Technische Hochschule Rosenheim
Technical University of Applied Sciences

Objektorientierte Programmierung
Kapitel 7 – Sortieren

Prof. Dr. Kai Höfig

Sortieren

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Sortieren ist ein Vorgang, bei dem durch **Vergleichen** und etwaigem **Tauschen** von Elementen eine Ordnung herstellt wird, bei der das *kleinste* Element an erster Stelle steht. Wir wollen uns heute ein paar solcher Sortierverfahren erarbeiten.

    1. **Vergleichen und Tauschen**
    2. **Sortieren durch Auswählen**
    3. **Sortieren durch Einfügen**
    4. **Sortieren durch Teilen und Herrschen**
    5. **Quicksort**

- Auf der Webseite https://visualgo.net/de/sorting werden die verschiedenen Verfahren visualisiert.

Vergleichen und Tauschen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Um zwei primitive Datentypen (`short`, `int`, `float`, etc.) zu vergleichen, können wir in Java die Vergleichsoperatoren `<`, `>` und `==` verwenden. Um zwei Objekte zu vergleichen kennen wir in Java die Interfaces `Comparable<T>` und `Comparator<T>`, mit ihren Methoden `compareTo(T other)` und `compare(T t1, T t2)`. Der Übersicht halber werden wir uns zunächst auf `int` und den `<`-Operator beschränken. Am Ende dieses Kapitels verallgemeinern wir die Algorithmen, um eigene Sortierordnungen zu definieren.

- Interessanter wird es beim Tausch; hier muss sich einer Hilfsvariablen bedient werden:

```
class Sortieren {
    static void swap(int[] a, int i, int j) {
        int hilf = a[i];
        a[i] = a[j];
        a[j] = hilf;
    }
}
```

- Zusammen mit den Syntaxelementen für bedingte (`if-else`) und wiederholte (`for, while`) Ausführung sind wir nun mit den Grundwerkzeugen zum sortieren vertraut.

Sortieren durch Auswählen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Eines der wohl einfachsten wie anschaulichsten Sortierverfahren ist das Sortieren durch Auswählen (**selection sort**).
  - Dabei betrachtet man das zu sortierende Array als zwei Teilarrays: ein sortierter Teil **S** (links) sowie ein unsortierter Teil **U** (rechts).
  - Zu Beginn ist der sortierte Teil leer, er endet also vor dem ersten Element.
  - Man sucht nun der Reihe nach immer das nächstkleinste Element im unsortierten Teil, und tauscht dieses mit dem ersten Element des unsortierten Teil. Dadurch wird der sortierte Teil nun um eins größer.

```
a = [|3 2 4 1 ]  // min = 1; tausche mit 3
    S|U

    [ 1|2 4 3 ]  // min = 2; korrekt positioniert
      S|U

    [ 1 2|4 3 ]  // min = 3; tausche mit 4
        S|U

    [ 1 2 3|4 ]  // 1-elementiges Array ist sortiert
          S|U

    [ 1 2 3 4|]  // fertig!
            S|U
```

Selection Sort Implementierung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Den Algorithmus (wiederholtes Minimum suchen und tauschen) können wir in Java wie folgt ausdrücken:

```
static void ssort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        // Position des aktuellen Minimums
        int p = i;

        // nach kleinerem Wert suchen, Position merken
        for (int j = i+1; j < a.length; j++)
            if (a[j] < a[p])
                p = j;

        // tauschen, wenn noetig
        if  (i != p)
            swap(a, i, p);
    }
}
```

- Man sieht: Es wird hier immer nur getauscht, zu keiner Zeit wird das Array dupliziert. *Selection Sort* ist also ein sog. **in-place Sortierverfahren**, bei dem, außer für ein paar Hilfsvariablen, kein zusätzlicher Speicher benötigt wird.

- Der Aufwand dieses Sortierverfahrens ist O(n^2), wie man unschwer an den zwei geschachtelten `for`-Schleifen erkennen kann. Da dieses Verfahren in-place ist, eignet es sich besonders für Arrays.

