Einleitung

Andreas Hohenauer

Algorithmen und Datenstrukturen

14. September 2025

Ablauf

- **Samstag:**
  - Einleitung: Laufzeitanalyse, einfaches Sortieren
  - Grundlagen Graphentheorie
  - Rekursion
  - Sortieralgorithmen (Mergesort, Quicksort, Heapsort)
  - Mittagspause ca. 12:30
- **Sonntag:**
  - Datenstrukturen (Hashing, Dictionaries, ...)
  - Vertiefende Graphentheorie
  - Algorithmen auf Graphen (DFS/BFS, Dijkstra, Spannbäume,...)
  - Optimierungsalgorithmen (Überblick)
- **Konsultationsstunde:** Terminvorschlag Freitag, 17. Oktober 2025, 16:30 bis 18:30
- **Prüfung:** Freitag, 14. November 2025

Unterlagen

- Skriptum
  - Anhang: Lösung zu testrelevanten Beispielen
- Folien
- Mustertest
- Python-Codes
- Verfügbar
  - OPAL Skriptenportal
  - `github.com/hoa-spg/ibs/`

Warum Algorithmen und Datenstrukturen?

Algorithmen und Datenstrukturen sind das unsichtbare „Betriebssystem der Effizienz“

- Sie entscheiden, ob Programme in Sekunden oder in Stunden laufen.
- Bessere Hardware bringt höchstens lineare Beschleunigung – ein besserer Algorithmus kann den Aufwand von Jahren auf Sekunden reduzieren.
- Algorithmische Effizienz reduziert Hardware-, Energie- und Klimatisierungskosten – ein Gewinn für den Geldbeutel und die Umwelt.
- Erst durch clevere Algorithmen werden viele Ideen überhaupt praktikabel – sie machen den Unterschied zwischen „theoretisch möglich“ und „im Alltag nutzbar“.

Grundbegriffe

Definition (Problemstellung)

Die **Problemstellung** ist die formale Spezifikation, welche Beziehung zwischen Eingaben und gültigen Ausgaben gefordert ist (z. B. „sortiere die Elemente nichtabsteigend“ oder „finde einen kürzesten Pfad“).

Definition (Algorithmus)

Ein **Algorithmus** ist eine endliche, eindeutig beschriebene Folge von Anweisungen, die für jede zulässige Eingabe nach endlich vielen Schritten eine Ausgabe liefert (Terminierung) und dabei die geforderte Aufgabe korrekt löst (Korrektheit).

Grundbegriffe

Definition (Instanz)

Eine **Instanz** eines Problems ist eine konkrete Eingabe, auf die das Problem angewandt werden soll (z. B. eine Liste konkreter Zahlen für das Sortierproblem oder ein bestimmter Graph für ein Pfadproblem).

Definition (Ausgabe/Ergebnis)

Die **Ausgabe** (das **Ergebnis**) eines Algorithmus ist das von ihm für eine gegebene Instanz produzierte Objekt (Wert, Struktur), das die Spezifikation der Problemstellung erfüllt. Korrektheit bedeutet, dass jede erzeugte Ausgabe eine *gültige Lösung* ist, und Vollständigkeit (wo gefordert), dass eine Lösung gefunden wird, sofern eine existiert.

Grundbegriffe

- Arten von Algorithmen: Greedy, Divide-and-Conquer, Dynamische Programmierung, Backtracking, Branch-and-Bound, Graphalgorithmen, Numerische Verfahren, Parallele und verteilte Algorithmen, Online-Algorithmen, Approximationsalgorithmen
- Anwendungsgebiete: Sortieren, Suchen, Graphen, Optimierung, Kryptographie, Geometrie, Numerik, Datenbanken, Maschinelles Lernen, ...

Mengen

Definition

Menge nach Cantor „Zusammenfassung bestimmter, wohlunterschiedener Objekte unserer Anschauung oder unseres Denkens zu einem Ganzen“

- Konkrete Elemente können in Mengen nur einmal enthalten sein.
- Es gibt keine Reihenfolge der Elemente in der Menge. Es wird lediglich festgehalten ob ein Element zugehörig zu einer Menge ist, oder nicht.

Eine Menge ohne Elemente nennt man *leere Menge*. Sie wird entwedet mit $\emptyset$ oder $\{\}$ angeschrieben.

Beispiel

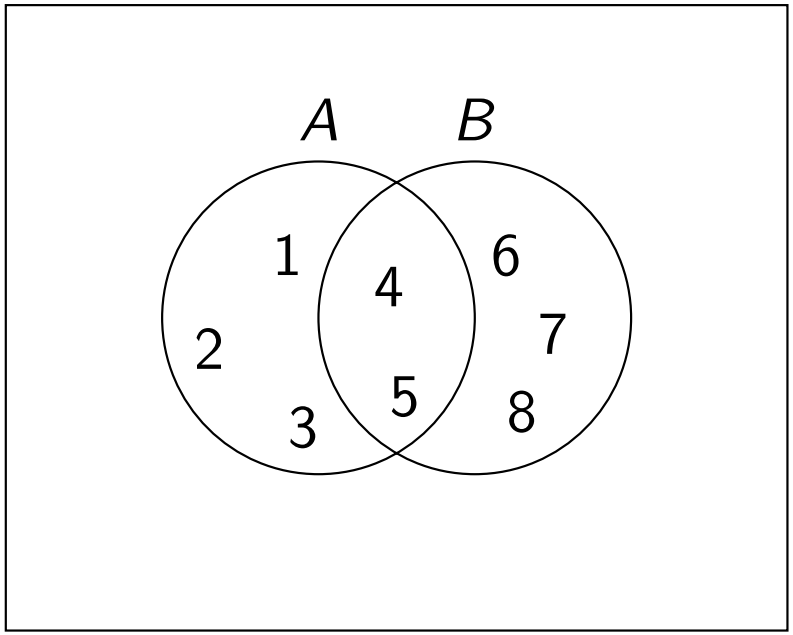
Wir betrachten die folgenden Mengen:

$$A = \{1, 2, 3, 4, 5\}$$

$$B = \{4, 5, 6, 7, 8\}$$

$$D = \{6, 7, 8\}$$

$$E = \{7, 8\}$$



Im (Euler-)Venn-Diagramm werden Mengen grafisch dargestellt. Im konkreten Beispiel sind D und E nicht dargestellt, jedoch die Mengen A und B wie links definiert.

Zugehörigkeit zu einer Menge

Um die Zugehörigkeit zu einer Menge zu notieren, wird das Symbol "$\in$" verwendet. So ist beispielsweise $3 \in A$, oder $7 \in E$.

Oftmals werden in diesem Zusammenhang Variablen verwendet, um die Elemente einer Menge anzusprechen. Beispiel: die Summe aller Elemente der Menge A, also aller $x \in A$ kann angeschrieben werden als

$$\sum_{x \in A} x = 1 + 2 + 3 + 4 + 5 = 15.$$

Mengentheoretische Begriffe und Notation anhand von Beispielen

- **Schnittmenge:** enthält alle Elemente die in beiden Mengen enthalten sind.

$$A \cap B = \{4, 5\}$$

- **Vereinigungsmenge:** enthält alle Elemente die in einer der beiden Mengen (oder beiden) enthalten sind.

$$A \cup B = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

- **Differenzmenge:** enthält alle Elemente der ersten Menge die jedoch nicht in der zweiten Menge enthalten sind.

$$A \backslash B = \{1, 2, 3\}$$

$$B \backslash A = \{6, 7, 8\}$$

Mengentheoretische Begriffe und Notation anhand von Beispielen

- **(Echte) Teilmenge:** (alle Elemente einer Menge sind auch in einer anderen Menge enthalten, die Mengen sind *nicht* gleich)

$$E \subset (B \backslash A)$$

- **(Unechte) Teilmenge:** entweder (echte) Teilmenge, oder die Mengen sind gleich!

$$D \subseteq (B \backslash A)$$

Im konkreten Fall gilt nicht $D \subset (A \backslash B)$, da D keine echte Teilmenge von $(B \backslash A)$ ist.

Kardinalität

Unter *Kardinalität* versteht man die *Größe* oder *Mächtigkeit* einer Menge. Bei endlichen Mengen entspricht dies der Anzahl der darin enthaltenen Elemente. Die Schreibweise sind die vertikalen Striche.

Bemerkung: dies hat nichts mit dem Betrag (Zahlen) oder der Längen von Vektoren zu tun. Diese Begriffe sind für Mengen gar nicht sinnvoll anwendbar.

Beispiel:

$$F = \{2, 4, 6, 8, 11\}$$

$$|F| = 5$$

Beispiel:

$$|\{\}| = 0$$

Potenzmenge

Die Potenzmenge einer Menge enthält alle ihre Teilmengen als Elemente.

$$\mathcal{P}(X) = \{U \mid U \subseteq X\}$$

Die Potenzmenge einer Menge X wird entweder durch $\mathcal{P}(X)$ oder oft auch als 2^X angeschrieben.

Beispiel:

$$M = \{1, 2, 3\}$$

$$\mathcal{P}(M) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

Kartesisches Produkt

Das kartesische Produkt ist die Menge aller geordneten Paare der Elemente zweier Mengen.

Motivation

- Bestimmte Programmteile verursachen je nach Datenmenge unterschiedliche *Laufzeiten* bei der Ausführung.
- Bei geringen Datenmengen bleibt dies oft unbemerkt und spielt keine Rolle.
- Reale Programme arbeiten jedoch mit großen Datenmengen, umfangreichen Datenbanken, führen komplexe Berechnungen durch, etc.
- Durch die theoretische Laufzeitanalyse kann man etwaige Flaschenhälse und Performanceprobleme erkennen.
- Relevant ist hierbei weniger die konkrete Ausführungsgeschwindigkeit, sondern das Wachstum der Laufzeit in Abhängigkeit von der Anzahl der Eingabedaten.
- In der praktischen Arbeit werden *Profiler* verwendet, um zu analysieren wieviel Zeit in welchen Programmteilen verbracht wird.
- Für theoretische Analysen wird die O-Notation verwendet.

