Sortierverfahren 1

- Sie wissen, warum Sortieren wichtig ist
- Sie können den Aufwand von Sortieralgorithmen bestimmen
- Sie unterscheiden internes und externes Sortieren
- Sie kennen die drei einfachen internen Sortierverfahren
  - Bubble Sort
  - Selection Sort
  - Insertion Sort

Basiert auf Material von:

Kurt Bleisch
Stephan Neuhaus
Karl Rege
Marcela Ruiz
Jürgen Spielberger

Einführung

Motivation

- Sortieren als (eigenständige) Aufgabe:
  - Worte in Wörterbuch
  - Dateien in einem Verzeichnis
  - Buchkatalog in der Bibliothek
  - Theaterprogramm
  - Rangliste
  - Karten in Kartenspiel

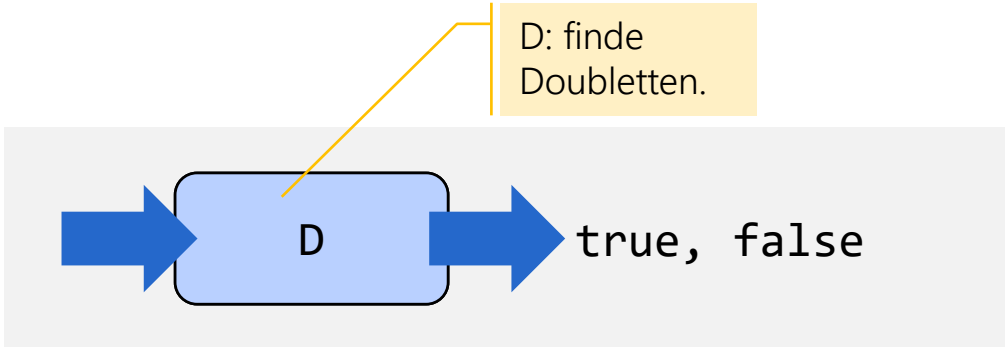
- Sortieren zur Steigerung der Effizienz eines Algorithmus:
  gewisse (schnelle) Algorithmen funktionieren nur, wenn die Daten sortiert sind, wie zum Beispiel Binäre Suche.

Motivation: sind zwei gleiche Karten im Spiel?

Joe hat an diesem Abend viel verloren; er hegt den Verdacht, dass nicht alles mit rechten Dingen zugegangen ist. Sicherheitshalber will er überprüfen, ob keine zusätzliche Karten ins Spiel eingebracht wurden, d.h. keine Karte doppelt vorhanden ist.

Algorithmus 1

Im ersten Algorithmus wird jede Karte mit jeder verglichen:

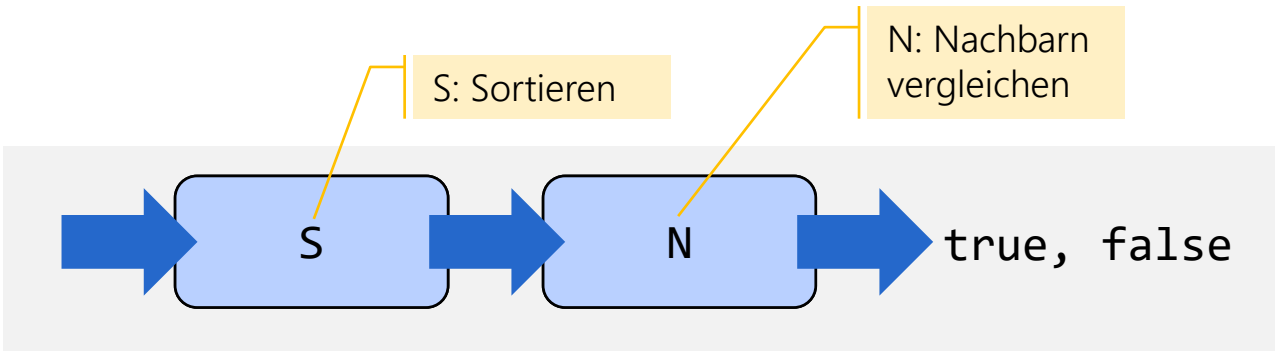


Was ist der Aufwand?

```
Public static boolean duplicates(Objekt[] a){
   for (int i = 0; i < a.length; i++)
      for (int j = i + 1; j < a.length; j++)
         if (a[i].equals(a[j])
            return true;
   return false;
}
```

