
Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Tanya Braun (Übungen)

sowie viele Tutoren



Allgemeine Lernziele **Vorlesung**

Einem Vortragenden **zuhören zu lernen**,

... der über ein nicht ganz triviales Thema referiert

Dem Vortragenden beim Vortrag **gedanklich folgen** können

- Dieses Lernziel kann man kaum zuhause erwerben.

Vorlesung dient der Vorbereitung

für das Erarbeiten der Inhalte durch

- Lösen von Übungsaufgaben
- Formulierung von Fragen
- Diskussion in Übungsgruppe

Organisatorisches: Übungen

- **Start:** Montag, 8. April 2018
- **Übungen:** Montags
Anmeldung über Moodle nach dieser Veranstaltung (18.00 h)
- **Übungsaufgaben** stehen jeweils kurz nach der Vorlesung am Freitag über Moodle bereit
- Aufgaben sollen in einer **2-er Gruppe** bearbeitet werden
- **Abgabe der Lösungen** erfolgt bis Donnerstag in der jeweils folgenden Woche nach Ausgabe bis 12 Uhr in der IFIS-Teeküche (1 Kasten pro Gruppe)
- Bei Programmieraufgaben: C++ oder Java
- Bitte unbedingt Namen, Matrikelnummern und Übungsgruppennummern auf Abgaben vermerken

Organisatorisches: Prüfung

- Die **Eintragung in den Kurs** und in eine Übungsgruppe ist **Voraussetzung**, um an dem Modul Algorithmen und Datenstrukturen teilnehmen zu können und Zugriff auf die **aktuellen Unterlagen** zu erhalten
- Am Ende des Semesters findet eine **Klausur** statt
- **Voraussetzung** zur **Teilnahme an der Klausur** sind mindestens **50% der gesamtöglichen Punkte aller Übungszettel**

Literatur (dienst als „Skript“!)

Th. Cormen, C.E. Leiserson, R. Rivest, C. Stein,
Algorithmen: Eine Einführung,
4. Auflage, Oldenbourg, 2013

Zusätzlich empfohlene Literatur

U. Schöning,
Algorithmik, Spektrum, 2011

M. Dietzfelbinger, K. Mehlhorn, P. Sanders
Algorithmen und Datenstrukturen - Die Grundwerkzeuge,
Springer, 2014

R. Sedgewick, K. Wayne,
Algorithmen und Datenstrukturen,
4. Auflage, Pearson, 2014

Teilnehmerkreis und Voraussetzungen

Studiengänge

- Bachelor **Informatik**
- Bachelor/Master **Mathematik in Medizin und Lebenswissenschaften**
- Bachelor **Medieninformatik**
- Bachelor **Medizinische Ingenieurwissenschaft**
- Bachelor **Medizinische Informatik**
- Bachelor **IT-Sicherheit**
- Bachelor **Robotik und Autonome Systeme**

Vorausgesetzte Kenntnisse

- Einführung in die Programmierung
- Lineare Algebra und Diskrete Strukturen 1

Ausblick über IFIS Module

- **Bachelor-Programm**
 - Algorithmen und Datenstrukturen
 - Datenbanken
 - Non-Standard-Datenbanken und Data-Mining
 - Cloud- und Web-Technologien
- **Master-Programm**
 - Informationssysteme
 - Datenmanagement
 - Mobile und verteilte Datenbanken
 - Semantic Web
 - Intelligente Agenten
 - Web-Mining Agents
 - Projektpraktikum Web Mining Agents
 - Aktuelle Themen Data Science und KI

Spezielle Lernziele in diesem Kurs

- **Weg vom Problem zum Algorithmus** gehen können
 - **Auswahl** eines Algorithmus aus Alternativen unter Bezugnahme auf vorliegende Daten und deren Struktur
 - **Entwicklung** eines Algorithmus mitsamt geeigneter Datenstrukturen (Terminierung, Korrektheit, ...)
- **Analyse von Algorithmen** durchführen
 - Anwachsen der Laufzeit bei Vergrößerung der Eingabe
- Erste Schritte in Bezug auf die **Analyse von Problemen** gehen können
 - Ja, Probleme sind etwas anderes als Algorithmen!
 - Probleme können in gewisser Weise „schwer“ sein
 - Prüfung, ob Algorithmus optimal

Beispielproblem: Summe der Elemente eines Feldes $A[1..n]$ bestimmen

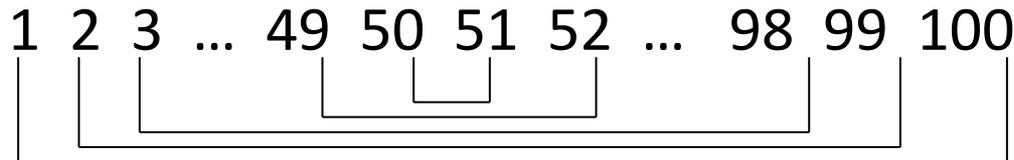
• Algorithmus?