Komplexität von Algorithmen

Bei Algorithmen betrachten wir typischerweise:

- **Zeitkomplexität** (Anzahl elementarer Schritte / Vergleiche / Operationen)
- **Speicherplatzkomplexität** (zusätzlicher Speicherbedarf, z. B. Hilfsarrays, Rekursionsstack)
- Weitere zählbare Parameter (z. B. Anzahl Vergleiche, Anzahl Verschiebungen beim Sortieren)

Unterscheidung von Fällen:

- **Worst-Case**: ungünstigste Eingabe
- **Average-Case**: durchschnittliche (erwartete) Eingabe
- **Best-Case**: günstigste Eingabe

Komplexität von Algorithmen

- Untersucht werden die Eigenschaften von Algorithmen in Abhängigkeit von den Eingabedaten.
- Wesentliche Kenngröße ist die Anzahl der Daten, oft mit n oder m bezeichnet.
  - *Beispiel:* Sortieren von n Zahlen
  - *Beispiel:* Tiefensuche in Graphen mit n Knoten und m Kanten
  - *Beispiel:* Suchen eines Elementes in n vorhandenen Einträgen
- Zentrale Frage: wie hängt die Laufzeit von n (oder m) ab?
- Von Interesse ist lediglich die **Größenordnung**!
- Diese Größenordnung wird mit der Bachmann-Landau-Notation, oder kurz O-**Notation** dargestellt.

Komplexität von Algorithmen

- Ein sehr günstiges Laufzeitverhalten ist, wenn bei n Datensätzen **ca.** n Schritte ausgeführt werden.
  - Man nennt dies: **lineares** Laufzeitverhalten
  - Der Algorithmus durchläuft $O(n)$ Schritte
- Werden ungefähr n^2 viele Schritte ausgeführt:
  - **Quadratisches** Laufzeitverhalten
  - Der Algorithmus durchläuft $O(n^2)$ Schritte
- Höhere Laufzeiten sind oftmals in der Praxis problematisch, selbst bei quadratischer Laufzeit ist oft schon Vorsicht angebracht

Definition ((informell) Laufzeit eines Algorithmus anhand der O-Notation)

Ein Algorithmus A hat bei n Eingabedaten eine Laufzeit von $O(f(n))$, wenn bei seiner Ausführung *in etwa* ("in der Größenordnung von") $f(n)$ Schritte ausgeführt werden.

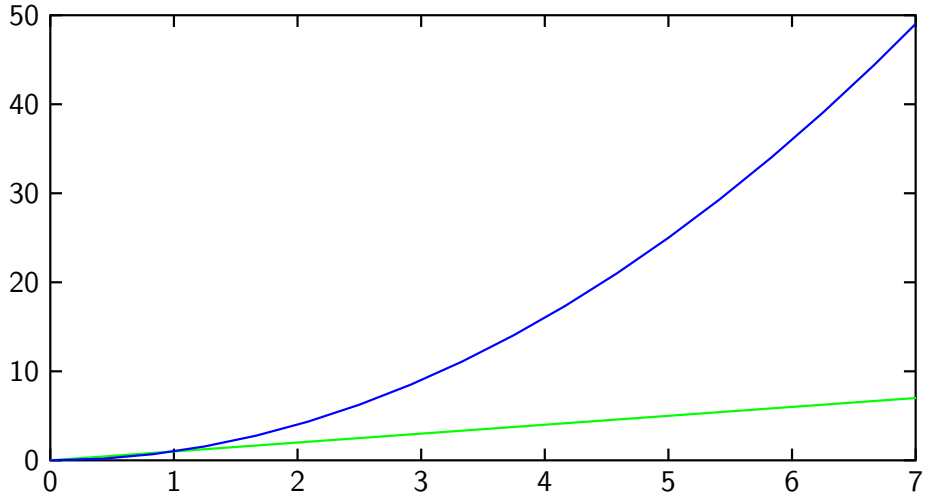
Häufige Komplexitätsklassen

Notation	**Name**	**Beispiel**
$O(1)$	konstant	Array-Zugriff
$O(\log n)$	logarithmisch	Binäre Suche
$O(n)$	linear	Lineare Suche
$O(n \log n)$	linearithmisch	Merge Sort
$O(n^2)$	quadratisch	Bubble / Insertion / Selection Sort
$O(n^3)$	kubisch	Naive Matrixmultiplikation
$O(2^n)$	exponentiell	Teilmengen aufzählen
$O(n!)$	faktoriell	Alle Permutationen

Für große n:

$$O(1) \ll O(\log n) \ll O(n) \ll O(n \log n) \ll O(n^2) \ll O(2^n) \ll O(n!).$$

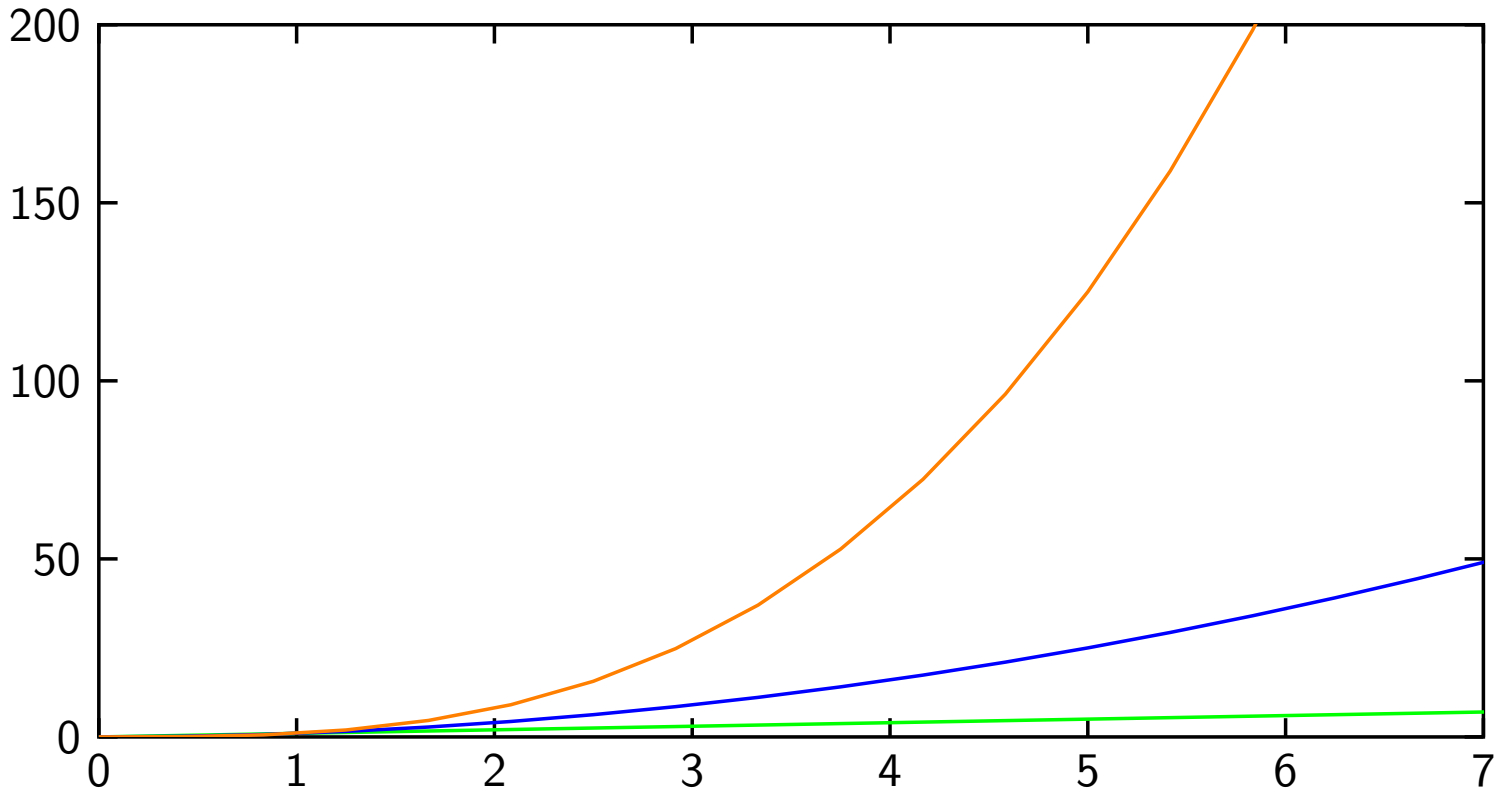
Wachstum von $f(n)$



$f(n) = n$, $f(n) = n^2$

Vergleich von linearem zu quadratischem Laufzeitverhalten

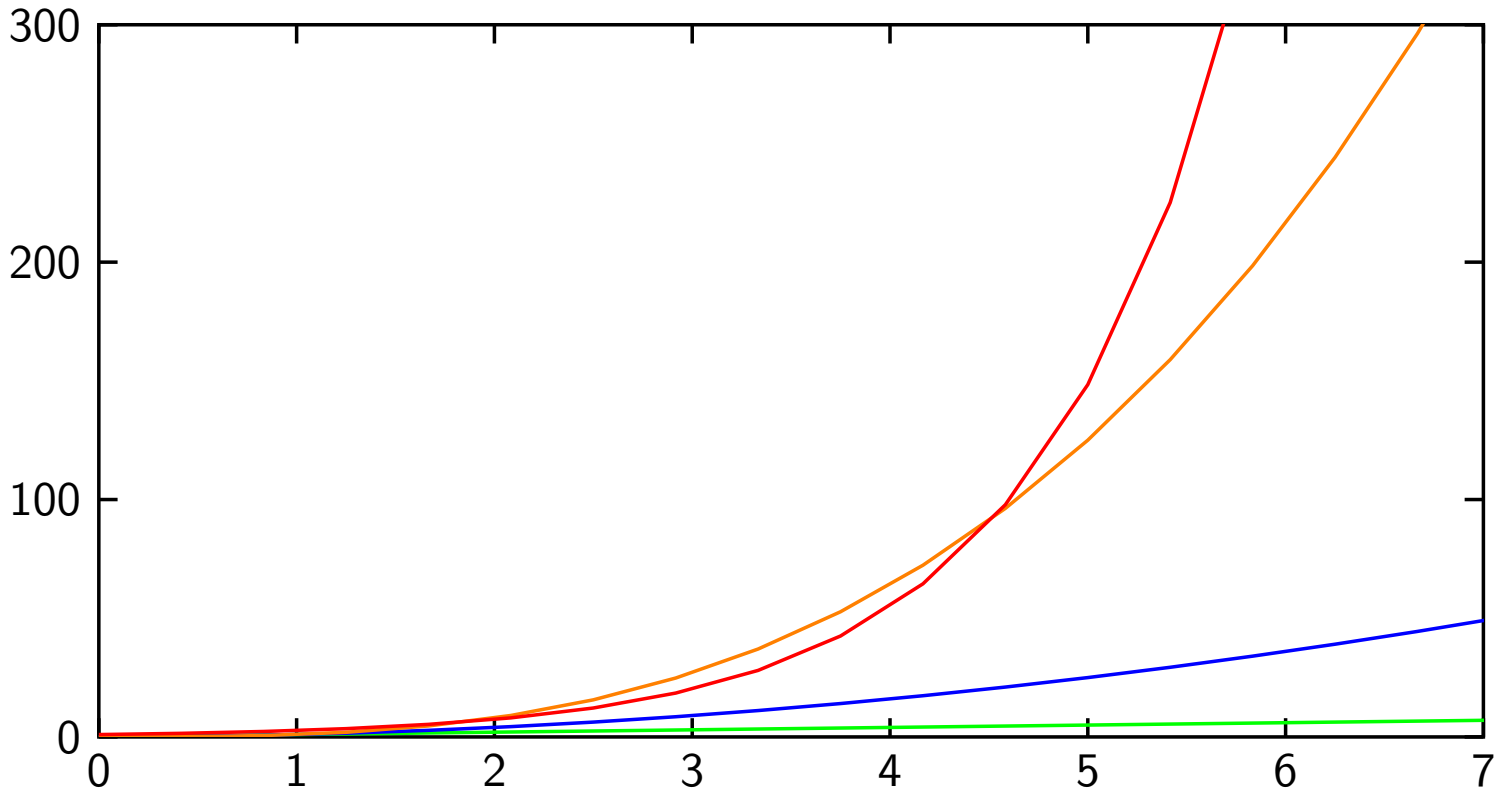
Wachstum von $f(n)$



$f(n) = n$, $f(n) = n^2$, $f(n) = n^3$

