2 Analyse von Algorithmen

Lernziele

- Wie zeigt man die Korrektheit eines Algorithmus?
- Wie kann man die Laufzeit abschätzen?
- Wie kann man den Speicherbedarf abschätzen?
- Asymptotische Laufzeit und Wachstum von Funktionen

Untersuchung am Beispiel eines Suchalgorithmus

Einführung: Das Sortierproblem

Das Abstrakte Sortierproblem wird beschrieben durch

- Eingabe: Eine Folge von n ganzen Zahlen $(a_1, a_2, \ldots a_n)$
- Ausgabe: Eine Permutation $(a'_1, a'_2, \ldots a'_n)$ mit $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Beispiel:

- Eingabe: (31, 41, 59, 26, 41,58)
- Ausgabe: (26, 31, 41, 41, 58, 59)

Einführung: Das Sortierproblem

Anwendungen

- Teilproblem vieler Anwendungen
- Viele Lösungen (Algorithmen) vorhanden
- Lösungen basieren z.T. auf unterschiedlichen Entwurfstechniken
- Problem ist gut verstanden, d.h. eine untere Schranke für die Laufzeit ist bekannt.
- Optimal lösbar, d.h. die besten Algorithmen haben eine Laufzeit in der Größenordnung der unteren Schranke

Einführung: Das Sortierproblem

Eine Lösung: Insertion-Sort

- Einfach zu verstehen
- Wenig effizient
- Dennoch gut bei kleinen oder vorsortierten Folgen
- Leicht zu implementieren
- Entspricht dem Vorgehen
  beim Einsortieren von Karten

Einführung: Das Sortierproblem

Achtung

- Wir benutzen hier Arrays
- Wir indizieren Arrays mit Indizes 1 … n
- In vielen Programmiersprachen werden Indizes 0 … n-1 verwendet

Einführung: Das Sortierproblem

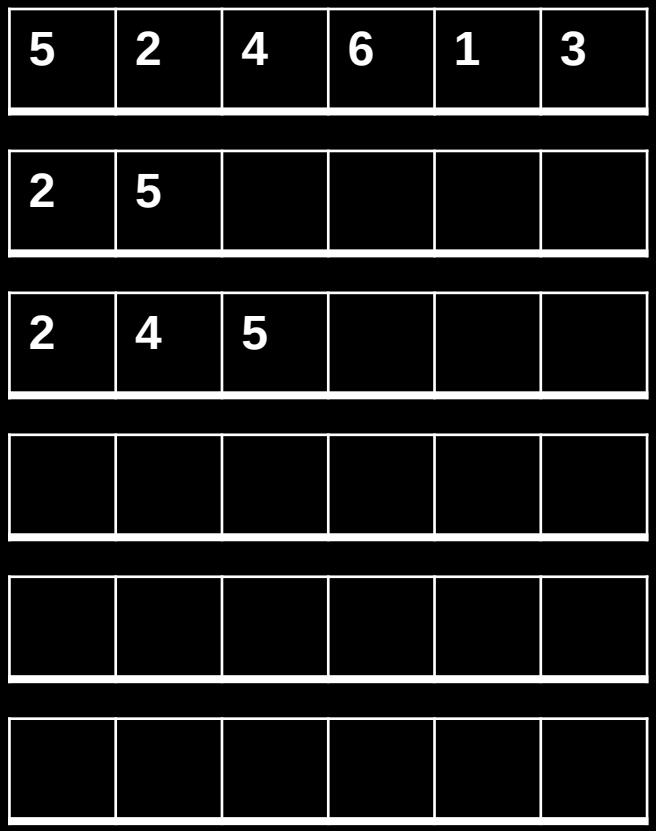
Beispiel Insertion-Sort

```
 1:  function INSERTION-SORT(A)
 2:      for j from 2 to length[A] do
 3:          key ← A[j]
 4:          i ← j – 1
 5:          while i > 0  ∧ A[i] > key do
 6:              A[i+1] ← A[i]
 7:              i ← i – 1;
 8:          end while
 9:          A[i+1] ← key
10:      end for
11: end function
```

Einführung: Das Sortierproblem

Beispiel Insertion-Sort

```
 1:  function INSERTION-SORT(A)
 2:      for j from 2 to length[A] do
 3:          key ← A[j]
 4:          i ← j – 1
 5:          while i > 0  ∧ A[i] > key do
 6:              A[i+1] ← A[i]
 7:              i ← i – 1;
 8:          end while
 9:          A[i+1] ← key
10:      end for
11: end function
```



Analyse

Wichtig und interessant sind:

- Korrektheit des Algorithmus
  Es ist zu zeigen, dass der Algorithmus für alle Instanzen des Problems eine korrekte Lösung berechnet
- Komplexität des Algorithmus
- Der Ressourcenbedarf des Algorithmus an Laufzeit und Speicher wird untersucht. In der Praxis relevant ist vor allem die Laufzeit.
- Wichtig ist ein aussagekräftiges Maß für diese Größen.

Analyse der Korrektheit

In Analogie zur Beweistechnik der vollständigen Induktion wird mit Invarianten (Schleifeninvarianten) gearbeitet

Man zeigt folgende Punkte:

- Initialisierung
  Die Invariante ist vor der ersten Iteration der Schleife wahr.
- Aufrechterhaltung
  Wenn die Invariante vor einer Iteration der Schleife erfüllt ist, dann ist sie auch vor Beginn der nächsten Iteration erfüllt.
- Terminierung
  Wenn die Schleife endet, dann ist die Invariante auch wahr. Sie liefert einen nützlichen Hinweis, um die Korrektheit des Algorithmus zu beweisen.

Analyse der Korrektheit: INSERTION-SORT

Schleifeninvariante

Die Teilfolge a_1 , … a_{j-1} ist sortiert.

Analyse der Korrektheit: INSERTION-SORT

Initialisierung

Vor Beginn des ersten Durchlaufs (Z. 2) wird j auf 2 gesetzt.