Sortieren durch Einfügen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Verwandt, aber etwas kniffliger zu realisieren ist das Sortieren durch Einfügen (**insertion sort**). Hierbei unterscheidet man ebenso zwischen dem bereits sortierten Teil `S` und dem unsortierten Teil `U`. Man sucht nun aber nicht das Minimum in `U`, sondern nimmt das erste Element `x` aus `U` heraus und fügt es an der richtigen Stelle in `S` ein, indem man alle Elemente `y` > `x` aus `S` um eins nach rechts tauscht:

```
a = [ 3 2 4 1 ]  // 3 einsortieren; bereits am richtigen Ort
    S|U

    [ 3|2 4 1 ]  // 2 einsortieren
      S|U

        2
    [ 3 -|4 1 ]  // Element herausnehmen, S vergrößern
    [ - 3|4 1 ]  // größere nach rechts schieben
    [ 2 3|4 1 ]  // einfügen
       S|U

          4
    [ 2 3 -|1 ]  // Element herausnehmen, S vergrößern
    [ 2 3 4|1 ]  // nichts zu schieben, einfügen
         S|U

            1
    [ 2 3 4 -|] // Element herausnehmen, S vergrößern
    [ 2 3 - 4|] // größere nach rechts schieben
    [ 2 - 3 4|]
    [ - 2 3 4|]
    [ 1 2 3 4|] // einfügen, fertig!
           S|U
```

Insertion Sort Implementierung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Auch hier ist der Aufwand O(n^2), da wiederum zwei geschachtelte Schleifen zu finden sind. Es wird ebenso *in-place* sortiert, man sieht aber auch: es sind je nach Datenlage viele Tauschvorgänge nötig, um das Herausnehmen aus U und das Einfügen in S zu erreichen.

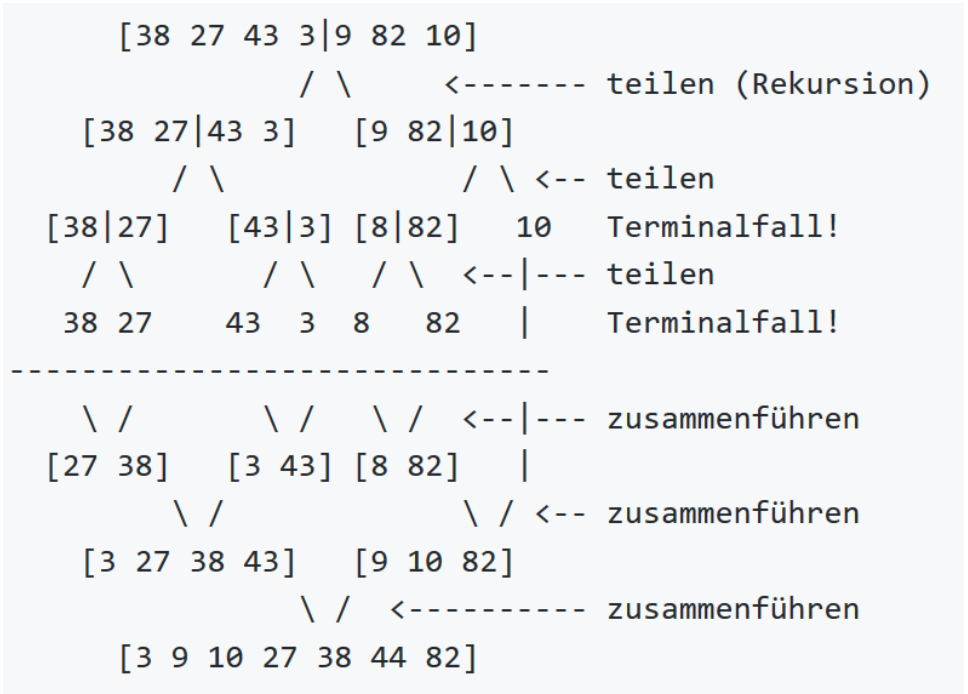
- Da das Einfügen und Entfernen in verketteten Listen einfach und schnell ist, ist dieses Verfahren besonders für verkettete Listen geeignet.

```
static void isort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        // aktuellen Wert herausnehmen
        // (a[i] kann also überschrieben werden)
        int x = a[i];

        // alle Elemente eins nach rechts rücken
        // bis Einfügeposition gefunden
        int j = i-1;
        while (j >= 0 && a[j] > x) {
            swap(a, j, j + 1);
            j--;
        }

        // an der "freien" Stelle einfügen
        a[j+1] = x;
    }
}
```

