LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

Algorithmen und Datenstrukturen

Kapitel 1: Algorithmen und ihre Analyse

Skript zur Vorlesung

Algorithmen und Datenstrukturen

Sommersemester 2015

Ludwig-Maximilians-Universität München

basierend auf dem Skript von Prof. Dr. Martin Ester, Prof. Dr. Daniel A. Keim, Dr. Michael Schiwietz und Prof. Dr. Thomas Seidl

Der Begriff des Algorithmus I

- **Algorithmus (Definition):**
  - **präzise, endliche** Verarbeitungsvorschrift
  - **Elementaroperationen** werden von einer mechanisch oder elektronisch arbeitenden Maschine durchgeführt.
- **Abfolge** der einzelnen Verarbeitungsschritte müssen **eindeutig** hervorgehen.
- **Wahlmöglichkeiten** sind zugelassen, deren Auswahl jedoch genau festliegen muss.
- Ausführung eines Algorithmus (**Prozess**) geschieht durch ein Ausführungsorgan (**Prozessor**)
- Abarbeitung der spezifizierten Elementaroperationen werden in der **festgelegten Reihenfolge abgearbeitet**.

Beispiele für „Alltagsalgorithmen“

Prozess	Algorithmus	Typische Schritte im Algorithmus
Pullover stricken	Strickmuster	Stricke Rechtsmasche, stricke Linksmasche
Modellflugzeug bauen	Montageanleitung	Leime Teil A an den Flügel B
Kuchen backen	Rezept	Nimm 3 Eier, schaumig schlagen
Kleider nähen	Schnittmuster	Nähe seitlichen Saum
Beethovensonate spielen	Notenblatt	♪♫♫♪♫♪♪♪♫♪♫

- Ähneln Algorithmen, sind aber selten exakt ausformuliert.
- Häufig sind Teile enthalten, die mehrdeutig interpretiert werden können.

Hier in der Vorlesung:

- nur **Computer-Algorithmen**
- Verwendung von **Java** zur Notation

Der Begriff des Algorithmus II

- Durch einen Algorithmus werden mittels einer Reihe von Anweisungen Objekte manipuliert, um von einer spezifischen Eingabe eine spezifische Ausgabe zu erhalten.

- **Formal:** Ein Algorithmus beschreibt eine Abbildung

$$f: E \to A$$

E: Menge der zulässigen Eingabedaten
A: Menge der Ausgabedaten.

- **aber:** Nicht jede Abbildung $f: E \to A$ lässt sich durch einen Algorithmus realisieren (Berechenbarkeit!)

Typische Eigenschaften von Algorithmen I

Abstrahierung:

- Ein Algorithmus löst eine Klasse von Problemen
- Wahl eines konkreten, aktuell zu lösenden Problems aus dieser Klasse erfolgt über geeignete Wahl von Parametern

Finitheit:

- **Statische Finitheit:** Beschreibung eines Algorithmus besitzt eine endliche Länge.
- **Dynamische Finitheit:** Algorithmus darf zu jedem Zeitpunkt nur endlich viel Platz belegen.

→ endliche Datenstrukturen und Zwischenergebnisse

Typische Eigenschaften von Algorithmen II

Terminierung:

- Algorithmen sind **terminierend**, die für jede Eingabe nach endlich vielen Schritten ein Resultat liefern und anhalten, ansonsten sind sie **nicht-terminierend**.
- Beispiele für nicht-terminierende Algorithmen: Betriebssysteme, Überwachung von Anlagen, Verkehrsampeln, Befehlsholezyklus in CPU

Determinismus:

- Algorithmen sind **deterministisch**, wenn zu jedem Zeitpunkt ihrer Ausführung höchstens eine Möglichkeit der Fortsetzung besteht, ansonsten sind sie **nicht-deterministisch**.

Typische Eigenschaften von Algorithmen III

Determiniertheit:

- Normalerweise liefert ein Algorithmus stets das gleiche Ergebnis, wenn er mit den gleichen Eingabewerten und Startbedingungen wiederholt wird.
- Nicht-determinierte Algorithmen können nützlich sein, wenn exakte Lösungsalgorithmen eine hohe Komplexität haben → heuristische Methoden (Verzicht auf erschöpfende Bearbeitung aller Fälle)

- Ein terminierender, deterministischer Algorithmus ist immer determiniert.
- Ein terminierender, nicht-deterministischer Algorithmus kann determiniert oder nicht-determiniert sein.