Die Teilfolge (a_1 , … a_{j-1}) ist dann (a_1)

Eine 1-elementige Folge ist trivialerweise sortiert

Analyse der Korrektheit: INSERTION-SORT

Beispiel Insertion-Sort

```
 1:  function INSERTION-SORT(A)
 2:      for j from 2 to length[A] do
 3:          key ← A[j]
 4:           i ← j – 1
 5:          while i > 0  ∧ A[i] > key do
 6:              A[i+1] ← A[i]
 7:               i ← i – 1;
 8:          end while
 9:          A[i+1] ← key
10:     end for
11: end function
```

Analyse der Korrektheit: INSERTION-SORT

Aufrechterhaltung

Funktion der Schleife

- Die for-Schleife bewegt die Elemente a_{j-1}, a_{j-2}, … jeweils eine Position nach rechts (Z. 5-8), bis der richtige Einfügeplatz für a_j gefunden ist.
- Dann wird a_j an diese Stelle geschrieben (Z. 9)

Analyse der Korrektheit: INSERTION-SORT

Aufrechterhaltung

Betrachtung der Schleife

- Der Schlüssel von a_j wird gemerkt.
- In der vorherigen Iteration wurde eine sortierte Teilfolge (a_1, … a_{j-1}) hergestellt.
- In der while-Schleife wird das erste Element a_i gesucht mit $a_i \leq a_j < a_{i+1}$
- a_j wird durch Verschieben der größeren Elemente zwischen a_i und a_{i+1} einsortiert.
- Danach sind die Elemente a_i, … a_j sortiert.
- Somit ist die Schleifeninvariante zum Start der nächsten Iteration wieder erfüllt.

Analyse der Korrektheit: INSERTION-SORT

Beendigung/Terminierung

Die Schleife endet mit $j = n + 1$.

Es wird mit diesem Wert von j kein Schleifendurchlauf mehr durchgeführt.

Da die Invariante für $(a_1, \ldots a_{j-1})$ gilt und da $j = n + 1$, gilt folglich $(a_1, \ldots a_n)$ ist sortiert.

Analyse der Komplexität

- Analyse erlaubt den „besten“ Algorithmus auszusuchen oder die ungünstigeren auszusortieren
- Analyse ist Vorhersage des Ressourcenverbrauchs eines Algorithmus
  - Rechenzeit
  - Speicher
  - Kommunikationsbandbreite
  - …
- Meist ist Rechenzeit wichtig

Analyse der Komplexität

Rechenzeit hängt ab von

- Probleminstanz, d.h. Eingabe
- Eingabelänge (problemabhängig)
  - Bei vielen Algorithmen Anzahl der Elemente der Eingabe (Sortieren, Fouriertransformation, …)
  - Bei manchen Algorithmen Anzahl der Bits der Eingabe (Binäre Addition, Multiplikation, …)

Analyse der Komplexität

Analyse benötigt

- Modell der Implementierung auf einem Rechner
- Modell der Ressourcen
- Kostenmodell für die Benutzung der Ressourcen

Oft benutzt

Random Access Machine (RAM), spezielle Art von Registermaschine

- Befehle werden sequentiell abgearbeitet
- Einfache, praxisgerechte Elementarbefehle und Datenstrukturen

Analyse des Ressourcenverbrauchs

Rechenzeit kann beschrieben werden

- Durch Anzahl der einfachen Operationen, durch einfaches Zählen von Schritten
- Durch Zuweisen von Rechenzeitbedarf für bestimmte Operationen

Analyse von INSERTION-SORT

1:	function INSERTION-SORT(A)	
2:	for j from 2 to length[A] do	n
3:	key ← A[j]	n-1
4:	i ← j – 1	n-1
5:	while i > 0 ∧ A[i] > key do	$\sum_{j=2}^{n} t_j$
6:	A[i+1] ← A[i]	$\sum_{j=2}^{n}(t_j - 1)$
7:	i ← i – 1;	$\sum_{j=2}^{n}(tj - 1)$
8:	end while	
9:	A[i+1] ← key	n-1
10:	end for	
11:	end function	

Problem ist die while-Schleife:

t_j steht für die Anzahl von Durchläufen der jeweiligen Zeile für ein bestimmtes j.

Analyse von INSERTION-SORT

1:	function INSERTION-SORT(A)	
2:	for j from 2 to length[A] do	n
3:	key ← A[j]	n-1
4:	i ← j – 1	n-1
5:	while i > 0 ∧ A[i] > key do	$\sum_{j=2}^{n} t_j$
6:	A[i+1] ← A[i]	$\sum_{j=2}^{n}(t_j - 1)$
7:	i ← i – 1;	$\sum_{j=2}^{n}(tj - 1)$
8:	end while	
9:	A[i+1] ← key	n-1
10:	end for	
11:	end function	

Gesamtlaufzeit: $T(n) = 4n - 3 + \sum_{j=2}^{n}(3t_j - 2)$

Analyse von INSERTION-SORT

Beobachtung

- Wir zählen einfach Operationen
- Laufzeit ist von Problemgröße n abhängig
- Bei gegebener Problemgröße ist die Laufzeit noch von der Art des Problems abhängig
- Laufzeit ist abhängig davon, wie oft die innere Schleife durchlaufen wird
- Dies ist vorab nicht ohne weiteres zu bestimmen

Analyse von Algorithmen

Arten der Analyse

Best case: Mindestlaufzeit bei günstiger Art des Problems.

Worst case: Maximallaufzeit bei ungünstiger Art des Problems.

Average case: Laufzeit im Mittel

Analyse von INSERTION-SORT

1:	function INSERTION-SORT(A)	
2:	for j from 2 to length[A] do	n
3:	key ← A[j]	n-1
4:	i ← j – 1	n-1
5:	while i > 0 ∧ A[i] > key do	$\sum_{j=2}^{n} t_j$
6:	A[i+1] ← A[i]	$\sum_{j=2}^{n}(t_j - 1)$
7:	i ← i – 1;	$\sum_{j=2}^{n}(tj - 1)$
8:	end while	
9:	A[i+1] ← key	n-1
10:	end for	
11:	end function	

Gesamtlaufzeit: $T(n) = 4n - 3 + \sum_{j=2}^{n}(3t_j - 2)$

Analyse von INSERTION-SORT

Best case

Günstig, falls in Zeile 5 die korrekte Einfügestelle sofort gefunden ist, d.h. $t_j = 1$ für alle j.