Algorithmus 2

Ein besserer Algorithmus könnte zuerst die Karten sortieren (natürlich nicht mit einem Algorithmus $O(n^2)$). Dann sind gleiche Karten benachbart; beim nochmaligen Durchgehen durch den Stapel, werden diese dann leicht gefunden.



Aufwand Sortieren? O(n * log(n))

Aufwand Vergleichen? O(n)

Aufwand Total? O(n * log(n))

Joe hatte recht: das Spiel ist manipuliert worden. Beim anschliessenden Duell wurde er dann aber leider erschossen.

Wie sortiere ich einen Kartenstapel?

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

Insertion Sort

Der Spieler nimmt eine Karte nach der anderen auf und sortiert sie in die bereits aufgenommenen Karten ein.

Der Spieler nimmt die jeweils niedrigste der auf dem Tisch verbliebenen Karten auf und kann sie in der Hand links (oder rechts) an die bereits aufgenommenen Karten anfügen.

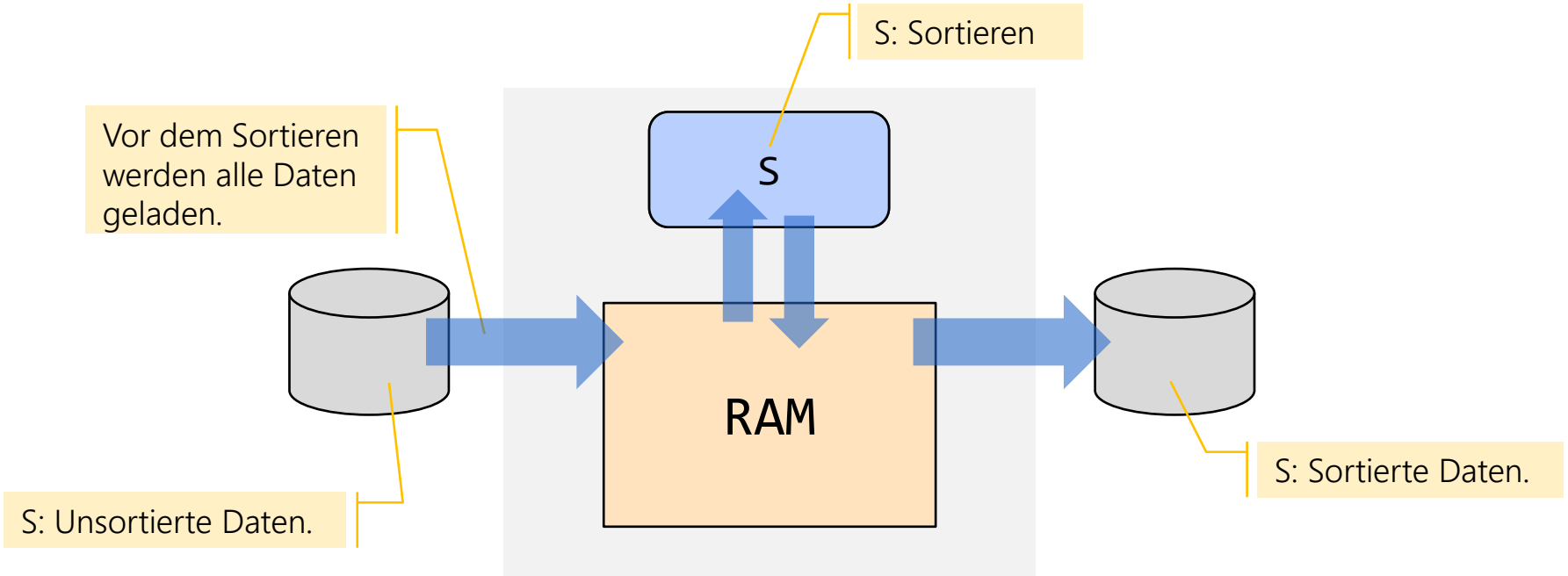
Selection Sort

Der Spieler nimmt alle Karten auf, macht einen Fächer daraus und fängt jetzt an, die Hand zu sortieren, indem er benachbarte Karten solange vertauscht, bis alle in der richtigen Reihenfolge liegen.