Motivation

- **Algorithmen als Problemlösung:**
  Entwicklung neuer Algorithmen zur Behebung ungelöster Probleme/Aufgaben

- **Steigerung der Effizienz von Algorithmen:**
  Entwicklung möglichst guter Algorithmen.

- **Fragestellungen der Informatik bei Algorithmen:**
  - Man kann beweisen, dass es zu jedem Algorithmus unendlich viele, äquivalente Algorithmen gibt, die die gleiche Aufgabe lösen.
  - Dabei interessant für Informatiker:
    - Suche nach schnelleren oder kompakteren Algorithmen
    - Beweis, dass es solche nicht geben kann

Grundlegende Bestandteile von Algorithmen I

Zur Ausführung eines Algorithmus benötigt man verschiedene Arten von **Kontrollstrukturen**:

Sequenz (Folge von Anweisungen)

- Zu einem Zeitpunkt wird nur ein Schritt ausgeführt
- Jeder Schritt wird genau einmal ausgeführt
- Die Schritte werden nacheinander ausgeführt
- Mit der Beendigung des letzten Schrittes endet der gesamte Algorithmus

- In Java:
  `{<Anweisung>; <Anweisung>; ...; <Anweisung>;}`

Grundlegende Bestandteile von Algorithmen II

Selektion (Auswahl, bedingte Anweisung)

- Einfache Form:

Pseudo-Code:	**Java:**

Falls Bedingung
  dann Sequenz
	`if (<Bedingung>)`
	  `<Sequenz>`

- Bedingte Anweisung mit Alternative (allgemeine Form)

Pseudo-Code:	**Java:**

Falls Bedingung
  dann Sequenz 1
  sonst Sequenz 2
	`if (<Bedingung>)`
	  `<Sequenz 1>`
	  `else <Sequenz 2>`

Anmerkung:
Einfache Form ist Spezialfall der allgemeinen Form

Grundlegende Bestandteile von Algorithmen III

- Mehrfachauswahl:

Pseudo-Code:	**Falls** Bedingung 1 **dann** Anweisung(en) Bedingung 2 **dann** Anweisung(en) ... Bedingung n **dann** Anweisung(en) **sonst** Anweisung(en)

Java:

```
switch (<Ausdruck>) {
  case (<Konstante 1>):
        <Anweisung>, ... , <Anweisung>; break;
  case (<Konstante 2>):
        <Anweisung>, ... , <Anweisung>; break;
 ...
  case (<Konstante n>):
        <Anweisung>, ... , <Anweisung>; break;
  default: <Anweisung>, ... , <Anweisung>;
```

Grundlegende Bestandteile von Algorithmen IV

Iteration (Wiederholung, Schleife)

- Erste Form der Iteration:

Pseudo-Code:	**Java:**
Wiederhole Sequenz **bis** Bedingung *{Abbruchbedingung}*	`do <Sequenz>` `while (!<Bedingung>)`

- Zweite Form der Iteration:

Pseudo-Code:	**Java:**
Solange Bedingung Sequenz *{Rumpf der Schleife}*	`while (<Bedingung>)` `<Sequenz>`

Unterschiede: Bei der zweiten Form wird die Bedingung vor der Ausführung des Rumpfes geprüft (wichtig, falls Bedingung vor dem Eintritt in die Schleife geprüft werden soll)

Grundlegende Bestandteile von Algorithmen V

Iteration (Wiederholung, Schleife)

- Spezialfall: Endlosschleife

Pseudo-Code:	**Java:**
Wiederhole Sequenz **immer**	`while (true)` `<Sequenz>`

Anmerkungen:

- Unbeabsichtigte Endlosschleifen entstehen häufig, weil die Abbruchbedingung nicht korrekt formuliert wurde.
- Sequenz, Selektion und Iteration genügen, um **jeden Algorithmus** auszudrücken!

Effizienz und Komplexität

- **Effizienz** als wichtiges Kriterium zum Vergleich verschiedener Algorithmen zur Lösung ein und desselben Problems.
- Wird bestimmt durch den benötigten Aufwand des Algorithmus (seine **Komplexität**) in Abhängigkeit einer speziellen Eingabesituation
- Wesentliche Effizienzkriterien:
  - Die **Laufzeit des Algorithmus**
  - Der benötigte **Speicherplatz**
- Laufzeit ist in der Regel das wichtigste Kriterium

Häufig: „Trade-Off“ bei der Optimierung eines dieser beiden Kriterien dahingehend, dass das andere Kriterium verschlechtert wird.

