Grundlagen der Informatik

- Einführung in Berechenbarkeit und Komplexität -

Prof. Dr. Klaus Volbert

Hochschule für angewandte Wissenschaften
Fakultät Informatik und Mathematik

Wintersemester 2010/11
Regensburg, 12./13. Januar 2011

Entwicklung von Algorithmen

HOCHSCHULE REGENSBURG UNIVERSITY OF APPLIED SCIENCES

- Algorithmen stehen im Mittelpunkt der Informatik
- Hauptziel beim Entwurf von Algorithmen
  - Korrekte Problemlösung (totale Korrektheit)
    - Terminiertheit: Der Algorithmus endet für jede spezifizierte Eingabe
    - Partielle Korrektheit: Der Algorithmus liefert für jede spezifizierte Eingabe das geforderte Ergebnis
- Nebenziel beim Entwurf von Algorithmen
  - Effiziente Problemlösung hinsichtlich Zeit und Platz
- Interessant sind meist nur effiziente Algorithmen
- Wesentliche Effizienzmaße
  - Rechenzeitbedarf
    - Zeitkomplexität: Zählen der atomaren Schritte
  - Speicherplatzbedarf
    - Platzkomplexität: Zählen der verwendeten Speicherzellen

Komplexität von Algorithmen

HOCHSCHULE REGENSBURG UNIVERSITY OF APPLIED SCIENCES

- Es gibt unendlich viele Algorithmen zur Lösung von Problemen (in Wissenschaft, Technik, Wirtschaft, …)
- Funktional gleichwertige Algorithmen können sich erheblich in der Effizienz/Komplexität unterscheiden
- Ein Algorithmus ist umso effizienter, je weniger er von den beiden Ressourcen Rechenzeit und Speicherplatz verwendet
- Komplexität hängt u.a. von der Eingabe für das Programm ab
- Faktoren zur Bestimmung der Komplexität
  - Messungen auf einer konkreten Maschine
  - Aufwandsermittlung in einem idealisierten Rechnermodell (z.B. RAM)
  - Asymptotische Komplexitätsabschätzung durch ein abstraktes Komplexitätsmaß in Abhängigkeit der Problem-/Eingabegröße