Bubble Sort

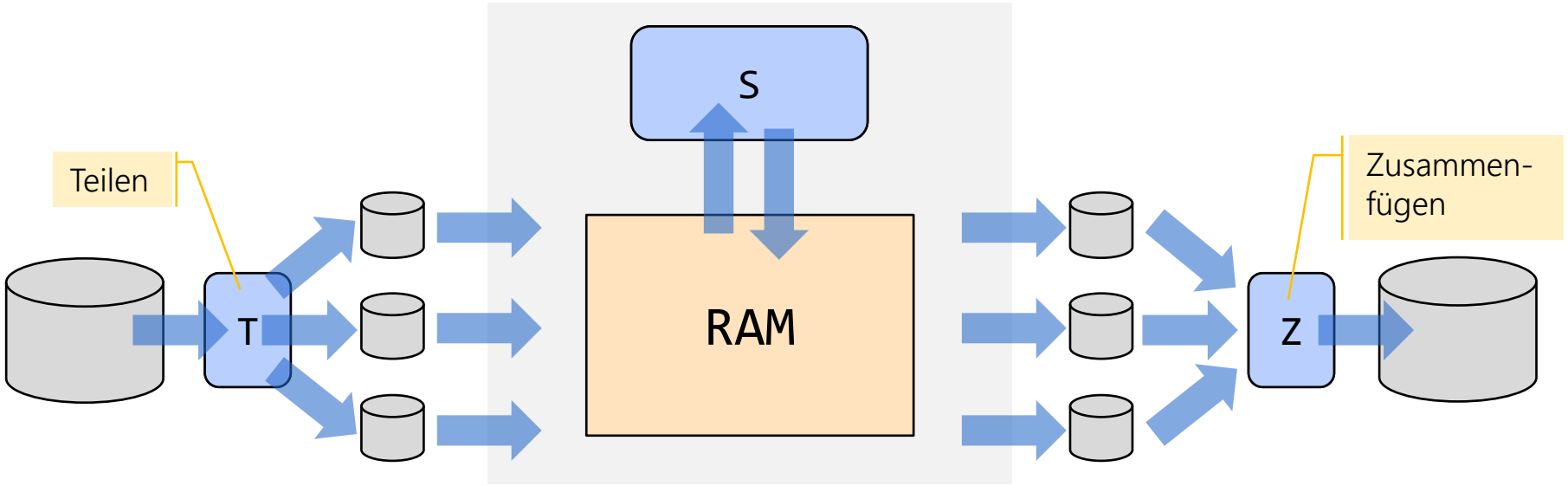
Internes Sortieren

- Wenn die Anzahl der Datensätze und deren jeweiliger Umfang sich in Grenzen halten, kann man alle Datensätze im Arbeitsspeicher eines Computers sortieren.
- Man spricht dann von einem internen Sortiervorgang (Engl. internal sort):



Externes Sortieren

- Können nicht alle Datensätze gleichzeitig im Arbeitsspeicher gehalten werden, dann muss ein anderer Sortieralgorithmus gefunden werden.
- Man spricht dann von einem externen Sortiervorgang (Engl. external sort).



Externes Sortieren: Algorithmus

Algorithmus-Skizze (später in Lektion 12 mehr):

- Teilen die grosse zu sortierende Datei n Teile (klein genug, dass sie in den Hauptspeicher passen).
- Dateien werden nacheinander in Speicher (parallel?) eingelesen, intern sortiert und wieder in Dateien geschrieben.
- Die sortierten Dateien werden schliesslich zu einer sortierten Datei zusammengefügt (merge).