Laufzeitanalyse I

- 1. Ansatz: Direktes Messen der **Laufzeit** (z.B. Millisekunden)
  - abhängig von vielen Parametern (Rechenkonfiguration, Rechnerlast, Compiler, Betriebssystem, Programmiertricks, u.a.)
  - Daher kaum übertragbar und ungenau
- 2. Ansatz: Zählen der benötigten **Elementaroperationen** in Abhängigkeit von der jeweiligen Größe der Eingabe
  - **Algorithmische Verhalten** wird als Funktion der benötigten Elementaroperationen dargestellt
  - Charakterisierung ist abhängig von der jeweiligen Problemstellung und dem zugrundliegenden Algorithmus
  - Beispiele für Elementaroperationen: Zuweisungen, Vergleiche, arithmetische Operationen, Zeigerdereferenzierungen, Arrayzugriffe

Laufzeitanalyse II

- **Maß** für die Größe der Eingabe ist abhängig von der Problemstellung

Beispiel:

Problem:	**Größe der Eingabe:**
Suche eines Elementes in einer Liste	Anzahl der Elemente
Multiplikation zweiter Matrizen	Dimension der Matrizen
Sortierung einer Liste von Zahlen	Anzahl der Zahlen

Beispiel: Laufzeitanalyse I

13	7	5	23	8	18	17	31	3	11	9	30	24	27	21	19

```
int seqsearch(int[] a, int x) {
    int i = 0, high = a.length - 1;
    while ((i <= high) && (a[i] != x))
        i++;
    if (i <= high) return i; else return -1;
}
```

→ Wesentliche Operationen (**Grundoperationen**):
Ausführungen der **while**-Schleife

Diese Anzahl ist **abhängig** von:

- Größe des Arrays (n; fest vorgegeben)
- Position des gesuchten Elementes im Array (variabel)

Laufzeitanalyse III

Wir unterscheiden daher zwischen:

- dem **durchschnittlichen Zeitbedarf** $T_{\emptyset}(n)$ eines Algorithmus, charakterisiert durch die durchschnittliche Anzahl $A_{\emptyset}(n)$ benötigter Grundoperationen für alle Eingaben der Größe n.
- Den **Zeitbedarf im schlechtesten Fall** $T_{worst}(n)$, charakterisiert durch $A_{worst}(n)$, die Anzahl benötigter Grundoperationen im schlechtesten Fall aller Eingaben der Größe n.

Laufzeitanalyse VI

Sei E_n die Menge aller möglichen Eingaben der Größe n und $a(e), e \in E_n$, die Anzahl von Grundoperationen, die ein gegebener Algorithmus bei Eingabe von e ausführt.

Sei weiterhin $p(e)$ die Wahrscheinlichkeit, mit der die Eingabe e auftritt ($\sum_{e \in E_n} p(e) = 1$).

Dann gilt:

$$A_{\emptyset}(n) = \sum_{e \in E_n} p(e) \cdot a(e)$$

$$A_{worst}(n) = \max\big(a(e)\big)$$

Beispiel: Laufzeitanalyse II

- Sei q die Wahrscheinlichkeit, dass x im Array vorhanden ist und sei jede Position für x gleichwahrscheinlich.
- Bezeichne weiterhin e_i, $0 \leq i \leq n-1$, die Menge aller Eingaben mit $x = A[i]$ und entsprechend e_n die Menge aller Eingaben, die x nicht enthalten.

- Dann gilt:
  $p(e_i) = \frac{q}{n}$, $0 \leq i \leq n-1$ und $p(e_n) = 1 - q$

Beispiel: Laufzeitanalyse III

- Hieraus ergibt sich:

$$A_{\emptyset}(n) = \sum_{e \in E_n} p(e) \cdot a(e) = \sum_{i=0}^{n} p(e_i) \cdot a(e_i) = \left(\sum_{i=0}^{n-1} \frac{q}{n} \cdot (i+1) \right) + (1-q) \cdot n =$$

$$= \frac{q}{n} \cdot \left(\sum_{i=1}^{n} i \right) + (1-q) \cdot n = \frac{q}{n} \cdot \frac{n \cdot (n+1)}{2} + (1-q) \cdot n =$$

