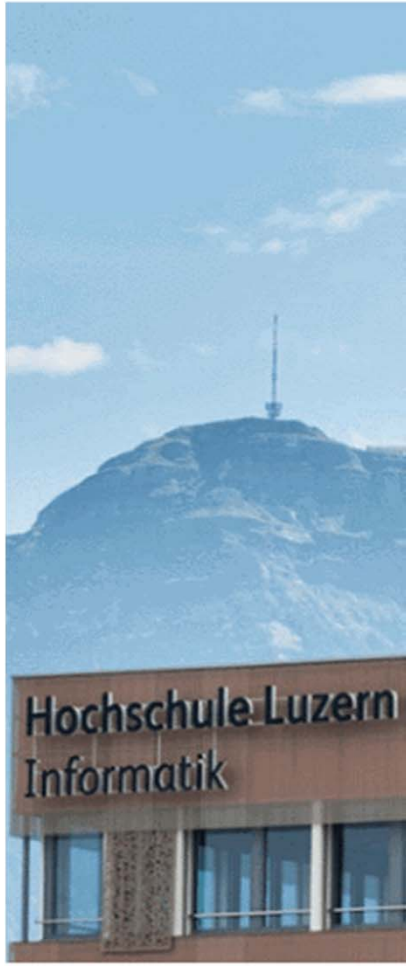
Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

INFORMATIK



Algorithmen & Datenstrukturen

Sortieren – Grundlagen

Hansjörg Diethelm

Inhalt

- Motivation fürs Sortieren
- Voraussetzungen fürs Sortieren
- Kategorie «vergleichsbasierte Sortieralgorithmen»
- Kategorie «Radix-Sortieralgorithmen»
- Aspekte beim Sortieren

Lernziele

Sie …

- können die Motivation fürs Sortieren darlegen.
- kennen die Voraussetzungen und die damit verbundenen Java-Interfaces, damit man Datenelemente sortieren kann.
- wissen, welche Zeitkomplexität vergleichsbasierte und Radix-Sortierverfahren bestenfalls besitzen.
- können einen einfachen Entscheidungsbaum aufzeichnen.
- können den Unterschied zwischen internem und externem Sortieren erklären.
- können anschaulich aufzeigen, was ein stabiler Sortieralgorithmus garantiert.
- wissen, wie man die Zeitkomplexität häufig praktisch differenziert.

Motivation fürs Sortieren

Ordnung bringt's

Es liegt **Unordnung** vor (unsortiert):

- Späteres Suchen bzw. späterer → **Zugriff** ist mühsam.
- Lineares Suchen hat einen Aufwand von **O(n)**.
  - z.B. n = 10'000'000 → Rechenzeit $\sim$ 10'000'000

Es liegt **Ordnung** vor (sortiert):

- Späteres Suchen bzw. späterer Zugriff ist einfacher und schneller.
- Binäres Suchen («sukzessives Halbieren») hat einen Aufwand von **O(ld n)** bzw. **O($\log_2$ n)**.
  - z.B. n = 10'000'000 → Rechenzeit $\sim$ 23

  Also hier rund 500'000 mal schnellerer Zugriff!

Zugriff auf Daten

- Effizienter Zugriff auf Daten ist essentiell in der Informatik.
- Beispiele:
  - weit entferntestes Grafik-Objekt
  - Person mit bestimmter ID
  - Prozess mit höchster Priorität
  - Zugsverbindung mit Abfahrtszeit früher als x
  - bester Student
  - Manager mit Salär kleiner als x
  - Website mit mehr als x Hits
  - Taxi, das am nächsten ist
  - ...
- «**Big Data** lässt grüssen!»

Voraussetzungen fürs Sortieren

Ordnung

- Menge von Datenelementen.
- Unter den Datenelementen existiert eine → **totale Ordnung** bzw. eine lineare Ordnung, d.h. für zwei beliebige Datenelemente x und y gilt:
  - x ist **kleiner** y ODER
  - x und y sind **gleich** ODER
  - x ist **grösser** y
- Damit ist für die Datenelemente eine **sortierte Folge** festgelegt.
- Datenelemente müssen sich also entsprechend vergleichen lassen.