Dann werden Zeile 6 und 7 nicht ausgeführt

Dann: $T_{BC}(n) = 4n - 3 + (n - 1) = 5n - 4$

Gilt, falls die Folge bereits sortiert ist

Laufzeit ist lineare Funktion $T_{BC}(n) = a * n + b$

Analyse von INSERTION-SORT

1:	function INSERTION-SORT(A)	
2:	for j from 2 to length[A] do	n
3:	key ← A[j]	n-1
4:	i ← j – 1	n-1
5:	while i > 0 ∧ A[i] > key do	$\sum_{j=2}^{n} t_j$
6:	A[i+1] ← A[i]	$\sum_{j=2}^{n}(t_j - 1)$
7:	i ← i – 1;	$\sum_{j=2}^{n}(tj - 1)$
8:	end while	
9:	A[i+1] ← key	n-1
10:	end for	
11:	end function	

Gesamtlaufzeit: $T(n) = 4n - 3 + \sum_{j=2}^{n}(3t_j - 2)$

Analyse von INSERTION-SORT

Worst case

- Ungünstig, falls Zeilen 5-8 immer ganz durchlaufen werden bis Anfang der Folge.
- Es gilt:
- Dann: $T_{WC(}n) = 4n - 3 + \left(\frac{n(n-1)}{2} - 1\right) + n(n-1) = \frac{3}{2}n^2 + \frac{7}{2}n - 4$
- Laufzeit ist quadratische Funktion $T_{WC}(n) = an^2 + bn + c$

Analyse von INSERTION-SORT

Average case

- Worst case ist obere Schranke für die Laufzeit
- Schlechtester Fall kommt für manche Algorithmen häufig vor.
  Bsp. Suche in Datenbanken: Schlechtester Fall ist, falls gesuchtes Element nicht gefunden wird.
- Oft ist mittlerer Fall ähnlich schlecht wie schlechtester Fall. Bei Insertion Sort ergibt sich auch im mittleren Fall eine quadratische Laufzeit.

Wachstum von Funktionen

Motivation

- Man untersucht die Laufzeit von Algorithmen für große Eingabelängen
- Dann ist im Wesentlichen der Wachstumsgrad der Laufzeiten relevant
- Man nennt dies die Untersuchung der asymptotischen Effizienz von Algorithmen
- Problem: Die Aussage gilt dann nicht (exakt) für kleine Eingabelängen.
- Anders formuliert: Ein Algorithmus A mit besserer asymptotischer Effizienz ist u.U. für geringe Eingabegrößen die ungünstigere Wahl.

Wachstum von Funktionen

Aysmptotische Notation

Die O-Notation beschreibt obere Schranken für das Wachstum von Funktionen (Groß-O).

Die Ω-Notation beschreibt untere Schranken für das Wachstum von Funktionen (Groß- Ω bzw. Groß-Omega)

Die θ-Notation beschreibt exakte Schranken für das Wachstum von Funktionen (Groß- θ bzw. Groß-Theta)

Vernachlässigbare Anteile dürfen weggelassen werden:

Genauere Analyse ist oft schwierig oder gar unmöglich.

Konstante Summanden und Faktoren

Lineare Beschleunigung ist leicht möglich (schnellerer Rechner …)

Wachstum von Funktionen

Die O-Notation

Für eine gegebene Funktion g(n) ist O(g(n)) die folgende Menge von Funktionen

$$O(g(n)) = \{f(n) : \text{es ex. positive Konstanten } c_1, n_0 \text{ mit } 0 <= f(n) \leq c_1\, g(n) \text{ für alle } n \geq n_0\}$$

Man schreibt: $f \in O(g)$ oder auch $f = O(g)$.

g(n) ist eine asymptotische obere Schranke für f(n)