Sortieren durch Teilen und Herrschen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Heute wenden wir Rekursion auf das Sortierproblem an, um die Probleme einfacher beschreiben zu können.
- Möchte man Sortieren rekursiv beschreiben, so stellt man fest:
  - Ein leeres oder einelementiges Array ist bereits sortiert (Terminalfall).
  - Ein mehr-elementiges Array teilt man in zwei Teile, sortiert diese (Rekursion), und fügt diese dann zusammen.



Merge Sort Implementierung (1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- In Java realisieren wir diese Rekursion wie folgt; das Zusammenführen (`merge`) ist hier der Übersicht halber ausgelagert.

```
static int[] msort(int[] a) {
    // Terminalfall -- bereits sortiert.
    if (a.length < 2)
        return a;

    int p = a.length / 2;

    // Teillisten sortieren
    int[] l = msort(Arrays.copyOfRange(a, 0, p));
    int[] r = msort(Arrays.copyOfRange(a, p, a.length));

    // sortierte Teillisten zusammenfuehren
    return merge(l, r);
}
```

- Dieses Verfahren wird **merge sort** genannt, und hat (analog zum einfügen in einen Binärbaum) die Komplexität O(n log n) -- im Hinblick auf den Rechenaufwand. Der Speicheraufwand kann hier aber je nach Implementierung deutlich höher sein, wenn Kopien der Arrays erstellt werden. Auch hier gilt: Da das Anhängen an bzw. Teilen von Listen sehr effizient implementierbar ist, ist *merge sort* besonders für verkettete Listen geeignet.

Merge Sort Implementierung (2)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Es verbleibt die Frage, wie nun zwei sortierte Arrays zusammengeführt werden können. Hierzu nimmt man abwechselnd das jeweils kleinere Element von beiden Arrays und fügt es in ein neues ein:

```
private static int[] merge(int[] a, int[] b) {
    // neues Array so groß wie a und b zusammen
    int[] res = new int [a.length + b.length];

    // drei Indizes: für res, a und b
    int i = 0, l = 0, r = 0;
    while (l < a.length && r < b.length) {
        if (a[l] < b[r])
            res[i++] = a[l++];
        else
            res[i++] = b[r++];
    }

    // links oder rechts noch was übrig?
    while (l < a.length)
        res[i++] = a[l++];
    while (r < b.length)
        res[i++] = b[r++];

    return res;
}
```

Quicksort

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Quicksort verbindet die Grundidee von merge sort, aber teilt das Array nicht systematisch in zwei Hälften, sondern dadurch, dass in der linken und rechten Hälfte nur Elemente welche kleiner bzw. größer als ein Pivotelement sind. Man kann zeigen, dass Quicksort in vielen Fällen effizienter ist als merge sort, daher wird er in vielen Bibliotheken als Standardimplementierung verwendet.

```
static void qsort(int[] a) {
    qsort(a, 0, a.length);
}

private static void qsort(int[] a, int from, int to) {
    // kurze Arrays bereits sortiert
    if (to - from < 2)
        return;

    int p = partition(a, from, to);
    if (from < p - 1)
        qsort(a, from, p);
    if (p < to)
        qsort(a, p, to);
}
```

Comparable und Comparator

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Die Sortierordnung ist in den obigen Beispielen immer über den <-Operator hergestellt worden. Will man nun Objekte vergleichen, so müssen diese entweder `Comparable`, und damit die Methode `compareTo`, implementieren, oder man verwendet einen `Comparator`, der mit `compare` zwei Objekte vergleichen kann. Zur Wiederholung: beide Methoden haben `int` als Rückgabewert, und zwar

- < 0 (i.d.R. -1), wenn das erste Element kleiner ist als das zweite;
- 0, wenn beide Elemente gleich sind; und
- > 0 (i.d.R. +1), wenn das erste Element größer ist als das zweite.