- $\text{summe}(A) = \sum_{i=1}^n A[i]$
- Aufwand?
- Wenn A n Elemente hat, n Schritte!
- Der Aufwand wird *linear* genannt

Spezialisierung des Problems

- Vorwissen: $A[i] = i$
- Das Problem wird sehr viel einfacher!

Ausnutzen der Einschränkung



Summe jedes Paares: 101

50 Paare: $101 * 50 = 5050$

- Lösungsverfahren:
function summe-2(A)
 $n \leftarrow \text{length}(A)$
 return $(n+1)*(n/2)$
- Nach Carl Friedrich Gauß (ca. 1786)
- Aufwand?
- Konstant, d.h. hängt (idealisiert!) nicht von n ab
- Entwurfsmuster: Ein-Schritt-Berechnung (konstanter Aufwand)

Algorithmen: Notation durch Programme

- Annahme: Serielle Ausführung
- Vgl. **Vorlesung „Einführung in die Programmierung“**
 - Variablen, Felder A[...]
 - Zuweisungen \leftarrow (oder auch $:=$)
 - Fallunterscheidungen **if ... then ... else ...**
 - Vergleich und Berechnungen für Bedingungstest
 - Schleifen **while ... do, for ... do**
 - Vergleich und Berechnungen für Bedingungstest
 - **procedure, function**
 - Auf Folien wird der jeweilige Skopus durch Einrückung ausgedrückt

Ein erstes Problem: Summe der Elemente

- **Gegeben: $A[1..n] : \mathbb{N}$**
 - Feld (Array) A von n Zahlen aus \mathbb{N} (natürliche Zahlen)
- **Gesucht:**
 - Transformation S von A , so dass gilt:
 - $\text{summe} = \sum_{i \in \{1, \dots, n\}} A[i]$
 - Also: Gesucht ist ein Verfahren S , so dass $\{P\} S \{Q\}$ gilt (Notation nach [Hoare](#))
 - Vorbedingung: P : true (keine Einschränkung)
 - Nachbedingung: Q : $\text{summe} = \sum_{i \in \{1, \dots, n\}} A[i]$

Ein zweites Problem: Summe der Elemente

- **Gegeben: $A[1..n] : \mathbb{N}$**
 - Feld (Array) A von n Zahlen aus \mathbb{N} (natürliche Zahlen)
- **Gesucht:**
 - Transformation S von A , so dass gilt:
 - $\text{summe-2} = \sum_{i \in \{1, \dots, n\}} A[i]$
 - Also: Gesucht ist ein Verfahren S , so dass $\{P\} S \{Q\}$ gilt (Notation nach Hoare)
 - Vorbedingung: $P: A[i] = i$
 - Nachbedingung: $Q: \text{summe-2} = \sum_{i \in \{1, \dots, n\}} A[i]$

Problem: In-situ-Sortierproblem

- **Gegeben: $A[1..n] : \mathbb{N}$**
 - Feld (Array) A von n Zahlen aus \mathbb{N} (natürliche Zahlen)
- **Gesucht:**
 - Transformation S von A , so dass gilt: $\forall 1 \leq i < j \leq n: A[i] \leq A[j]$
 - Nebenbedingung: Es wird intern kein weiteres Feld gleicher (oder auch nur fast gleicher Größe) verwendet
 - Also: Gesucht ist ein Verfahren S , so dass $\{ P \} S \{ Q \}$ gilt (Notation nach [Hoare](#))
 - Vorbedingung: $P = \text{true}$ (keine Einschränkung)
 - Nachbedingung: $Q = \forall 1 \leq i < j \leq n: A[i] \leq A[j]$
 - Nebenbedingung: nur „konstant“ viel zusätzlicher Speicher (feste Anzahl von Hilfsvariablen)