$$= q \cdot \frac{n+1}{2} + (1-q) \cdot n$$

Im Fall $q = 1$: $A_{\emptyset}(n) = \frac{n+1}{2}$

Im Fall $q = \frac{1}{2}$: $A_{\emptyset}(n) = \frac{n+1}{4} + \frac{n}{2} \approx \frac{3}{4} \cdot n$

$$A_{worst}(n) = \underbrace{\max\left(a(e_i)\right)}_{0 \le i \le n} = n$$

Landau-Symbole

- Durch Weglassen multiplikativer und additiver Konstanten wird allein das **Wachstum der Laufzeitfunktion** $T(n)$ betrachtet.
- Man erhält eine von der Programmumgebung und anderen äußeren Einflussgrößen **unabhängige Charakterisierung** der (asymptotischen) Komplexität des Algorithmus.
- Die Komplexität kann durch asymptotische obere und untere Schranken definiert werden.

→ **Landau-Symbole**

Definition Landau-Symbol Θ

Sei $g: \mathfrak{R} \to \mathfrak{R}$ eine Funktion.

Das Landau-Symbol $\Theta(g)$ ist definiert als die Menge

$$\Theta(g) := \{ f: \mathfrak{R} \to \mathfrak{R} \mid \exists\, c_1 > 0, c_2 > 0, n_0 \in \mathbb{N}, \text{ so dass } \forall\, n \geq n_0: \\ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$$

Gebräuchliche Schreibweise:

$f = \Theta(g)$ oder $f(n) = \Theta\big(g(n)\big)$ statt $f \in \Theta(g)$

g ist die asymptotische obere **und** untere Schranke von f.
Oft ist jedoch nur die obere Schranke interessant.

Definition Landau-Symbol O

Sei $g: \mathfrak{R} \to \mathfrak{R}$ eine Funktion.

Das Landau-Symbol $O(g)$ ist definiert als die Menge

$O(g) := \{ f: \mathfrak{R} \to \mathfrak{R} \mid \exists\, c > 0,\ n_0 \in \mathbb{N}$, so dass $\forall\, n \geq n_0$:

$$0 \leq f(n) \leq c \cdot g(n) \}$$

Gebräuchliche Schreibweise:

$f = O(g)$ oder $f(n) = O\big(g(n)\big)$ statt $f \in O(g)$

g ist die asymptotische obere Schranke von f.

(„ f wächst **höchstens** so schnell wie g“)

→ aus $f = \Theta(g)$ folgt $f = O(g)$.

Definition Landau-Symbol Ω

Analog lässt sich die untere Schranke (Landau-Symbol Ω) definieren. Ferner existieren die Landau-Symbole o und ω, welche hier nicht weiter vorgestellt werden.

Beispiele für $f = \Theta(g)$, $f = O(g)$ und $f = \Omega(g)$:

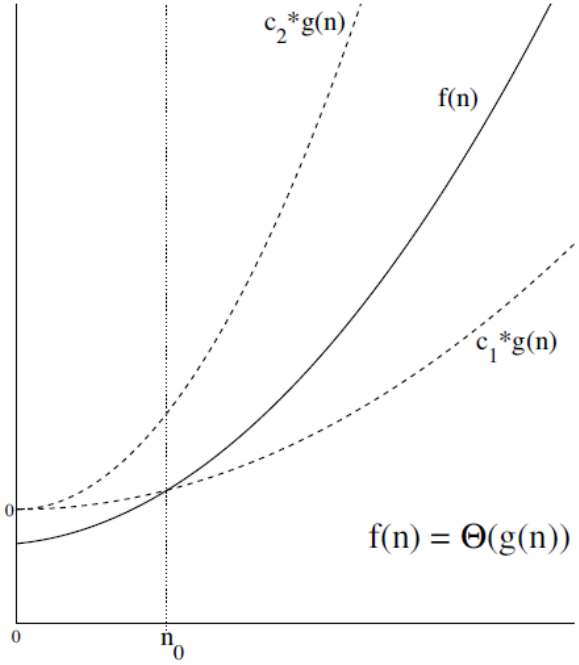
Mit dieser Notation gilt:

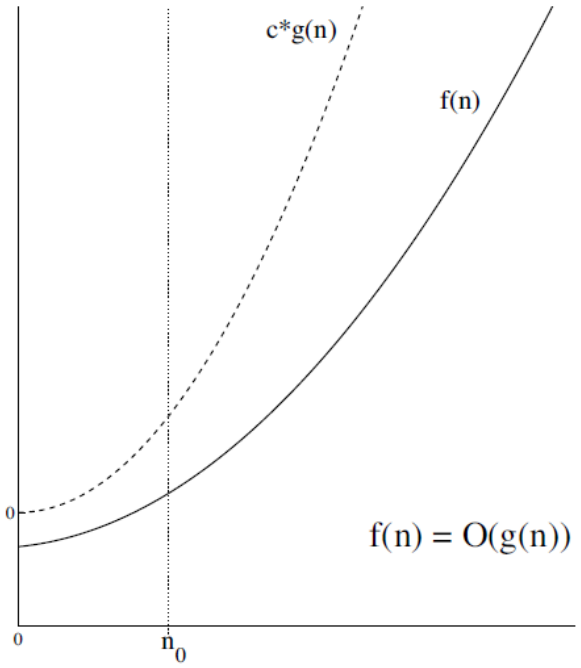
$$T_{\emptyset}(n) = O\left(A_{\emptyset}(n)\right) \text{ sowie } T_{worst}(n) = O\left(A_{worst}(n)\right)$$

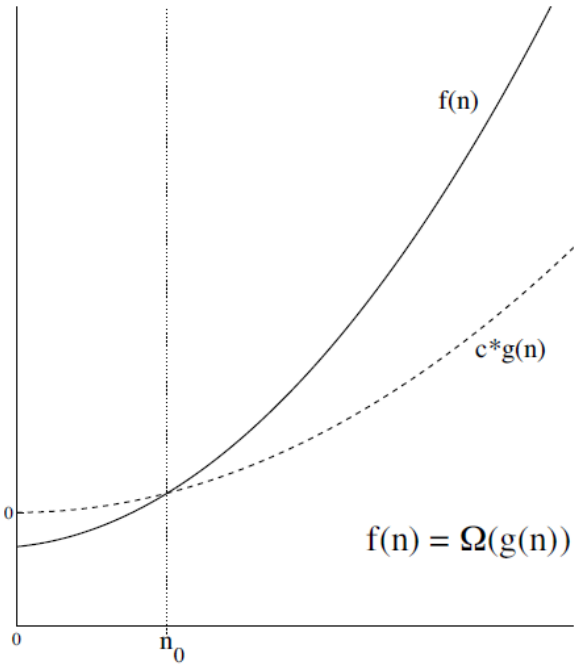
Im Beispiel ist also:

$$T_{\emptyset}(n) = O\left(q \cdot \frac{n+1}{2} + (1-q) \cdot n\right) = O(n) \quad \text{und } T_{worst}(n) = O(n)$$

Vergleich der Landau-Symbole







Alternative Definition Landau-Symbol O

- Obere Schranke (Landau Symbol O) über die Existenz von Grenzwerten definierbar:
- Funktionen bei der Laufzeitanalyse meist
  - monoton wachsend
  - von 0 verschieden
- Betrachtung des Quotienten $\frac{f(n)}{g(n)}$.
- Nach Definition gilt für $f = O(g)$: $\frac{f(n)}{g(n)} \leq c$, $n \geq n_0$.
- Existiert Grenzwert $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ (d.h. $< \infty$) $\Rightarrow$ $f = O(g)$.

Rechnen mit der O-Notation

Optimalität von Algorithmen:

- Ein Algorithmus A ist eine **(worst-case) optimale Lösung** eines gegebenen Problems, falls für jeden Lösungsalgorithmus B aus der Algorithmenklasse von B gilt:

$$T^A{}_{worst} = O(T^B{}_{worst})$$

Rechnen mit der O-Notation:

- Elimination von Konstanten:

$$2 \cdot n = O(n), \qquad \frac{n}{2} + 1 = O(n)$$

- Bilden oberer Schranken:

$$2 \cdot n = O(n^2), \qquad 3 = O(\log n)$$

Wichtige Klassen von Funktionen

	Sprechweise	Typische Algorithmen
$O(1)$	Konstant	
$O(\log n)$	Logarithmisch	Suchen auf einer Menge
$O(n)$	Linear	Bearbeiten jedes Elementes einer Menge
$O(n \cdot \log n)$		Gute Sortierverfahren, z.B. Heapsort
$O(n \cdot \log^2 n)$		
…		
$O(n^2)$	Quadratisch	Primitive Sortierverfahren
$O(n^k), k \geq 2$	Polynomiell	
…		
$O(2^n)$	Exponentiell	Backtracking-Algorithmen