Wachstum von Funktionen

Beispiel zur O-Notation

Es soll gezeigt werden, dass $\frac{1}{2}n^2 + 3n \in O(n^2)$

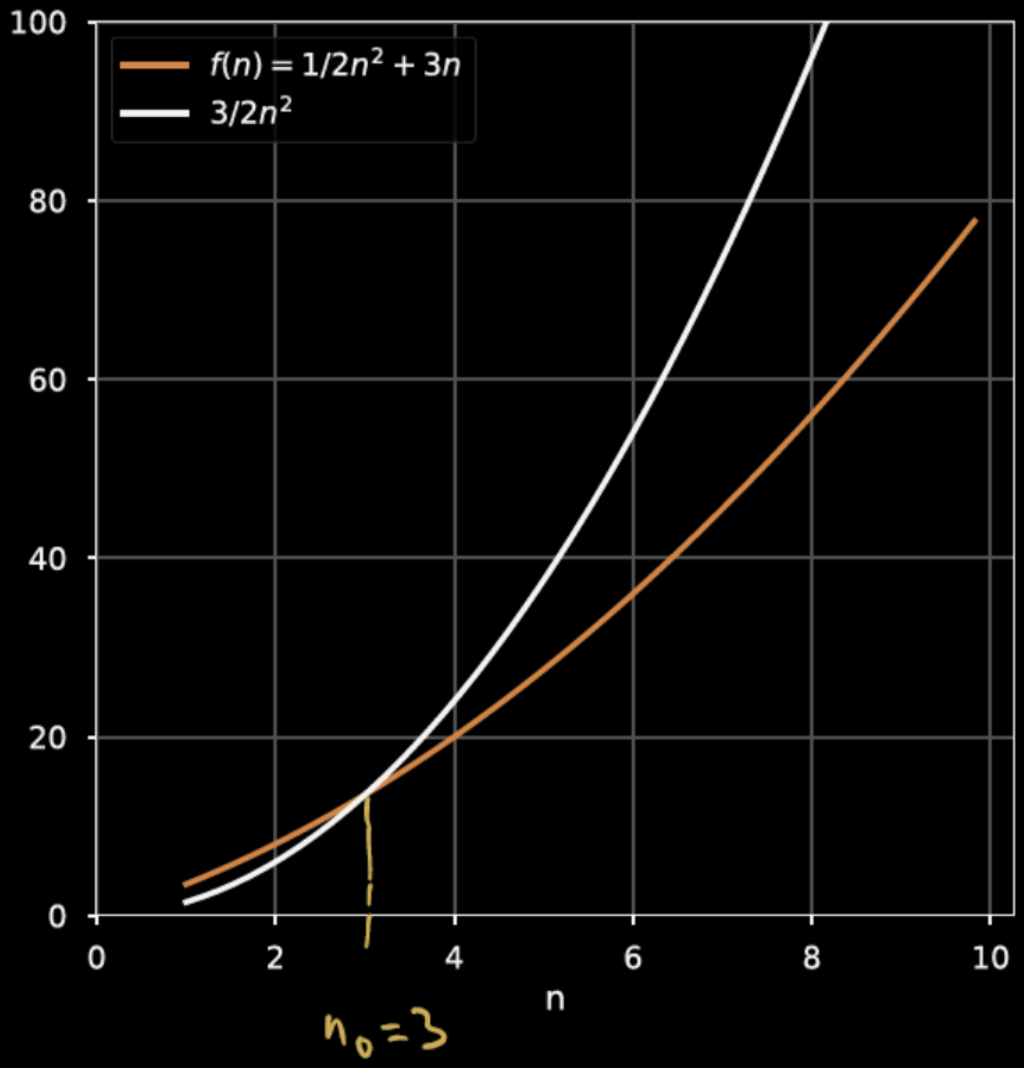
$$0 \leq \frac{1}{2}n^2 + 3n \leq c_1 n^2$$

$$0 \leq \frac{1}{2} + \frac{3}{n} \leq c_1$$

Wähle $c_1 = \frac{3}{2}$, ab $n_0 = 3$ erfüllt.

Wachstum von Funktionen

Beispiel



Wachstum von Funktionen

Beispiel zur O-Notation (2)

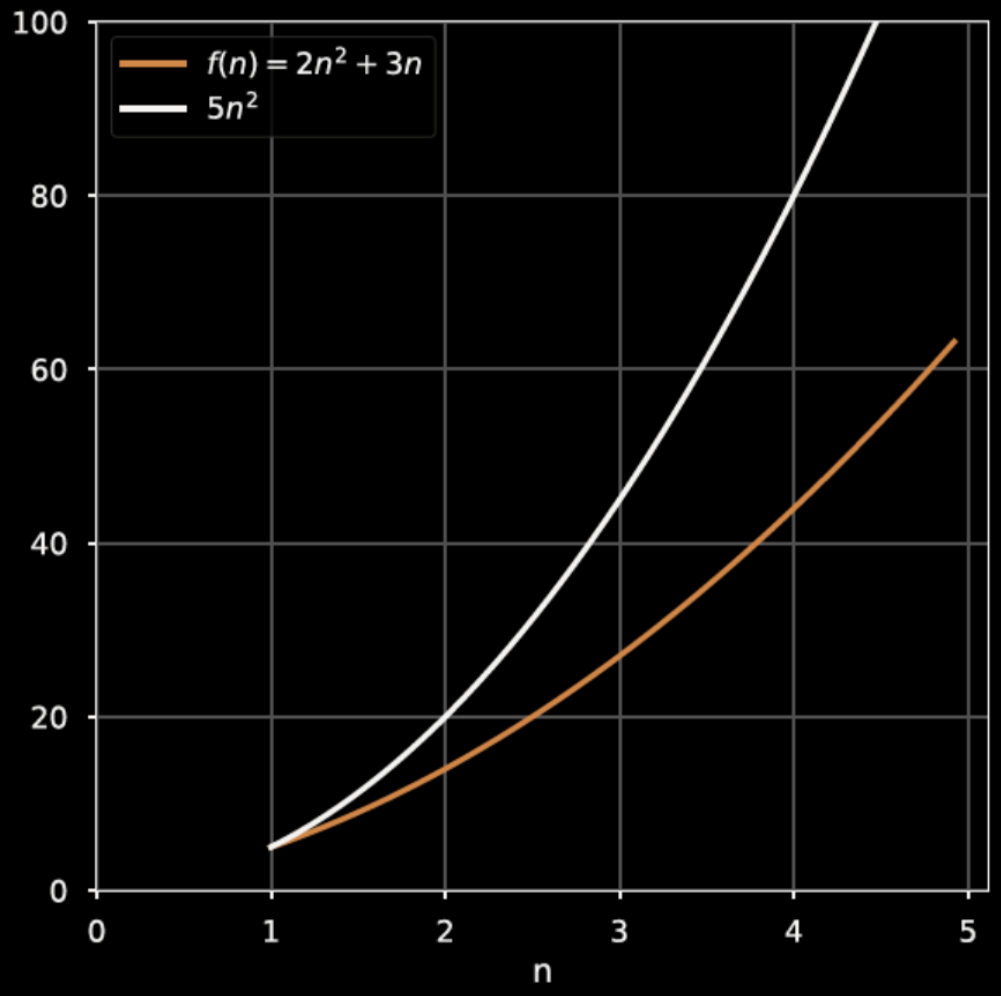
Es soll gezeigt werden, dass $2n^2 + 3n \in O(n^2)$

$$2n^2 + 3n \leq 2n^2 + 3n^2 = 5n^2 \leq c_1 n^2$$

Wähle: $c_1 = 5$, ab $n_0 = 1$ erfüllt.