Vergleich von n, n^2 und n^3

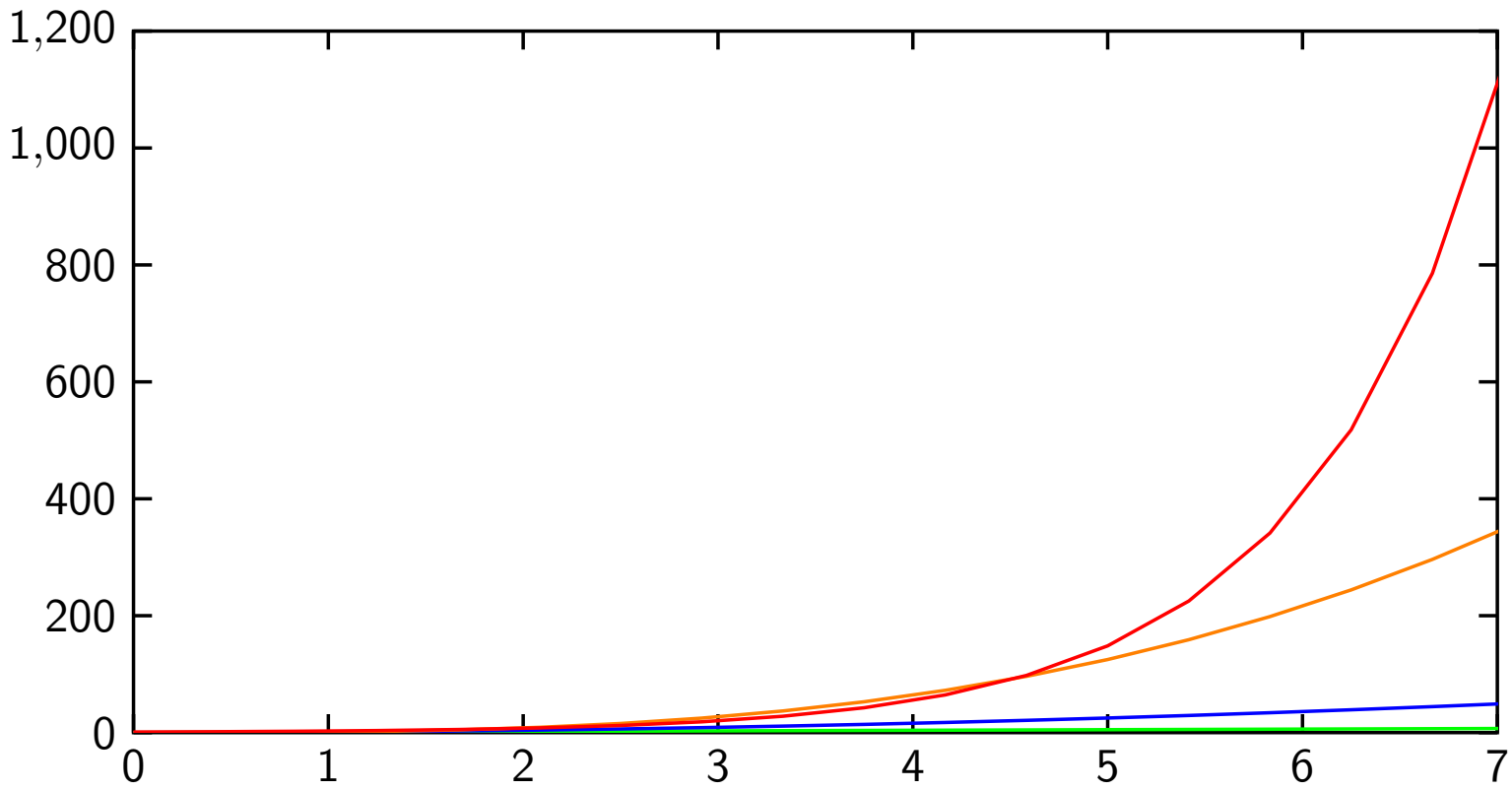
Wachstum von $f(n)$



$f(n) = n$, $f(n) = n^2$, $f(n) = n^3$, $f(n) = e^n$

Zusätzliche exponentielle Kurve, zunächst mit $y_{max} = 300$

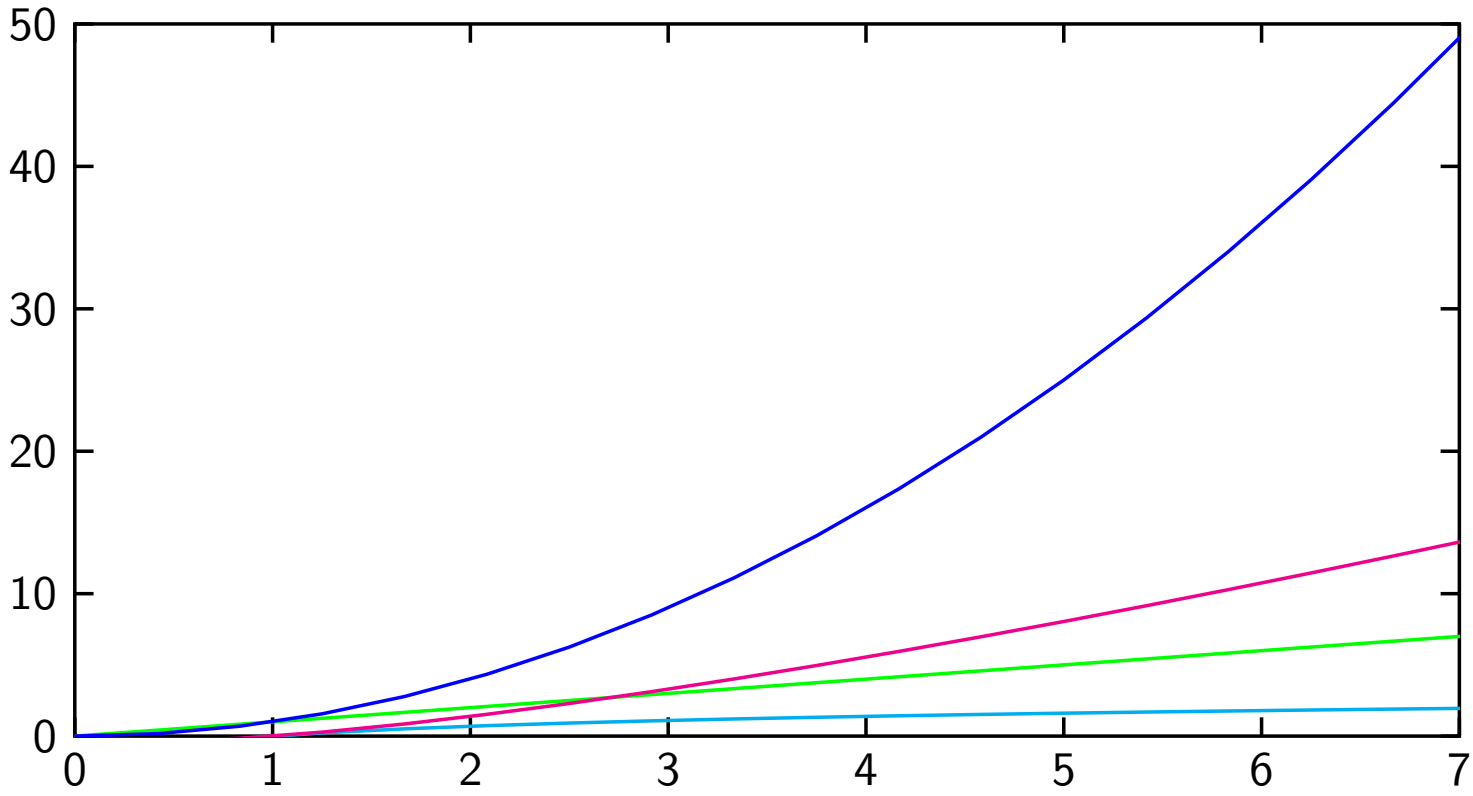
Wachstum von $f(n)$



$f(n) = n$, $f(n) = n^2$, $f(n) = n^3$, $f(n) = e^n$

Zusätzliche exponentielle Kurve, jetzt mit $y_{max} = 1200$

Wachstum von $f(n)$



$f(n) = n$, $f(n) = n^2$, $f(n) = \ln n$, $f(n) = n \ln n$

Weiteres Beispiel: die Logarithmus-Funktion. Diese strebt zwar gegen Unendlich, aber extrem langsam! Wichtige Konsequenz: $n \ln n$ verhält sich "fast" wie n