In-situ-Sortieren: Problemanalyse

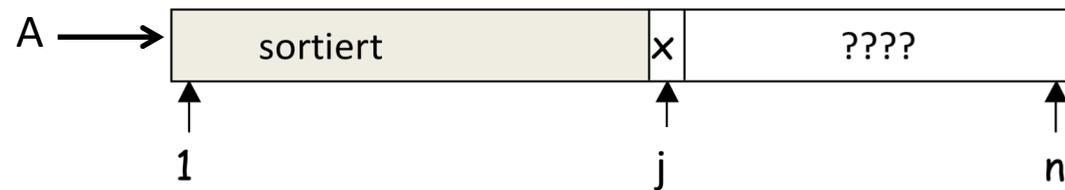
- Felder erlauben **wahlfreien** Zugriff auf Elemente
 - **Zugriffszeit** für ein Feld **konstant**
(d.h. sie hängt nicht vom Indexwert ab)
 - Idealisierende Annahme (gilt nicht für moderne Computer)
- Es gibt keine Aussage darüber, ob die Feldinhalte schon sortiert sind, eine willkürliche Reihenfolge haben, oder umgekehrt sortiert sind
 - Vielleicht lassen sich solche „**erwarteten Eingaben**“ aber in der Praxis feststellen
- **Aufwand das Problem zu lösen:** Man kann leicht sehen, dass jedes Element „falsch positioniert“ sein kann
 - **Mindestaufwand** im allgemeinen Fall: n Bewegungen
 - **Maximalaufwand** in Abhängigkeit von n ?

Aufwand zur Lösung eines Problems

- Gegeben ein **Problem** (hier: In-situ-Sortierproblem)
 - Damit verbundene Fragen:
 - Wie „langsam“ muss ein Algorithmus sein, damit alle möglichen Probleminstanzen korrekt gelöst werden?
 - Oder: Wenn wir schon einen Algorithmus haben, können wir noch einen „substantiell besseren“ finden?
- Jedes Eingabefeld A stellt eine **Probleminstanz** dar
- Notwendiger Aufwand in Abhängigkeit von der Eingabegröße heißt **Komplexität eines Problems**
 - Anzahl der notwendigen Verarbeitungsschritte in Abhängigkeit der Größe der Eingabe (hier: Anzahl der Elemente des Feldes A)
 - Komplexität durch jeweils „schlimmste“ Probleminstanz bestimmt
 - Einzelne Probleminstanzen können evtl. weniger Schritte benötigen

Beispiel 2: Sortierung

- Gegeben: $A = [4, 7, 3, 5, 9, 1]$
- Gesucht: In-situ-Sortierverfahren (aufsteigend)
- Aufgabe: Entwickle "Idee"



```
1: procedure INSERTION-SORT( $A$ )
2:   for  $j \leftarrow 2$  to  $length(A)$  do
3:      $key \leftarrow A[j]$ 
4:      $\triangleright$  Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ 
5:      $i \leftarrow j-1$ 
6:     while  $i > 0$  and  $A[i] > key$  do
7:        $A[i+1] \leftarrow A[i]$ 
8:        $i \leftarrow i-1$ 
9:      $A[i+1] \leftarrow key$ 
```

Entwurfsmuster / Entwurfsverfahren

- Schrittweise Berechnung
 - Beispiel: Bestimmung der Summe eines Feldes durch Aufsummierung von Feldelementen
- Ein-Schritt-Berechnung
 - Beispiel: Bestimmung der Summe eines Feldes ohne die Feldelemente selbst zu betrachten (geht nur unter Annahmen)
- Verkleinerungsprinzip
 - Beispiel: Sortierung eines Feldes
 - Unsortierter Teil wird immer kleiner, letztlich leer
 - Umgekehrt: Sortierter Teil wird immer größer, umfasst am Ende alles → Sortierung erreicht

Laufzeitanalyse

- Laufzeit als **Funktion der Eingabegröße**
- Verbrauch an Ressourcen: **Zeit, Speicher**, Bandbreite, Prozessoranzahl, ...
- Laufzeit bezogen auf serielle Maschinen mit wahlfreiem Speicherzugriff
 - **von Neumann-Architektur ...**
 - ... und Speicherzugriffszeit als konstant angenommen
- Laufzeit kann von der Art der Eingabe abhängen (**bester Fall, typischer Fall, schlechtester Fall**)
- Meistens: schlechtester Fall betrachtet

Aufwand für Zuweisung, Berechnung, Vergleich?