Schlüssel bzw. Key

- Zu sortierende Datenelemente müssen also **vergleichbar** sein.
- Typisch werden Datenelemente anhand ihrer **Schlüssel** bzw.
  ➔ **Keys** verglichen:
  - Als Key kann ein **einzelnes Attribut** fungieren, z.B. `personID`.
  - Key kann auch **eine Attribut-Kombination** sein, z.B. `lastName`, `firstName`.
- Entsprechende Java-Klasse implementiert also typisch das
  - Interface **`Comparable<T>`** mit `int compareTo(T o)` für die natürliche Ordnung, und gegebenenfalls auch das
  - Interface **`Comparator<T>`** mit `int compare(T o1, T o2)` für jede spezielle Ordnung.
  - Siehe Modul OOP, Input: O09_IP_ObjectEqualsCompare

Kategorie
«vergleichsbasierte Sortieralgorithmen»

Komplexität bei vergleichsbasierten Sortieralgorithmen

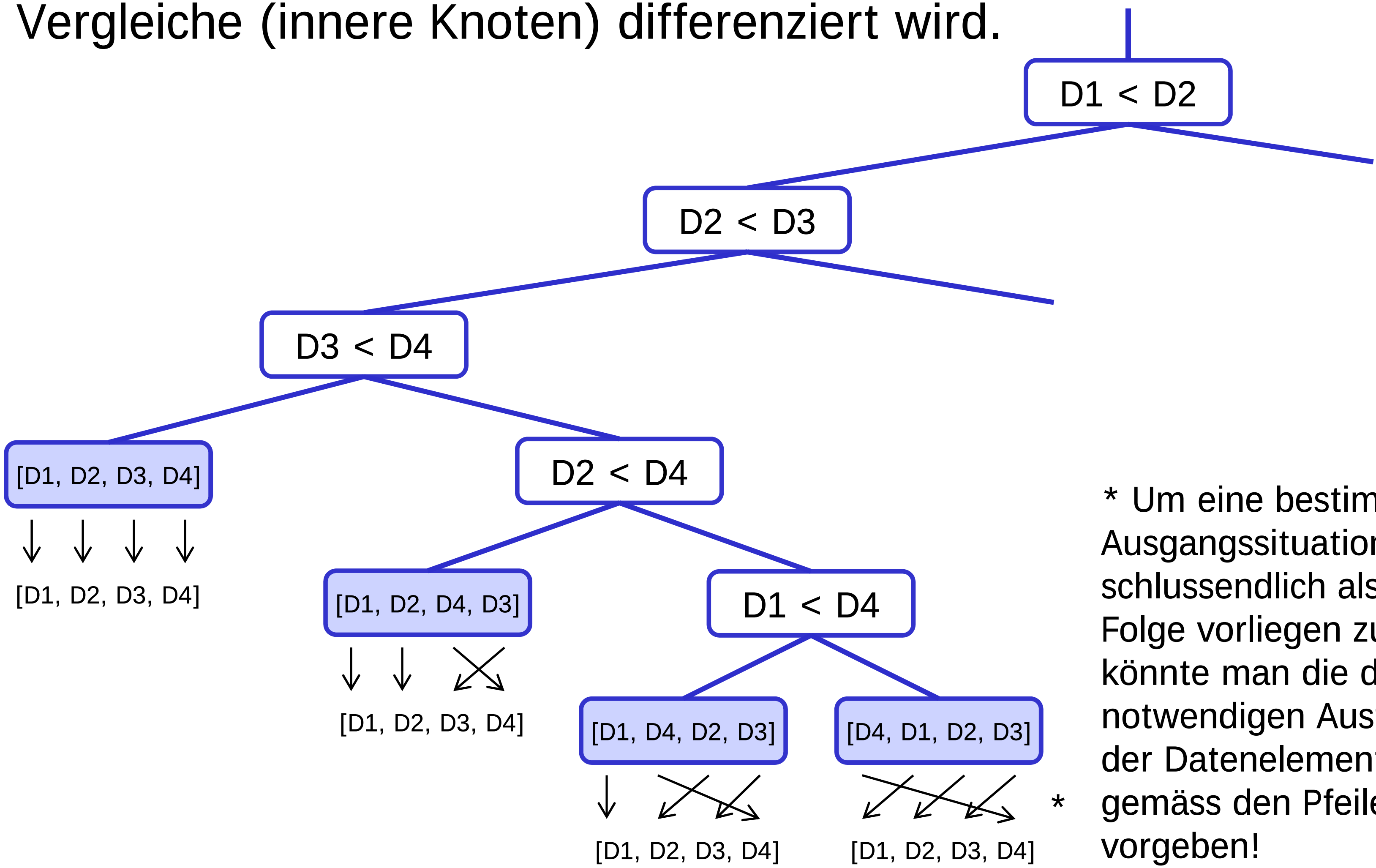
- Bei Sortieralgorithmen interessiert vor allem deren **Zeitkomplexität** O(g(n)) = ?
  D.h., wie verhält sich die Rechenzeit für das Sortieren von grossen Datenmengen bei einem bestimmten Algorithmus schlimmstenfalls, wenn sich die Datenmenge z.B. verdoppelt?
- Wir zeigen nachfolgend, dass im Falle von sogenannten ➔ **vergleichsbasierten Sortieralgorithmen** (Regelfall, massgebend für die Rechenzeit sind Vergleiche) das Sortierproblem **bestenfalls** mit der Zeitkomplexität **O(n · log n)** lösbar ist.
- Die Ausgangsfrage lautet somit: Welche Zeitkomplexität steckt inhärent im Problem des vergleichsbasierten Sortierens?