Das Problem des externen Sortierens lässt sich auf das des internen Sortierens zurückführen bzw. setzt voraus, dass ein Teil der Daten intern sortiert werden kann.

Sortierschlüssel

Sortierschlüssel: Sortieren von Datensätzen

Gegeben sei eine Menge von Datensätzen der Form:

- Der Sortierschlüssel ist ein Teil des Inhaltes.
- Der Sortierschlüssel kann aus einem oder mehreren Teilfeldern bestehen, für die eine sinnvolle Ordnung gegeben ist.
- Bei Textfeldern kann dies eine lexikographische Anordnung sein – bei Zahlen eine Anordnung entsprechend ihrer Grösse.
- Der übrige Inhalt der Datensätze ist beliebig und wird nicht weiter betrachtet.

Sortierschlüssel: Definition

- Def: Sortierschlüssel sind Kriterien, nach denen Datensätze sortiert oder gesucht werden können. Beispiel:

```
class Student {
   String  name;
   String  vorname;
   int     matrikelNr;
   short   alter;
   short   studiengang;
   String  fachbereich;
}
```

Datensätze mit dieser Struktur können nach beliebigen Feldern oder Felderkombinationen sortiert werden, so etwa nach:

1. MatrikelNr
2. Name, Vorname, Alter
3. Alter
4. Studiengang
5. Studiengang, Name, Vorname

- Die Sortierung nach [Alter] führt dazu, dass viele Datensätze den gleichen Sortierschlüssel haben.
- Die Sortierung mit dem Sortierschlüssel [Name, Vorname, Alter] führt dazu, dass wenige/keine Datensätze den gleichem Sortierschlüssel haben.
- Die Sortierung nach [MatrikelNr] führt zu einer eindeutigen Sortierung, das heisst, es gibt zu jeder Matrikelnummer höchstens einen Datensatz. In diesem Fall sprechen wir von einem eindeutigen Sortierschlüssel.

Sortierschlüssel: Eigenschaften

- Für Sortierung ist kein eindeutiger Sortierschlüssel notwendig, doch ist er nur sinnvoll, wenn er den Datensatz **weitgehend bestimmt**.
- Dies wären beim obigen Beispiel z.B. die Schlüssel
  - [Name, Vorname] oder
  - [Name, Vorname, Alter] oder
  - [Name, Vorname, MatrikelNr] (eindeutig)

- Beim Anlegen einer **relationalen Datenbank** sollte dagegen in jeder Tabelle ein **eindeutiger Schlüssel** vorhanden sein.

Sortierschlüssel: Vergleich

Um Datensätze sortieren zu können, müssen wir die Schlüsselwerte entsprechend der gewählten Ordnungsrelation vergleichen können.

- Zahlen:

  Im Falle des Schlüssels [MatrikelNr] können wir zwei Studenten S1 und S2 direkt vergleichen:
  `S1.MatrikelNr <= S2.MatrikelNr`
- Strings:

  Im Falle von String-Schlüsseln, z.B. Namen, benötigen wir den Methodenaufruf:
  s1.compareTo(S2)
- Kombinierte Schlüssel:

  Interface Comparable, resp. Comparator implementieren.

Implementation der Klasse Student

```
class Student implements Comparable<Student> {
   private String name;
   private String firstName;
   private int matrikelNr;

   int compareTo(Student s2) {
      int i = name.compareTo(s2.name);
      i = (i != 0)?i:firstName.compareTo(s2.firstName);
      i = (i != 0)?i:matrikelNr - s2.matrikelNr;
      return i;
      }
   }

   ...
   if (s1.compareTo(s2) < 0)
      System.out.println("s1 kommt vor s2");
```

compareTo() für Name, Vorname und Matrikel-Nr.