Wachstum von Funktionen

Beispiel (2)



Wachstum von Funktionen

Die Ω–Notation

Für eine gegebene Funktion g(n) ist

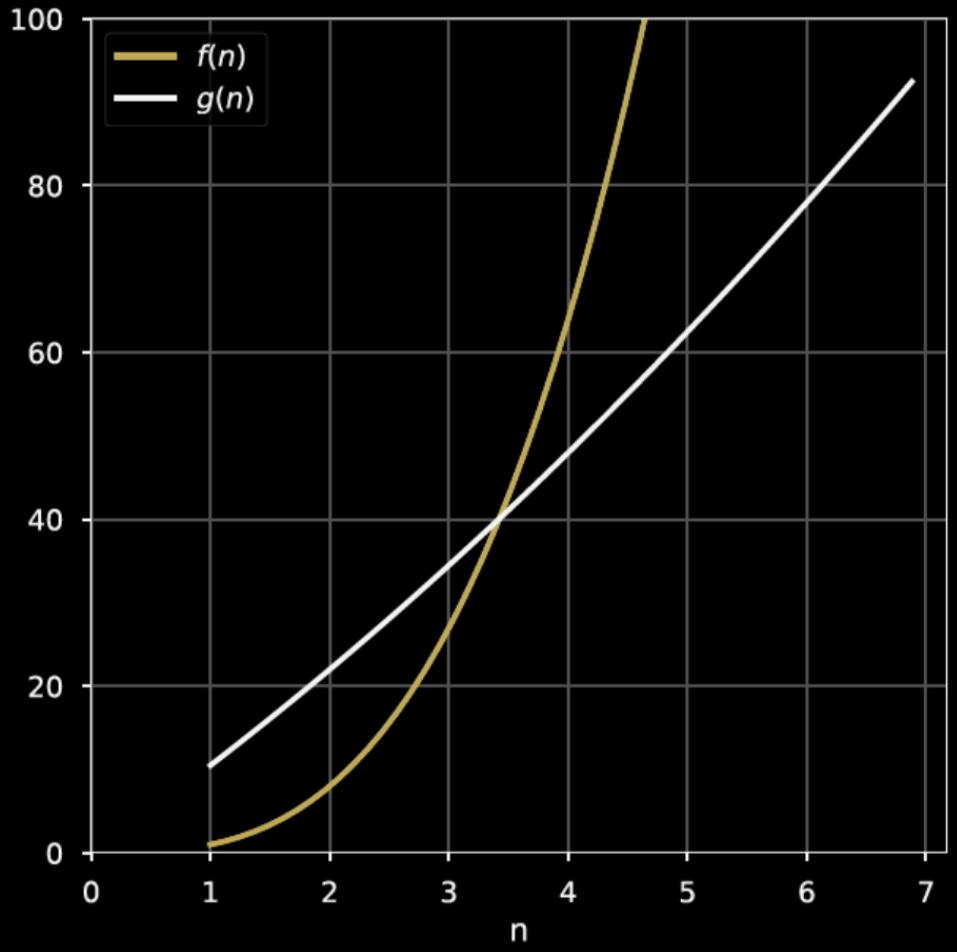
$$\Omega(g(n)) = \{f(n) : \text{ es ex. positive Konstanten } c_1, n_0 \text{ mit } 0 \leq c_1\, g(n) <= f(n) \text{ für alle } n \geq n_0\}$$

Man schreibt: $f \in \Omega(g)$ oder auch $f = \Omega(g)$.

g(n) ist eine asymptotische untere Schranke für f(n)

Wachstum von Funktionen

Veranschaulichung der Ω–Notation



Wachstum von Funktionen

Beispiel zur Ω–Notation

Es soll gezeigt werden, dass $\frac{1}{2}n^2 + 3n \in \Omega(n^2)$

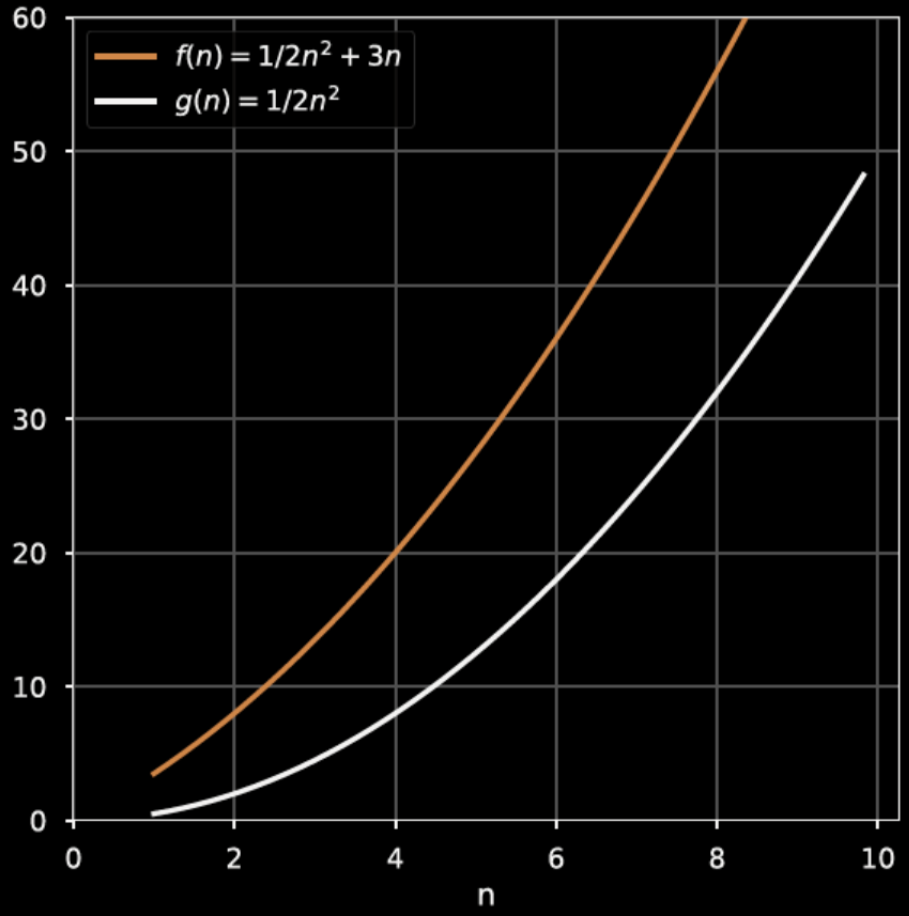
$$0 \leq c_1 n^2 \leq \frac{1}{2}n^2 + 3n$$

$$0 \leq c_1 \leq \frac{1}{2} + \frac{3}{n}$$

Wähle $c_1 = \frac{1}{2}$, ab $n_0 = 1$ erfüllt.