Beispielhaftes Setting

- Z.B. seien $n = 4$ Datenelemente D zu sortieren.
- Auf den Datenelementen sei eine totale Ordnung definiert, d.h. die Datenelemente lassen sich mit < == > vergleichen und damit entsprechend ordnen bzw. sortieren.
- Der Einfach- und Verständlichkeit halber entspreche D1 dem kleinsten und D4 dem grössten Datenelement.
- [D1,D2,D3,D4] wäre dann die sortierte Folge.
- Mit $n = 4$ sind $n! = 24$ verschiedene Ausgangssituationen (vgl. ➔ Permutationen) für das Sortieren denkbar, also [D1,D2,D3,D4] [D1,D2,D4,D3] [D1,D4,D2,D3] [D4,D3,D2,D1] …

Entscheidungsbaum

Jedes Blatt im Entscheidungsbaum repräsentiert genau eine der n! möglichen Ausgangssituationen, welche durch die vorangehenden Vergleiche (innere Knoten) differenziert wird.



* Um eine bestimmte Ausgangssituation schlussendlich als sortierte Folge vorliegen zu haben, könnte man die dafür notwendigen Austausche der Datenelemente gemäss den Pfeilen fest vorgeben!

Schlussfolgerungen

- Mit Hilfe eines **binären** Entscheidungsbaumes kann man also alle n! Ausgangssituation differenzieren. Für jede mögliche Ausgangssituation steht ein Blatt.
- Mit jedem inneren Knoten ist ein Vergleich verbunden.
- Damit man alle möglichen Ausgangssituationen differenzieren bzw. sortieren kann, sind demnach entsprechend der Baumhöhe h mindestens (h-1) Vergleiche C (Compares) notwendig:

$$C \geq (h-1)$$

- Bei einem Binärbaum mit B Blättern gilt:

$$h \geq \log_2(B)+1$$

- Weil hier B = n! gilt resultiert:

  ➔ Es sind mindestens $\mathbf{C \geq \log_2(n!)}$ Vergleiche notwendig.

Abschätzung von $\log_2(n!)$

- $\log_2(n!) = \log_2(1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n/2 \cdot \ldots \cdot (n-1) \cdot n)$
- $\log_2(n!) > \log_2(n/2 \cdot \ldots \cdot (n-1) \cdot n)$ D.h. erste Faktoren einfach weggelassen!
- $\log_2(n!) >> \log_2(n/2)^{n/2}$ D.h. alle Faktoren nur auf n/2 gesetzt!

$$= n/2 \cdot \log_2(n/2) = \mathbf{n}/2 \cdot (\mathbf{\log_2 n} - 1)$$

- Damit resultiert $\mathbf{\log_2(n!)} \in \mathbf{\Omega(n \cdot \log n)}$, was bedeutet, dass $\log_2(n!)$ **mindestens** so schnell wächst wie $(n \cdot \log n)$ bzw. das
  ➔ **Sortierproblem bestenfalls mit $O(n \cdot \log n)$ lösbar ist**.

- Beachte:
  $f(n) \in \mathbf{O}(g(n))$, d.h. $f(n)$ wächst **höchstens** so schnell wie $g(n)$.
  $f(n) \in \Omega(g(n))$, d.h. $f(n)$ wächst **mindestens** so schnell wie $g(n)$.

Beispiele (Auszug)

- **Einfache Sortieralgorithmen ➔ $O(n^2)$**
  - direktes Einfügen (Insertion Sort)
  - direktes Auswählen (Selection Sort)
  - direktes Austauschen (Bubble Sort)
- **Höhere Sortieralgorithmen ➔ $O(n \cdot \log n)$**
  - Quicksort
  - Heapsort
  - Mergesort
- **Sortiernetzwerke mit $O(n)$ Prozessoren ➔ $O(\log n)$**
  - parallelisierter Mergesort

Kategorie
«Radix-Sortieralgorithmen»

Radix-Sortieralgorithmen

- Radix-Sortieralgorithmen bedingen **spezielle Anforderungen an die Schlüssel** und finden entsprechend selten Verwendung. Ein eigentliches Vergleichen wird damit hinfällig!
- Beispiel: n = 4 Datenelemente D1, D2, D3, D4.
  Als Schlüssel komme genau jeder int-Werte **1 ... n** ein Mal vor.
  z.B. [D1/**2**, D2/**4**, D3/**1**, D4/**3**]
  Die Schlüsselwerte geben hier direkt die Reihenfolge für die Sortierung an: [D3/**1**, D1/**2**, D4,/**3** D2/**4**]
- Damit ist bereites gezeigt, dass ➔ **Radix-Sortieralgorithmen** das Sortierproblem **bestenfalls** mit der Zeitkomplexität **O(n)** lösen können!
- **Beispiele**: Counting-Sort, Radix-Sort, Bucket-Sort