Implementation der Klasse Student

```
class Student {
   String name;
   String firstName;
   int matrikelNr;
}
```

Comparator für Name, Vorname und Matrikel-Nr.

```
class MyComparator implements Comparator<Student> {
   int compare(Student s1, Student s2) {
      int i = s1.name.compareTo(s2.name);
      i = (i != 0)?i:s1.firstName.compareTo(s2.firstName);
      i = (i != 0)?i:s1.matrikelNr - s2.matrikelNr;
      return i;
   }
}
```

```
MyComparator c = new MyComparator<Student>();
if (c.compare(s1,s2) < 0)
   System.out.println("s1 kommt vor s2");
```

Collation: Alphabetische Sortierung

- Computer Systeme ordnen jedem Buchstaben einen Code zu, z.B. ASCII, Unicode, EBCDIC. Resultat des (String-)Vergleichs ist durch diese Ordnung festgelegt: A..Z,..,a..z
- In Auswertungen (z.B. Telefonbüchern) wird aber oft eine länderspezifische Sortierung gefordert.
- Mittels einer Collation (dt. Kollation = Einsortierungsregeln) kann eine spezifische Sortierreihenfolge festgelegt werden.

Sortierung Deutschland

DIN 5007 Variante 1:
```
ä und a sind gleich
ö und o sind gleich
ü und u sind gleich
ß und ss sind gleich
```

DIN 5007 Variante 2:
```
ä und ae sind gleich
ö und oe sind gleich
ü und ue sind gleich
ß und ss sind gleich
```

Sortierung Österreich

```
ä folgt auf a
ö folgt auf o
ü folgt auf u
ß folgt auf ss
St. folgt auf Sankt
```

Sortierung Finnland

```
å folgt auf z
ä folgt auf å
ö folgt auf ä
ü und y sind gleich
w folgt auf v (ab 2006)
```

Collation: Collator, Locale

- Collator-Klasse (java.text)
  Implementiert das Comparator-Interface und führt einen länderspezifischen Zeichenfolgenvergleich durch. Mit dieser Klasse erstellen Sie Such- und Sortierroutinen für Text.
  - `getInstance()`:
    Liefert eine Instanz mit den Ländereinstellungen des Systems (Erzeugung über Fabrikmuster).
  - `getInstance(Locale desiredLocale)`:
    Liefert eine Instanz mit den Einstellungen einer Land/Sprache-Kombination.
- Locale-Konstruktoren (java.util)
  - `Locale(String language);` language in ISO 639-1 alpha-2 oder alpha-3: de, fr, en, ...
  - `Locale(String language, String country);` country in ISO-3166: DE, FR, UK, US, CH, ...
  - `Locale(String language, String country, String variant);` variant: Variation von Locale.

```
Collator col = Collator.getInstance();
Collections.sort(a, col)
```

statische Methode.

Sortiert nach Default.

```
Locale loc = new Locale("de","CH");
Collator col = Collator.getInstance(loc);
Collections.sort(a, col)
```

Sortiert nach Schweizerdeutsch.

```
Collator col = Collator.getInstance(Locale.GERMAN);
Collections.sort(a, col)
```

Für etwas mehr als 100 Sprachen und Länder gibt es vordefinierte Konstanten.

Sortieralgorithmen

Sortier-Algorithmen: Annahmen

- Es wird nur nach dem Schlüssel sortiert, d.h. der Inhalt der sortierten Daten spielt keine Rolle.
- Die Art des Schlüssels spielt (für den Algorithmus) ebenfalls keine Rolle; es kann deshalb auch ein einzelner Buchstabe genommen werden.

S	O	R	T	I	E	R	B	E	I	S	P	I	E	L

Das Ziel der Sortierung ist es, eine Reihenfolge gemäss der alphabetischen Ordnung herzustellen:

B	E	E	E	I	I	I	L	O	P	R	R	S	S	T

Sortier-Algorithmen: Hilfsmethode

In den meisten Algorithmen werden Elemente vertauscht, es wird deshalb die Existenz folgender **`swap`**-Methode angenommen:

Methoden-Generic: nur Parameter einer Methode und lokale Variablen generisch.

```
private static <K> void swap(K[] k, int i, int j) {
   K h = k[i]; k[i] = k[j]; k[j] = h;
}
```

Aufruf mit swap(a, 4, 7), der konkrete Typ wird anhand des Aufruf-Typs bestimmt.