Wachstum von Funktionen

Beispiel zur Ω-Notation



Wachstum von Funktionen

Die θ-Notation

Für eine gegebene Funktion g(n) ist

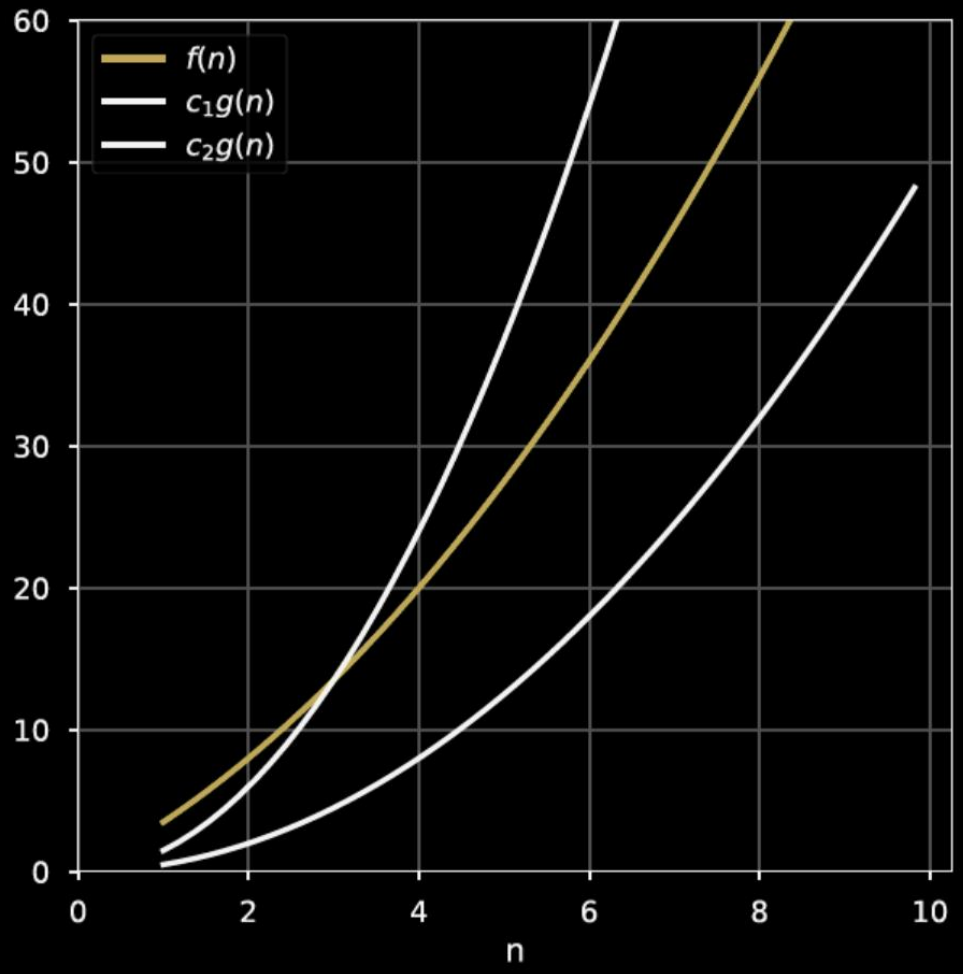
$$\theta(g(n)) = \{f(n) : \text{ es ex. positive Konstanten } c_1, c_2, n_0 \text{ mit } 0 \leq c_1\, g(n) <= f(n) \leq c_2\, g(n) \text{ für alle } n \geq n_0\}$$

Man schreibt: $f \in \theta(g)$ oder auch $f = \theta(g)$.

g(n) ist eine asymptotische exakte Schranke für f(n)