Bachmann-Landau Notation

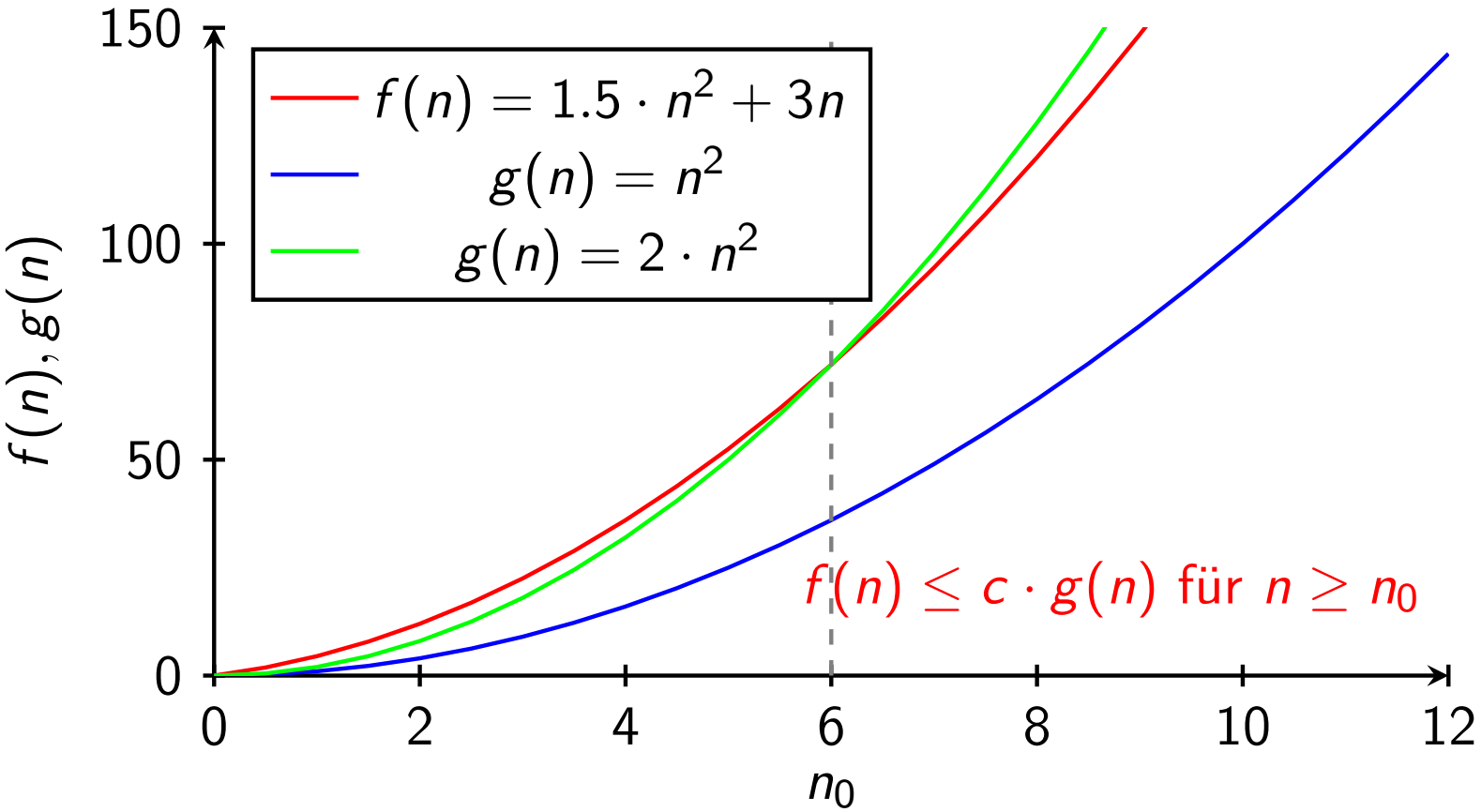
Eine Funktion $f(n)$ liegt in $O(g(n))$ wenn ab einem Wert n_0 und einer Konstante c die Funktionswerte von $f(n)$ kleiner sind als jene von $c \cdot g(n)$. Man schreibt hierfür auch $f(n) \in O(g(n))$.

Definition (Bachmann-Landau O-Notation)

$$O(g(n)) = \{f(n) \mid (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq f(n) \leq cg(n)\}$$

Somit ist $c \cdot g(n)$ ab n_0 eine **obere Schranke** für $f(n)$! Es müssen immer nur die Terme der höchsten Ordnung betrachtet werden, und diese ohne etwaige konstante Faktoren.

Bachmann-Landau Notation



Funktion $f(n) \in O(g(n))$. x-Achse: n, y-Achse: Funktionswerte $f(n), g(n), c \cdot g(n)$.

Bachmann-Landau Notation (2)

- Obere Schranke:

$$O(g(n)) = \{f(n) \mid (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq f(n) \leq cg(n)\}$$

- Untere Schranke:

$$\Omega(g(n)) = \{f(n) \mid (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq cg(n) \leq f(n)\}$$

- Genaue Wachstumsrate:

$$\Theta(g(n)) = \{f(n) \mid (\exists c_1, c_2, n_0 > 0), (\forall n \geq n_0) : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Rechenregeln

Addition

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

Multiplikation

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

Rechenregeln

Beispiel: Verschachtelte Schleifen:

```
for i = 1 to n:                  // O(n)
    for j = 1 to n:              // O(n)
        konstante Operation      // O(1)
```

- Gesamt: $O(n) \cdot O(n) \cdot O(1) = O(n^2)$
- Ebenso ist die Laufzeit in $\Theta(n^2)$

Rechenregeln

Beispiel: Lineare Suche: Suche nach Element x in Array der Größe n:

- Best-Case: x steht an erster Position $\Rightarrow O(1)$
- Worst-Case: x ist letzte Position oder nicht vorhanden $\Rightarrow O(n)$
- Average-Case (gleichverteilte Position): erwartete Position $n/2 \Rightarrow O(n)$

Die exakte zu erwartende Vergleichszahl ist $\approx n/2$, asymptotisch $\Theta(n)$.