Sortier-Algorithmen: Einfache Algorithmen

Es gibt sehr viele unterschiedliche Sortieralgorithmen. Sie unterscheiden sich punkto Komplexität und Performance.

Insertion Sort

Der Spieler nimmt eine Karte nach der anderen auf und sortiert sie in die bereits aufgenommenen Karten ein.

Der Spieler nimmt die jeweils niedrigste der auf dem Tisch verbliebenen Karten auf und kann sie in der Hand links (oder rechts) an die bereits aufgenommenen Karten anfügen.

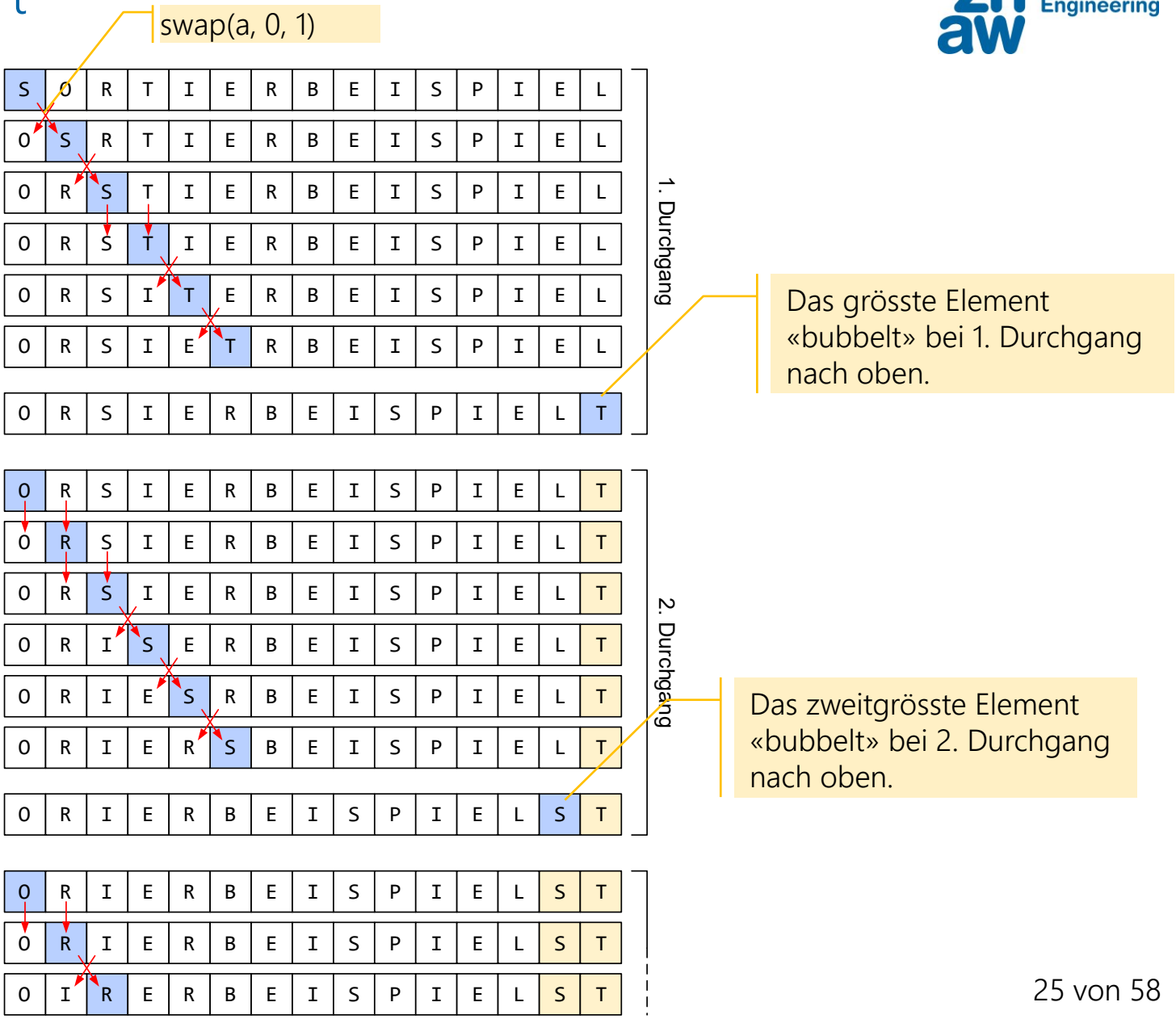
Selection Sort

Der Spieler nimmt alle Karten auf, macht einen Fächer daraus und fängt jetzt an, die Hand zu sortieren, indem er benachbarte Karten solange vertauscht, bis alle in der richtigen Reihenfolge liegen.

Bubble Sort

Bubble Sort

Bubble-Sort


swap(a, 0, 1)
1. Durchgang
2. Durchgang
Das grösste Element «bubbelt» bei 1. Durchgang nach oben.
Das zweitgrösste Element «bubbelt» bei 2. Durchgang nach oben.

Bubble-Sort: Algorithmus

- Nach dem 1. Durchgang hat man die folgende Situation:
  - Das grösste Element ist ganz rechts.
  - Alle anderen Elemente sind zwar zum Teil an besseren Positionen (also näher an der endgültigen Position), im Allgemeinen aber noch unsortiert.

- Beschreibung:
  - Der Array wird in mehreren Durchgängen von links nach rechts durchwandert.
  - Dabei werden Nachbarfeldern, die in falscher Reihenfolge stehen, vertauscht.
  - Dies wiederholt man so lange, bis der Array vollständig sortiert ist.