Wachstum von Funktionen

Veranschaulichung der θ-Notation



Wachstum von Funktionen

Beispiel zur θ-Notation

Es soll gezeigt werden, dass $\frac{1}{2}n^2 + 3n \in \theta(n^2)$

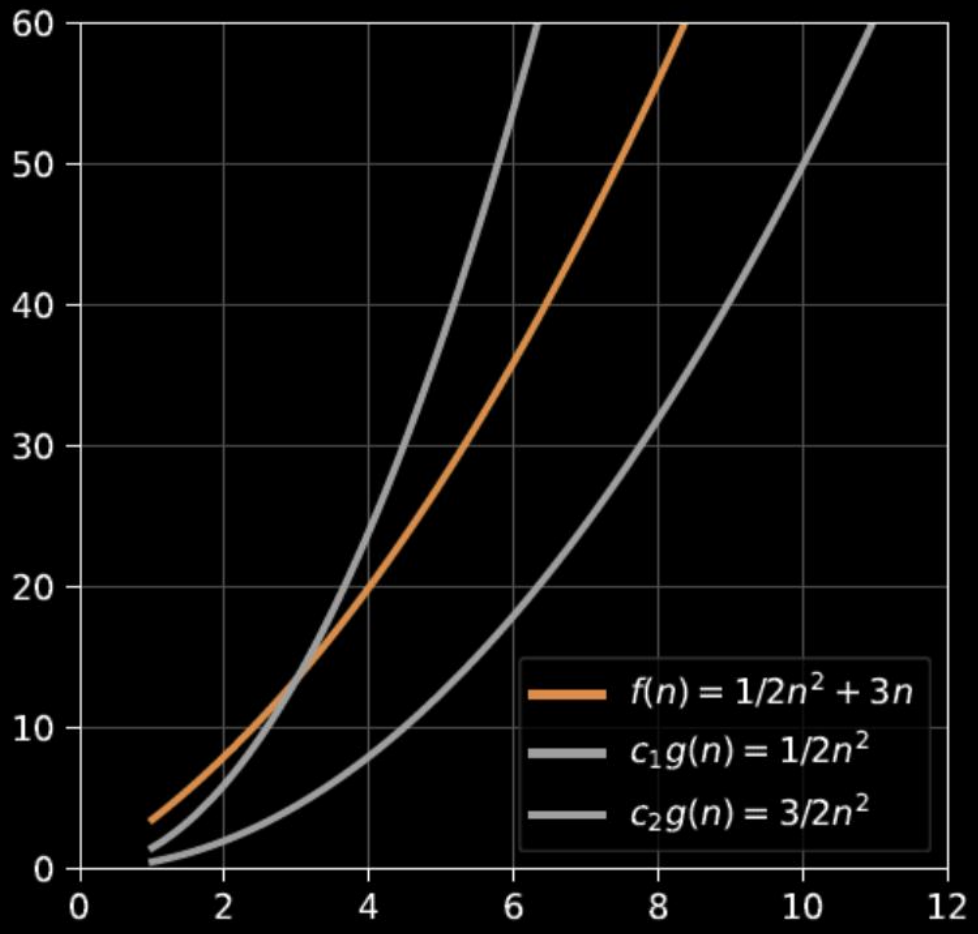
$$0 \leq c_1 n^2 \leq \frac{1}{2}n^2 + 3n \leq c_2 n^2$$

$$0 \leq c_1 \leq \frac{1}{2} + \frac{3}{n} \leq c_2$$

Wähle $c_1 = \frac{1}{2}$, $c_2 = \frac{3}{2}$, ab $n_0 = 3$ erfüllt.

Wachstum von Funktionen

Beispiel zur θ-Notation



Wachstum von Funktionen

Symbol	$\exists\, c_1, c_2 > 0\ \exists\, n_0 > 0 : \forall\, n \geq n_0$	Analogie
$f \in O(g)$	$0 \leq f(n) \leq c_1 \cdot g(n)$	$a \leq b$
$f \in \Omega(g)$	$0 \leq c_1 \cdot g(n) \leq f(n)$	$a \geq b$
$f \in \Theta(g)$	$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$a = b$