Beispiele

Beispiele:

- $3n^2 \in O(n^2)$
- $\frac{3}{4}n^3 + 5n^2 \in O(n^3)$
- $42n + \ln n + 5150 \in O(n)$
- $n \log n \in O(n^2)$
- $n \log n \in O(n \log n)$ (!) die kleinste obere Schranke!
- $n + m^2 \in O(n + m^2)$

Amortisierte Analyse

- klassische Analyse: Kosten einer einzelnen Operation isoliert
- Manche Datenstrukturen haben jedoch einzelne teure Operationen, die durch viele kostengünstige ausgeglichen werden.
- **Amortisierte Laufzeit** einer Operation: *durchschnittliche Laufzeit* pro Operation über eine Folge von Operationen, wobei teure Operationen durch viele günstigere ausgeglichen werden.

- Wir betrachten einfache Sortieralgorithmen als Beispiele für Algorithmen
- Ziel: Sortiere die Zahlen in einem Array (aufsteigend)

Array mit 10 Integer-Elementen

9	4	12	1	7	3	10	5	2	8
0	1	2	3	4	5	6	7	8	9

Abbildung: Array mit 10 Elementen (Index unter den Zellen).

Bubblesort

Algorithmus 1: Bubblesort

Input: Array a

Result: Sortiertes Array a

```
1 Function BubbleSort(a)
2     for i = 0; i < a.length − 1; i++ do
3         swapped = false
4         for j = 0; j < a.length − 1 − i; j++
           do
5             if a[j] > a[j + 1] then
6                 Swap(a, j, j+1)
7                 swapped = true
8         if notswapped then
9             Break
```

Bubblesort (Beispiel)

Durchlauf i	Array-Zustand
Initial	9 4 12 1 7 3 10 5 2 8
0	4 9 1 7 3 10 5 2 8 12
1	4 1 7 3 9 5 2 8 10 12
2	1 4 3 7 5 2 8 9 10 12
3	1 3 4 5 2 7 8 9 10 12
4	1 3 4 2 5 7 8 9 10 12
5	1 3 4 2 5 7 8 9 10 12
6	1 3 2 4 5 7 8 9 10 12
7	1 2 3 4 5 7 8 9 10 12
8	1 2 3 4 5 7 8 9 10 12 (sortiert)

Insertionsort

Algorithmus 2: Insertionsort

Input: Array a

Result: Sortiertes Array a

```
1 Function InsertionSort(a)
2     i = 1
3     while i < a.length do
4         j = i
5         while j > 0 ∧ a[j − 1] > a[j] do
6             Swap(a, j, j-1)
7             j = j − 1
8         i = i + 1
```

Insertionsort (Beispiel)

i (nächster Index)	Array-Zustand (linker Teil sortiert)
Initial	9 4 12 1 7 3 10 5 2 8
1	4 9 12 1 7 3 10 5 2 8
2	4 9 12 1 7 3 10 5 2 8
3	1 4 9 12 7 3 10 5 2 8
4	1 4 7 9 12 3 10 5 2 8
5	1 3 4 7 9 12 10 5 2 8
6	1 3 4 7 9 10 12 5 2 8
7	1 3 4 5 7 9 10 12 2 8
8	1 2 3 4 5 7 9 10 12 8
9	1 2 3 4 5 7 8 9 10 12 (sortiert)

Selectionsort

Algorithmus 3: Selectionsort

Input: Array a

Result: Sortiertes Array a

```
1 Function SelectionSort(a)
2     for i = 0; i < a.length − 1; i++ do
3         jMin = i
4         for j = i + 1; j < a.length; j++ do
5             if a[j] < a[jMin] then
6                 jMin = j
7         if jMin ≠ i then
8             Swap(a, i, jMin)
```

Selectionsort

i	Array-Zustand
Initial	9 4 12 1 7 3 10 5 2 8
0	1 4 12 9 7 3 10 5 2 8
1	1 2 12 9 7 3 10 5 4 8
2	1 2 3 9 7 12 10 5 4 8
3	1 2 3 4 7 12 10 5 9 8
4	1 2 3 4 5 12 10 7 9 8
5	1 2 3 4 5 7 10 12 9 8
6	1 2 3 4 5 7 8 12 9 10
7	1 2 3 4 5 7 8 9 12 10
8	1 2 3 4 5 7 8 9 10 12 (sortiert)

Laufzeitanalyse

Worst-Case Analyse einfacher Sortieralgorithmen bei n Elementen:

Algorithmus	**Vergleiche**	**Swaps**	**Gesamt**
Bubblesort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertionsort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selectionsort	$O(n^2)$	$O(n)$	$O(n^2)$

Die Gesamtlaufzeit von $O(n^2)$ ist in der Praxis problematisch, d.h. zu langsam. Daher kommen üblicherweise bessere Algorithmen zum Einsatz, die eine Gesamtlaufzeit von $O(n \log n)$ aufweisen.