  - Das Wandern des grössten Elementes ganz nach rechts kann man mit dem Aufsteigen von Luftblasen in einem Aquarium vergleichen (BubbleUp).
  - In unserem Beispiel (15 Buchstaben) sind spätestens nach 14 Durchgängen alle Elemente an ihrer endgültigen Position, folglich ist das Array geordnet.

Bubble-Sort

S	O	R	T	I	E	R	B	E	I	S	P	I	E	L
O	R	S	I	E	R	B	E	I	S	P	I	E	L	T
O	R	I	E	R	B	E	I	S	P	I	E	L	S	T
O	I	E	R	B	E	I	R	P	I	E	L	S	S	T
I	E	O	B	E	I	R	P	I	E	L	R	S	S	T
E	I	B	E	I	O	P	I	E	L	R	R	S	S	T
E	B	E	I	I	O	I	E	L	P	R	R	S	S	T
B	E	E	I	I	I	E	L	O	P	R	R	S	S	T
B	E	E	I	I	E	I	L	O	P	R	R	S	S	T
B	E	E	I	E	I	I	L	O	P	R	R	S	S	T
B	E	E	E	I	I	I	L	O	P	R	R	S	S	T

Originalarray

Nach 1. Durchgang

Nach 2. Durchgang

Nach 10. Durchgang sortiert.

Bubble-Sort: Implementation

Die folgende Java-Methode BubbleSort ordnet ein Array der Länge n, indem sie n-1 mal «bubbleUp» auf den noch ungeordneten Teil des Arrays anwendet.

```
static void BubbleSort1(char[] a) {
   for (int k = a.length-1; k > 0; k--) {
      // bubbleUp
      for (int i = 0; i < k; i++) {
         if (a[i] > a[i + 1]) swap (a, i, i + 1);
      }
   }
}
```

Bubble-Sort: Generische Implementation

Die folgende Java-Methode BubbleSort ordnet ein Array der Länge n, indem sie n-1 mal «bubbleUp» auf den noch ungeordneten Teil des Arrays anwendet.

```
static <T extends Comparable> void BubbleSortG(T[] a) {
   for (int k = a.length-1; k > 0; k--){
      // bubbleUp
      for (int i = 0; i < k; i++) {
         if (a[i].compareTo(a[i+1]) > 0) swap (a, i, i+1);
      }
   }
}
```