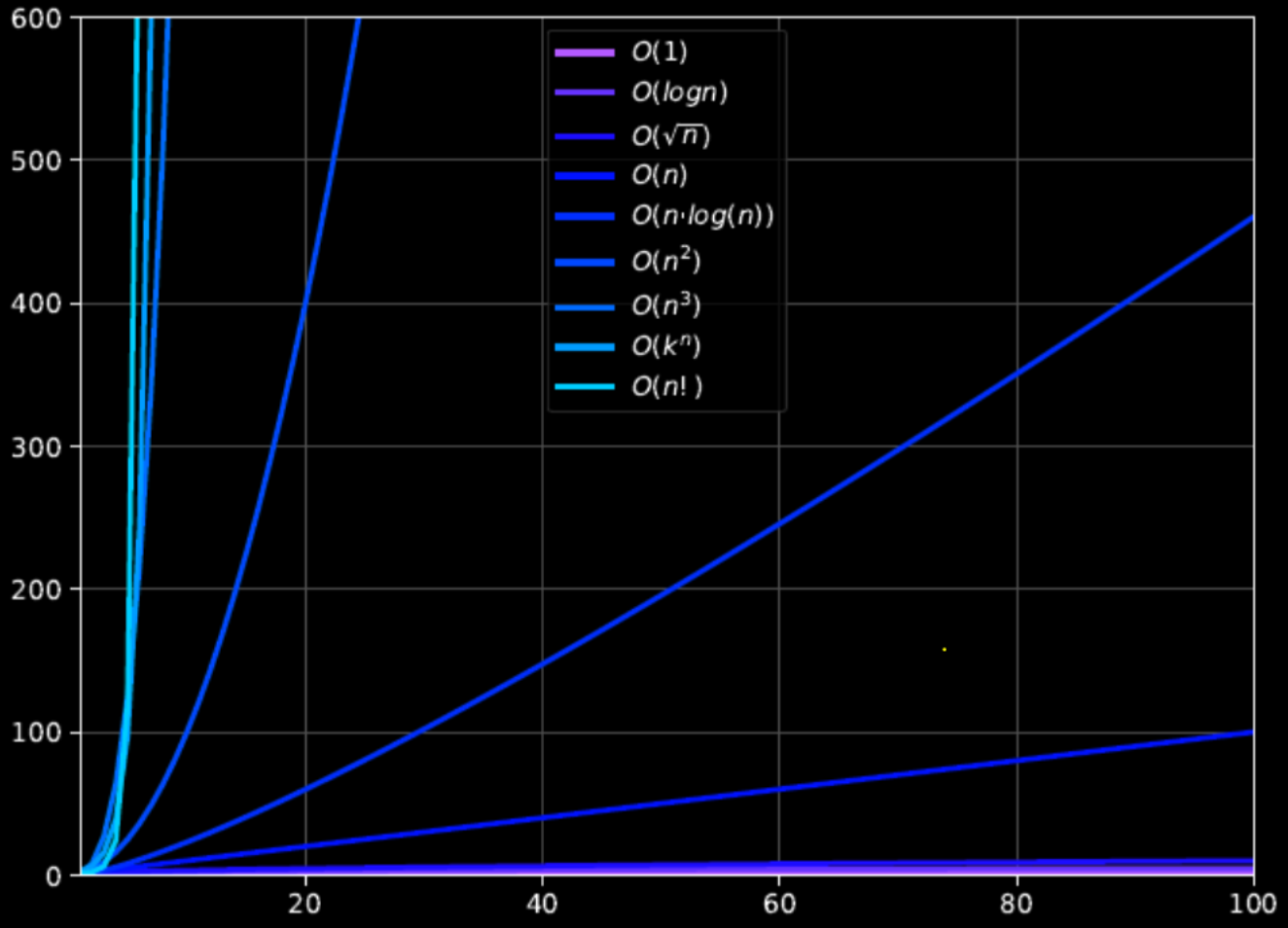
Wachstum von Funktionen

Hierarchie von Größenordnungen

Notation	Name	Beispiel
$O(1)$	konstant	Ist eine Binärzahl gerade?
$O(\log n)$	logarithmisch	Binäre Suche im sortierten Array
$O(\sqrt{n})$	Wurzelfunktion	Divisionen beim naiven Primzahltest
$O(n)$	linear	Lineare Suche im unsortierten Array
$O(n \log n)$	loglinear, super-linear	Gute vergleichsbasierte Sortieralgorithmen
$O(n^2)$	quadratisch	Einfache Sortieralgorithmen
$O(n^3)$	kubisch	
$O(n^k)$	polynomiell	Viele „einfache“ Algorithmen
$O(k^n)$	exponentiell	Erfüllbarkeitsproblem der Aussagenlogik (SAT) mit erschöpfender Suche
$O(n!)$	faktoriell	Traveling Salesman mit erschöpfender Suche

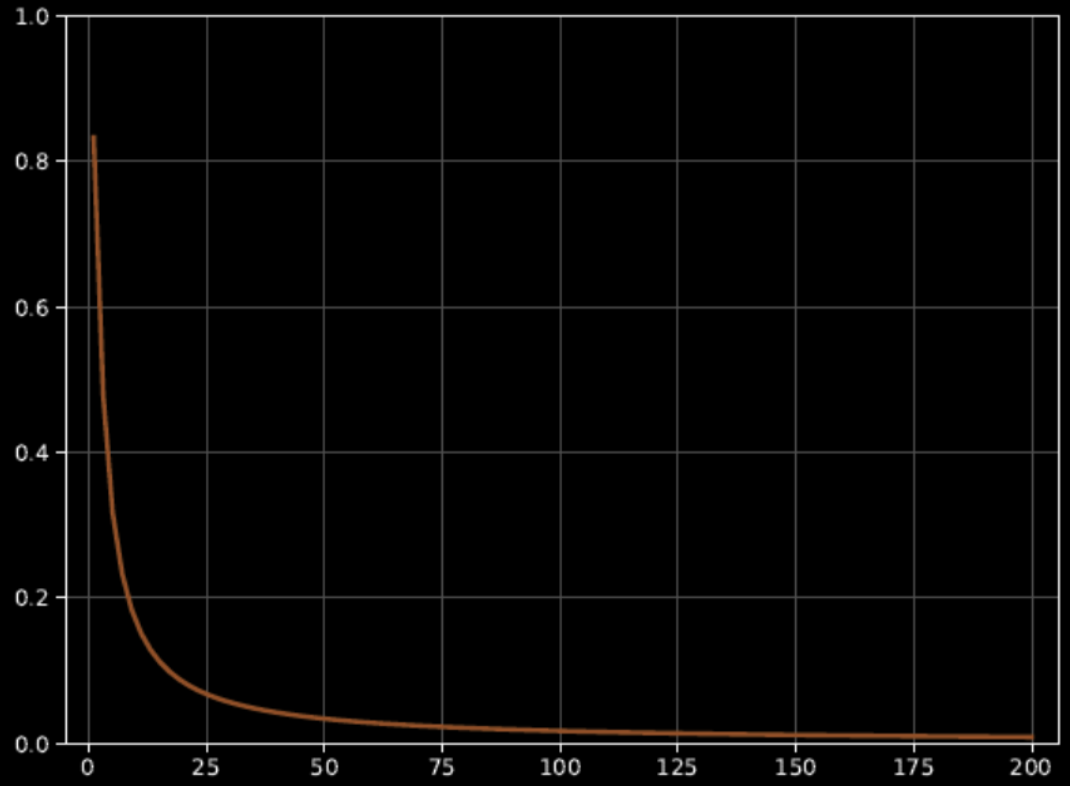
Wachstum von Funktionen

Hierarchie von Größenordnungen



Wachstum von Funktionen

Anteil von Termen niedriger Ordnung



Beispiel: Anteil von „5n + 10“ an „$3n^2$ + 5n + 10“

Wachstum von Funktionen

Skalierbarkeit

Annahme: 1 Rechenschritt benötigt 1ms

Maximale Problemgröße bei gegebener Rechenzeit:

T(n)	1 s	1 min	1 h
$O(n)$	1.000	60.000	3.600.000
$O(n \log n)$	140	4.895	204.094
$O(n^2)$	31	244	1.897
$O(n^3)$	10	39	153
$O(2^n)$	9	15	21

Wachstum von Funktionen

Skalierbarkeit

Annahme: Es wird auf einen 1000-mal so schnellen Rechner gewechselt.

Welche Problemgröße kann jetzt in gleicher Zeit berechnet werden?

Alte Problemgröße sei p.

T(n)	Neue Problemgröße
$O(n)$	$1000p$
$O(n \log n)$	fast $1000p$
$O(n^2)$	$\sqrt{1000p} = 31.6p$
$O(n^3)$	$\sqrt[3]{1000p} = 10p$
$O(2^n)$	$\log_2 1000 + p = 10 + p$

Wichtige Eigenschaften und Rechenregeln

Rechenregeln

$f(n) \in O(g(n))$ und c konstant $\quad c \cdot f(n) \in O(g(n))$

$f(n) \in O(g(n))$ und c konstant $\quad f(n) + c \in O(g(n))$

$O(f(n) + g(n)) \in O(\max(f(n),g(n))$

$O(f(n)) \cdot O(g(n)) \in O(f(n) \cdot g(n))$

Anwendung

O(f+g): Hintereinanderausführung von Programmteilen (Sequenzen)

$O(f) \cdot O(g)$ Schachtelung von Programmteilen