1: procedure INSERTION-SORT(A)	Zeit	Wie oft?
2: for $j \leftarrow 2$ to $length(A)$ do	c_1	n
3: $key \leftarrow A[j]$	c_2	$n - 1$
4: ▷ Insert $A[j]$ into $A[1..j - 1]$		
5: $i \leftarrow j - 1$	c_4	$n - 1$
6: while $i > 0$ and $A[i] > key$ do	c_5	$\sum_{j=2}^n t_j$
7: $A[i + 1] \leftarrow A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
8: $i \leftarrow i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
9: $A[i + 1] \leftarrow key$	c_8	$n - 1$

$t_j =$ Anzahl der Durchläufe der *while*-Schleife im j -ten Durchgang.

$c_1 - c_8 =$ un spezifizierte Konstanten.

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

Bester Fall: Feld ist aufsteigend sortiert

Ist das Array bereits aufsteigend sortiert, so wird die *while*-Schleife jeweils nur einmal durchlaufen: $t_j = 1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Also ist $T(n)$ eine **lineare Funktion** der Eingabegröße n .

Schlechtester Fall: Feld ist absteigend sortiert

Ist das Array bereits absteigend sortiert, so wird die *while*-Schleife maximal oft durchlaufen: $t_j = j$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + (c_6 + c_7) \frac{n(n-1)}{2}$$

Also ist $T(n)$ eine **quadratische Funktion** der Eingabegröße n .

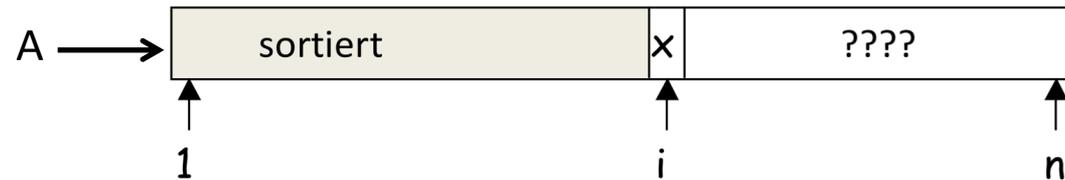
Schlimmster vs. typischer Fall

Meistens geht man bei der Analyse von Algorithmen vom (*worst case*) aus.

- *worst case* Analyse liefert **obere Schranken**
- In vielen Fällen ist der *worst case* die Regel
- Der aussagekräftigere (gewichtete) Mittelwert der Laufzeit über alle Eingaben einer festen Länge (*average case*) ist oft bis auf eine multiplikative Konstante nicht besser als der *worst case*.
- Belastbare Annahmen über die mittlere Verteilung von Eingaben sind oft nicht verfügbar.

Eine andere "Idee" für Sortieren

- Gegeben: $a = [4, 7, 3, 5, 9, 1]$
- Gesucht: In-situ-Sortierverfahren



```
1: procedure SELECTION-SORT( $A$ )
2:   for  $i \leftarrow 1$  to  $length(A)$  do
3:      $min \leftarrow i$ 
4:     for  $j \leftarrow i + 1$  to  $length(A)$  do
5:        $\triangleright$  find the minimum in the unsorted part
6:       if  $A[j] < A[min]$  then
7:          $min \leftarrow j$ 
8:        $x \leftarrow A[i]; A[i] \leftarrow A[min]; A[min] \leftarrow x$ 
9:        $\triangleright$  swap the found minimum into the sorted part
```

Zusammenfassung: Entwurfsmuster

- In dieser Vorlesungseinheit:
 - Schrittweise Berechnung
 - Ein-Schritt-Berechnung
 - Verkleinerungsprinzip
- Nächste Vorlesung
 - Teile und Herrsche
- „Später“:
 - Vollständige Suchverfahren
(z.B. Rücksetzen, Verzweigen und Begrenzen (Branch and Bound))
 - Approximative Such- und Berechnungsverfahren
(z.B. gierige Suche)
 - Schrittweise Annäherung
 - Dynamisches Programmieren (Berechnung von Teilen und deren Kombination, Wiederverwendung von Zwischenergebnissen)