Aspekte beim Sortieren

Stabiler vs. instabiler Sortieralgorithmus

- Ausgangssituation vor dem Sortieren:
  - Mehrere Datenelemente haben den gleichen Schlüssel bzw. sind gleich, z.B. [D1/**22**, D2/**7**, D3/**22**, D4/**5**, D5/**71**, D6/**10**].
  - D1 und D3 haben den gleichen Schlüssel **22** bzw. sind gleich.
- Stabiler Sortieralgorithmus:
  - Der Algorithmus **garantiert**, dass durch das Sortieren die **Reihenfolge unter gleichen Datenelementen nicht ändert**, d.h. [D4/**5**, D2/**7**, D6/**10**, D1/**22**, D3/**22**, D5/**71**].
  - Das Datenelement D1 steht nach wie vor links und D3 rechts.
- Instabiler Sortieralgorithmus:
  - Hier kann passieren, dass nach dem Sortieren die Reihenfolge unter gleichen Datenelementen nicht mehr dieselbe ist wie vorher, d.h. [D4/**5**, D2/**7**, D6/**10**, D3/**22**, D1/**22**, D5/**71**]!

Beispiel stabiles vs. instabiles Sortieren

- **Namensverzeichnis**
  1. Zuerst nach Vornamen sortieren.
  2. Dann nach Namen sortieren.

Giacometti, Giovanni
Giacometti, Alberto
Veronese, Paolo
Böcklin, Arnold
Vallotton, Félix

Giacometti, Alberto
Böcklin, Arnold
Vallotton, Félix
Giacometti, Giovanni
Veronese, Paolo

stabil

Böcklin, Arnold
Giacometti, Alberto
Giacometti, Giovanni
Vallotton, Félix
Veronese, Paolo

instabil

Böcklin, Arnold
Giacometti, Giovanni
Giacometti, Alberto
Vallotton, Félix
Veronese, Paolo

Internes vs. externes Sortieren

- Internes Sortieren:
  - Daten liegen im **Arbeitsspeicher** vor.
  - **Direktes Vergleichen** der Daten möglich.
  - Internes Sortieren ist primär von Bedeutung.
  - Beispiel: Sortieren von Arrays

- Externes Sortieren:
  - Daten liegen in einem **externen Massenspeicher** vor.
  - **Nur Lesen und Schreiben** möglich, kein direktes Vergleichen!
  - Interner Speicher ist für alle Daten zu klein!
  - Beispiel: Sortieren von sequentiellen Dateien

Zeitkomplexität

- Von Bedeutung sind vor allem **interne**, **vergleichsbasierte** Sortieralgorithmen.
- Betreffend deren Zeitkomplexität O(g(n)) sind primär die Anzahl erforderlicher **Vergleichsoperationen massgebend**.
- Die Rechenzeit hängt beim Sortieren natürlich von der **Anzahl n** zu sortierender Datenelemente ab. Aber nicht nur – **auch die Werte** können relevant sein! Falls die zu sortierenden Werte z.B. bereits mehrheitlich aufsteigend vorliegen, arbeitet ein Algorithmus vielleicht wesentlich schneller.
- Bei der Zeitkomplexität unterscheidet man deshalb häufig:
  - **Average Case** (Mittel über alle Permutationen)
  - **Worst Case**
  - **Best Case** (nicht besonders wichtig)

Zusammenfassung

- Der Zugriff auf geordnete bzw. sortierte Daten ist effizienter.
- Eine Ordnung bzw. Sortierung ist dann möglich, wenn auf den Datenelementen eine «totale Ordnung» existiert, d.h. es lassen sich zwei beliebige Datenelemente mit `<` `==` `>` vergleichen.
- Wichtige Java-Interfaces sind `Comparable` und `Comparator`.
- Vergleichsbasiertes Sortieren ist inhärent bestenfalls mit einer Zeitkomplexität von $O(n \cdot \log n)$ möglich. Radix basiertes Sortieren ist bestenfalls mit $O(n)$ möglich.
- Bei stabilen Sortieralgorithmen behalten gleiche Datenelemente ihre Reihenfolge untereinander bei.
- Bei internem Sortieren liegen alle Daten im Arbeitsspeicher vor.
- Man differenziert häufig: Average Case, Worst Case, Best Case.

Fragen?