- Für die Primitivdatentypen implementiert der entsprechende Wrappertyp das Interface `Comparable` sowie eine statische `compare` Methode:

```
System.out.println(Integer.compare(1, 5));  // "-1"
System.out.println(Integer.compare(3, 3));  // "0"

Integer i = 4;
System.out.println(i.compareTo(2));  // "1"
```

Sortieren mittels Comparator

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
static void ssort(Integer[] a, Comparator<Integer> c) {
    for (int i = 0; i < a.length; i++) {
        // Position des aktuellen Minimums
        int p = i;

        // nach kleinerem Wert suchen, Position merken
        for (int j = i+1; j < a.length; j++)
            if (c.compare(a[j], a[p]) < 0)
                p = j;

        // tauschen, wenn noetig
        if  (i != p)
            swap(a, i, p);
    }
}
```

Vormals

```
if (a[j] < a[p])
```

- Die Sortierordnung also allein dadurch definiert ist, ob das Ergebnis von `compareTo` bzw. `compare` negativ (also "kleiner") ist.

Sortierung umdrehen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Entsprechend kann, kann man so eigene Sortierordnungen definieren, z.B. absteigend, also die größte Zahl zuerst. Umgangssprachlich ausgedrückt heißt das nun, dass das Größte für den Sortieralgorithmus sinngemäß das Kleinste sein muss, das Ergebnis von compare im Vorzeichen genau umgekehrt.

```
Integer[] a = {3, 2, 4, 1};

Sortieren.ssort(a, new Comparator<Integer>() {
    public int compare(Integer a, Integer b) {
        return -1 * a.compareTo(b);  // Vorzeichen umdrehen!
    }
});
```

- Alternativ könnte man obiges auch so erreichen:

- `return b.compareTo(a)`, also durch vertauschte Reihenfolge
- `return Integer.compare(b, a)`, ebenso vertauschte Reihenfolge
- `return b - a` was negativ ist, wenn a > b

Ordnungen mit mehreren Kriterien

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Manchmal ist aber ein Sortierkriterium nicht genug; denken Sie z.B. an eine Namensliste: Hier wird zunächst nach Nachname, bei Gleichheit aber weiterhin nach Vorname sortiert. Ein solcher `Comparator` muss entsprechend hierarchisch vergleichen:

```
class PersonComparator implements Comparable<Person> {
    public int compare(Person a, Person b) {
        // Nachname gleich? Dann bitte nach Vorname sortieren.
        if (a.getNachname().equals(b.getNachname()))
            return a.getVorname().compareTo(b.getVorname());
        else
            return a.getNachname().compareTo(b.getNachname());
    }
}
```

Zusammenfassung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Sortieren beruht im Wesentlichen auf Vergleichen und Tauschen.
- Sortieralgorithmen stellen i.A. eine aufsteigende Ordnung her, d.h. das kleinste Element zuerst.
- Mit `Comparable` bzw. `Comparator` kann man diese Ordnungen nun beliebig definieren: ist ein Objekt kleiner als ein anderes, so muss ein Wert <0 zurückgegeben werden.
- Soll eine bestehende Ordnung umgedreht werden, so genügt es das Vorzeichen zu invertieren.
- Sortieren durch Auswählen und Einfügen haben jeweils die Komplexität O(n^2), wobei letzteres besonders für Listen geeignet ist.
- Merge Sort ist zwar mit O(n log n) effizienter, benötigt aber in der einfachen Implementierung den doppelten Speicher.
- Viele Betriebssysteme und Bibliotheken verwenden Quicksort als Standardsortierverfahren.