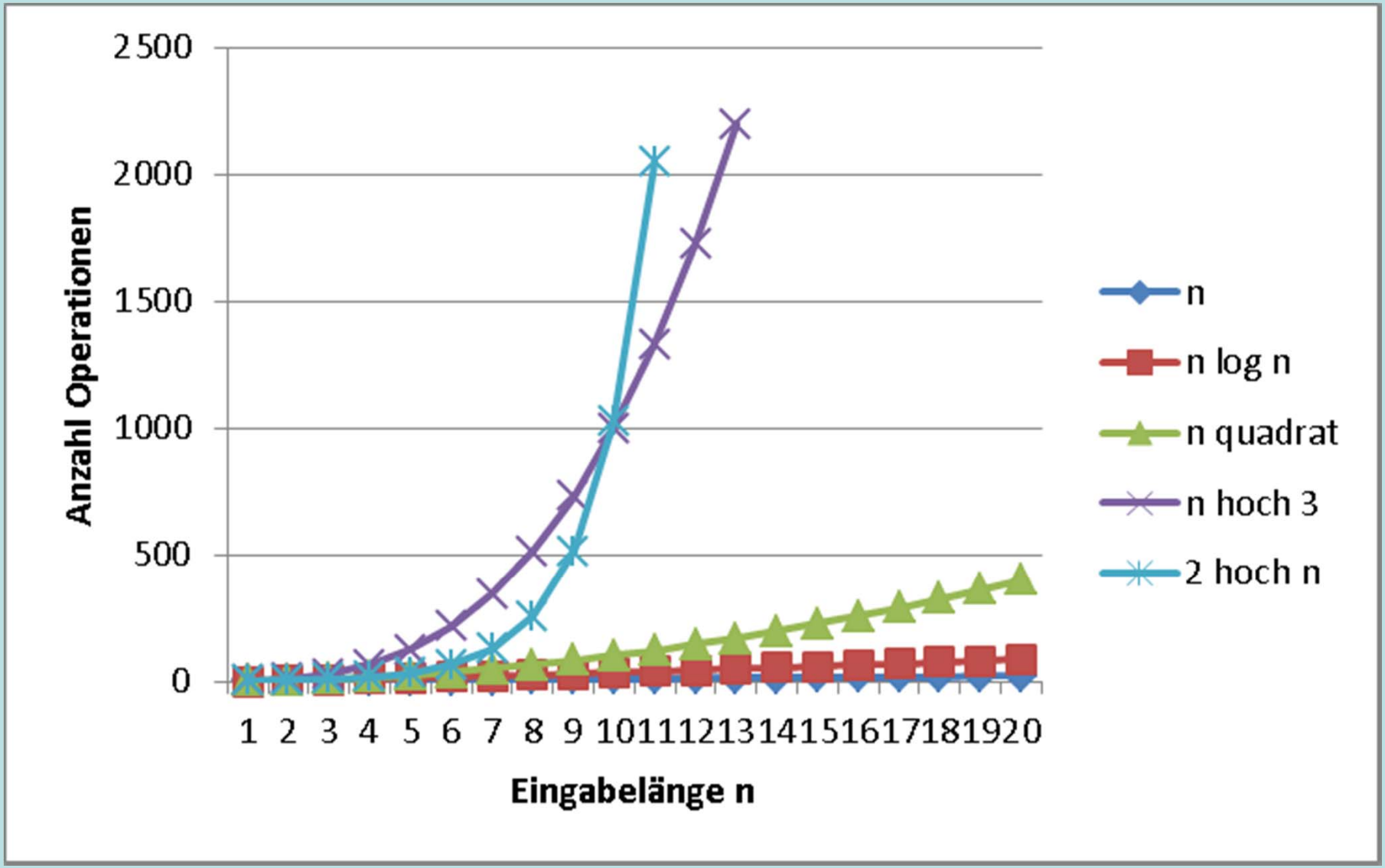
Beispiel: Laufzeitkomplexität

- Annahme:
  - Rechner kann 1.000 Operationen pro Sekunde ausführen (1.000 Hz)
  - Die folgende Tabelle zeigt jeweils die mögliche Problemgröße (Wert für n) bei vorgegebener Rechenzeit

Komplexität		1 Sekunde	1 Minute	1 Stunde	1 Tag	1 Woche
Linear	n	1.000	60.000	3.600.000	86.400.000	604.800.000
Super-Linear	$n \log_2 n$	140	4.895	204.094	3.943.234	24.631.427
Polynomiell	n^2	31	244	1897	9.295	24.592
	n^3	10	39	153	442	845
Exponentiell	2^n	9	15	21	26	29

- Effekt der technischen Entwicklung ist
  - bei linearem Laufzeitverhalten: ideal
  - bei polynomiellen Laufzeitverhalten: akzeptabel
  - bei exponentiellem Laufzeitverhalten: sehr schlecht

Wesentliche Wachstumsfunktionen



Asymptotische Kostenmaße (Landau-Symbole, O-Notation)

HOCHSCHULE REGENSBURG UNIVERSITY OF APPLIED SCIENCES

- Größenordnung der Komplexität in Abhängigkeit der Eingabegröße
  - Best Case, Worst Case, Average Case
- Seien $f, g: \mathrm{IN} \to \mathrm{IR}$ zwei beliebige Funktionen, dann definieren wir
  - Groß-O-Notation (O-Notation, O-Kalkül), Obere Schranke:

g wächst höchstens wie f

$$O\big(f(n)\big) = \{\, g(n) \mid \exists\, c \in \mathrm{IR}^+, n_0 \in \mathrm{IN}: \forall\, n \geq n_0, g(n) \leq c \cdot f(n) \,\}$$

- Verwendung des Logarithmus im O-Kalkül:
  - Mit $\log(n)$ ist $\log_2(n)$ gemeint, d.h. der Logarithmus zur Basis 2
  - Zielbasis a ist unwesentlich, da:

$$\log_b n = \frac{\log_a n}{\log_a b} = \frac{1}{\log_a b} \log_a n \;\; (\log_a b \text{ ist konstant})$$

Anmerkungen zur O-Notation

HOCHSCHULE REGENSBURG UNIVERSITY OF APPLIED SCIENCES

- Abstrakte Abschätzung zur Klassifizierung
  - Extrahiert dominante Terme (Optimierung von Konstanten oft ineffektiv)
  - Abstrahiert von Konstanten (Versteckte Konstanten können groß sein!)
  - Vernachlässigt „geringfügige“ Terme
- Rechenregeln

