unten durch $c \cdot g(n)$ beschränkt sein.

$$2 \cdot n^2 + 7 \cdot n - 20 \in \Omega(n^2)$$

$$2 \cdot n^2 + 7 \cdot n \log n - 20$$

$$7 \cdot n \log n - 20$$

5

$$2 \cdot n^2 + 7 \cdot n \log n + n^3$$

Landau-Symbole o und ω



- asymptotisch nicht scharfe Schranken
- obere Schranke
 - $f \in o(g) : \forall c > 0 \exists n_0 \forall n > n_0 : f(n) \leq c \cdot g(n)$
- f wächst langsamer als g (ist von kleinerer Größenordnung)
- $2 \cdot n \in o(n^2) \quad 4 \cdot n^2 \notin o(n^2)$
- untere Schranke
 - $f \in \omega(g) : \forall c > 0 \exists n_0 \forall n > n_0 : f(n) \geq c \cdot g(n)$
- f wächst schneller als g (ist von höherer Größenordnung)
- $2 \cdot n^3 \in \omega(n^2) \quad 4 \cdot n^2 \notin \omega(n^2)$

Zusammenfassung

O-Notation



$$f \in \Theta(g) : \exists c_1 > 0 \quad \exists c_2 > 0 \quad \exists n_0 \quad \forall n > n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$$f \in O(g) : \exists c > 0 \quad \exists n_0 \quad \forall n > n_0 : f(n) \leq c \cdot g(n)$$

$$f \in \Omega(g) : \exists c > 0 \quad \exists n_0 \quad \forall n > n_0 : f(n) \geq c \cdot g(n)$$

$$f \in o(g) : \forall c > 0 \quad \exists n_0 \quad \forall n > n_0 : f(n) \leq c \cdot g(n)$$

$$f \in \omega(g) : \forall c > 0 \quad \exists n_0 \quad \forall n > n_0 : f(n) \geq c \cdot g(n)$$

Zusammenfassung Eigenschaften



- Transitivität
 $f \in \Theta(g) \wedge g \in \Theta(h) \rightarrow f \in \Theta(h)$
 $f \in O(g) \wedge g \in O(h) \rightarrow f \in O(h)$
 $f \in \Omega(g) \wedge g \in \Omega(h) \rightarrow f \in \Omega(h)$
 $f \in o(g) \wedge g \in o(h) \rightarrow f \in o(h)$
 $f \in \omega(g) \wedge g \in \omega(h) \rightarrow f \in \omega(h)$
- Reflexivität
 $f \in \Theta(f) \quad f \in \Omega(f) \quad f \in O(f)$
- Symmetrie
 $f \in \Theta(g) \leftrightarrow g \in \Theta(f)$
- Austausch-Symmetrie
 $f \in O(g) \leftrightarrow g \in \Omega(f)$
 $f \in o(g) \leftrightarrow g \in \omega(f)$

Überblick



- Kontext
- Einführung
- Funktionenklasse \mathcal{H}
 - Definition, Illustration, Beispiele
- weitere Funktionenklassen
 - Definition, Illustration, Beispiele
- Anwendung in der Laufzeitbeschreibung

Uns reicht oft O



- Laufzeiten werden oft mit O beschrieben (Vernachlässigung der unteren Schranke).
- Oft wird $f = O(g)$ statt $f \in O(g)$ geschrieben.
- $f = O(h) \wedge g = O(h) \rightarrow f + g = O(h)$
- $a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = O(n^k)$
- $f(n) = c \cdot g(n) \rightarrow f = O(g)$

O-Kalkül



- Einfache Regeln

$$f = O(f)$$

$$O(O(f)) = O(f)$$

$$kO(f) = O(f)$$

$$O(f + k) = O(f)$$

- Addition

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

- Multiplikation

$$O(f) \cdot O(g) = O(f \cdot g)$$

Zusammenfassung



- Werkzeuge zur Analyse der Effizienz von Algorithmen
 - Effizienz, Laufzeitkomplexität: Wachstum (Wachstumsgrad, Wachstumsrate) der Rechenzeit bei steigender Anzahl der Eingabe-Elemente
- Beschreibung der asymptotischen Effizienz durch Funktionenklassen
 - Konzentration auf das Wesentliche bei der Laufzeitbeschreibung
- Landau-Symbole
 - zur Beschreibung von asymptotisch scharfen bzw. nicht scharfen Schranken für die Beschreibung der Laufzeitkomplexität
- O
 - zur Beschreibung oberer asymptotischer Schranken von Laufzeiten

Nächstes Thema



- Analyse von Algorithmen
 - Abschätzung der Laufzeit von Algorithmen