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \leq c \Rightarrow g(n) \in O(f(n))$$

  - $\forall f: f(n) \in O(f(n))$
  - $\forall f: \forall\, c \in \mathrm{IR}^+: c\, f(n) \in O\big(f(n)\big)$
  - $\forall g \in O\big(f(n)\big): \forall\, c \in \mathrm{IR}^+: g(n) + g(n) \in O\big(f(n)\big),\ c\, g(n) \in O\big(f(n)\big)$
  - $\forall g_1 \in O(f_1(n)): \forall g_2 \in O(f_2(n)): g_1(n) + g_2(n) \in O(f_1(n) + f_2(n))$
  - $\forall g_1 \in O(f_1(n)): \forall g_2 \in O(f_2(n)): g_1(n) + g_2(n) \in O(\max(f_1(n), f_2(n)))$
  - $\forall g_1 \in O(f_1(n)): \forall g_2 \in O(f_2(n)): g_1(n)\, g_2(n) \in O(f_1(n)\, f_2(n))$

Erweiterungen der O-Notation

- Seien $f, g: \mathrm{IN} \to \mathrm{IR}$ zwei beliebige Funktionen, dann definieren wir
  - Groß-Omega-Notation, Untere Schranke:

g wächst mindestens wie f

$$\Omega(f(n)) = \{\, g(n) \mid \exists\, c \in \mathrm{IR}^+, n_0 \in \mathrm{IN}: \forall\, n \geq n_0, g(n) \geq c \cdot f(n) \,\}$$

  - Groß-Theta-Notation, Exakte Schranke:

g wächst wie f

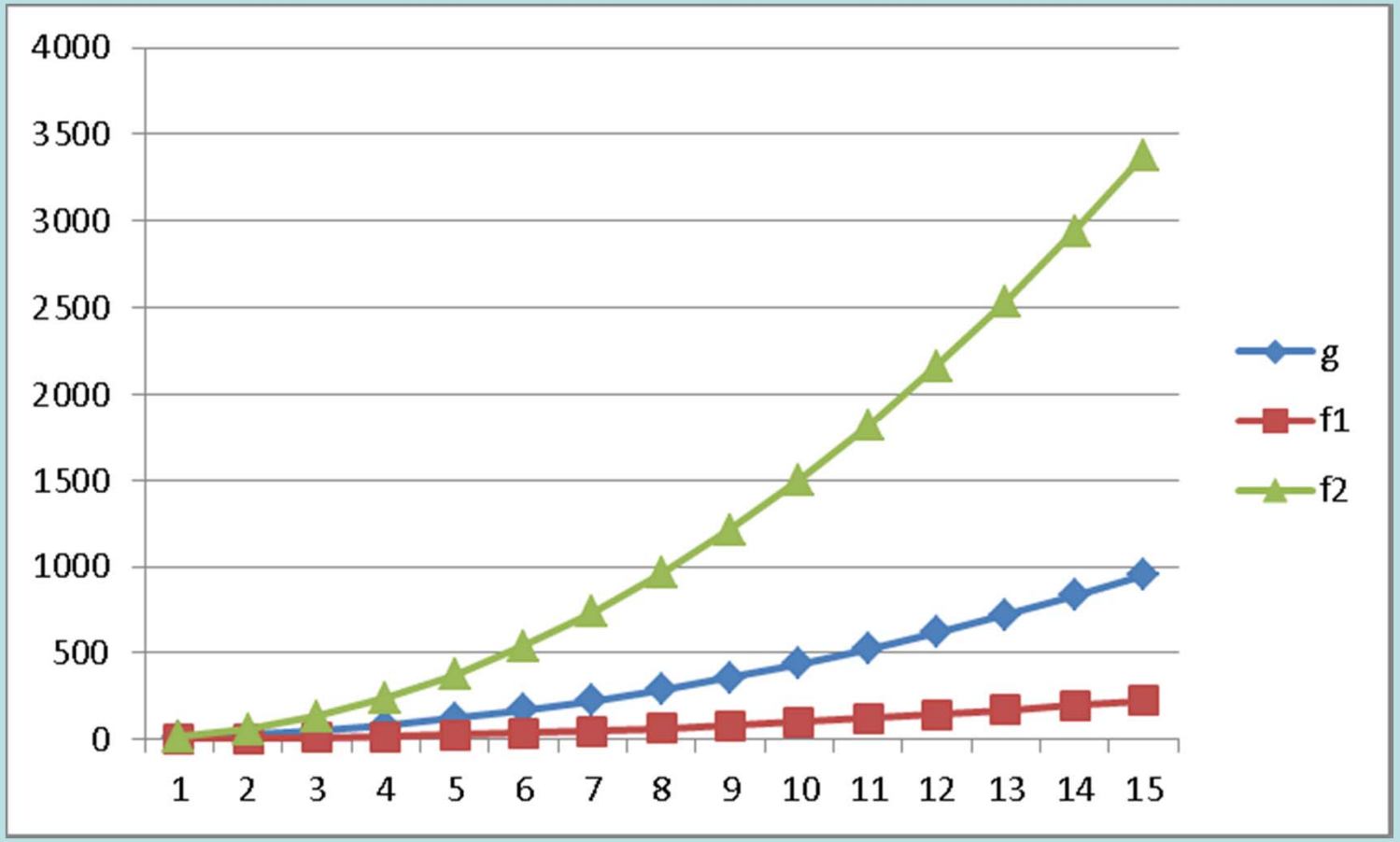
$$\Theta(f(n)) = \{\, g(n) \mid g(n) \in O(f(n)) \text{ und } g(n) \in \Omega(f(n)) \,\}$$

- Schreibweise:
  - Statt $g(n) \in O(f(n))$ wird auch $g(n) = O(f(n))$ (analog für Ω und Θ)
  - Vorsicht beim Umgang, da es sich nicht um ein echtes „=“ handelt

Beispiel zur Θ-Notation

- Betrachte $g(n) = 4n^2 + 3n + 7$
- Es gilt $g(n) = \Theta(n^2)$, da für alle n:

$$f_1(n) := n^2 \leq g(n) \leq 15n^2 =: f_2(n)$$



Beispiel MaxTeilSum

- Eingabe: $a_1, \ldots, a_n \in \mathrm{Z}$ (n ganze Zahlen)
- Ausgabe: Maximale Teilsumme, d.h.

$$s = \max_{1 \le i \le j \le n} \sum_{k=i}^{j} a_k$$

- Anwendungen
  - Erkennung von grafischen Mustern
  - Analyse von Aktienkursen
    (täglich neue Kurse: Ermittlung bester Ein-/Ausstiegszeitpunkt)
- Beispiel:
  - Eingabe: -13, 25, 34, 12, -3, 7, -87, 28, -77, 11
  - Ausgabe: 75 (ergibt sich aus $i = 2, j = 6$)

MaxTeilSum1

- Naiver Algorithmus: Durchlaufen aller möglichen Varianten

```
int MaxTeilsum1(int a[],int n) {
      int i, j, k, sum, max = -4294967296;

      for (i=0; i<n; i++) {
            for (j=i; j<n; j++) {
                  sum=0;
                  for (k=i; k<=j; k++) sum += a[k];
                  if (sum > max) max = sum;
            }
      }

      return max;
}
```

- Laufzeit: $O(n^3)$ (kubisch)

MaxTeilSum2

- Verbesserung: Statt immer wieder vollständige Summe zu berechnen: Erweiterung um den aktuellen Summanden

```
int MaxTeilsum2(int a[],int n) {
      int i, j, sum, max = -4294967296;

      for (i=0; i<n; i++) {
            sum=0;
            for (j=i; j<n; j++) {
                  sum += a[k];
                  if (sum > max) max = sum;
            }
      }

      return max;
}
```

- Laufzeit: $O(n^2)$ (quadratisch)

MaxTeilSum3

- Verbesserung: Berechnung des aktuellen Teilstücks. Sollte das aktuelle Teilstück einen negativen Wert annehmen, wird der Wert auf 0 gesetzt, ansonsten entsprechend erhöht

```
int MaxTeilsum3(int a[],int n) {
      int i, s, max = -4294967296, aktSum = 0;

      for (i=0; i<n; i++) {
            s = aktSum + a[i];
            if (s > 0) aktSum = s;
            else aktSum = 0;
            if (aktSum > max) max = aktSum;
      }

      return max;
}
```

- Laufzeit: $O(n)$ (linear)

Übersicht Laufzeitverhalten MaxTeilSum