Bubble-Sort: Optimierung

- Feststellung: Daten sind schon nach dem 10. Durchgang sortiert.
- Dies liegt daran, dass sich, wie oben bereits erwähnt, bei jedem Durchgang auch die Position der noch nicht endgültig sortierten Elemente verbessert.
- Wir können zwar den ungünstigsten Fall konstruieren, bei dem tatsächlich alle Durchgänge benötigt werden. Im allgemeinen können wir aber Bubble-Sort bereits nach einer geringeren Anzahl von Durchgängen abbrechen – im günstigsten Fall sind die Daten bereits nach dem 1. Durchgang sortiert.

S	O	R	T	I	E	R	B	E	I	S	P	I	E	L
O	R	S	I	E	R	B	E	I	S	P	I	E	L	T
O	R	I	E	R	B	E	I	S	P	I	E	L	S	T
O	I	E	R	B	E	I	R	P	I	E	L	S	S	T
I	E	O	B	E	I	R	P	I	E	L	R	S	S	T
E	I	B	E	I	O	P	I	E	L	R	R	S	S	T
E	B	E	I	I	O	I	E	L	P	R	R	S	S	T
B	E	E	I	I	I	E	L	O	P	R	R	S	S	T
B	E	E	I	I	E	I	L	O	P	R	R	S	S	T
B	E	E	I	E	I	I	L	O	P	R	R	S	S	T
B	E	E	E	I	I	I	L	O	P	R	R	S	S	T

Originalarray

Nach 1. Durchgang

Nach 2. Durchgang

Nach 10. Durchgang sortiert.

Bubble-Sort: Optimierung

- Bei jedem Durchgang testen, ob überhaupt etwas vertauscht wurde.
- Wenn in einem Durchgang nichts mehr vertauscht wurde, sind wir fertig.

```
static void BubbleSort1(char[] a) {
   for (int k = a.length-1; k > 0; k--) {
      boolean noSwap = true;
      for (int i = 0; i < k; i++) {
         if ( a[i] > a[i + 1])  {
            swap (a, i, i + 1);
            noSwap = false;
         }
      }
      if (noSwap) break;
   }
}
```

Übung

Schreiben Sie eine Methode, die überprüft ob ein Array (comparable) sortiert ist.

```
public boolean isSorted(Comparable[] arr) {
   for (int i = 1; i < arr.length; i++) {
      if (arr[i-1].compareTo(arr[i]) > 0) return false;
   }
   return true;
}
```

Dasselbe mit Comparator:

```
Comparator<Student> byAge = Comparator.comparingInt(Student::getAge);
```

Übung

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

Geht auch rekursiv – macht aber wenig Sinn:

Bubble-Sort: Laufzeit

- Wenn n die Anzahl der Elemente des Arrays A sind, dann wird die innere Schleife von Bubble-Sort
  - beim 1. Durchgang n-1 mal durchlaufen
  - beim 2. Durchgang n-2 mal durchlaufen
  - beim x. Durchgang n-x mal durchlaufen

- Wenn k der Aufwand für den Vergleich und die swap-Anweisungen in der inneren Schleife ist, ergibt sich daher als Laufzeit für den Worst-Case:

$$k \cdot ((n-1) + (n-2) + \ldots + 2 + 1) = k \cdot n \cdot (n-1) \cdot ½$$

Bubble-Sort: Aufwand

$$k \cdot ((n-1) + (n-2) + \ldots + 2 + 1) = k \cdot n \cdot (n-1) \cdot ½$$

Damit ergibt sich für Bubble-Sort qualitativ folgender Aufwand:

Best-Case:	Average-Case[1)]:	Worst-Case:
$k \cdot ((n-1)$	$(3/8) \cdot k \cdot n \cdot (n-2)$	$k \cdot n \cdot (n-1) \cdot ½$

Für die Grössenordnung folgt daraus:

Best-Case:	Average-Case:	Worst-Case:
$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Big Omega	Big Theta	Big O

Selection Sort

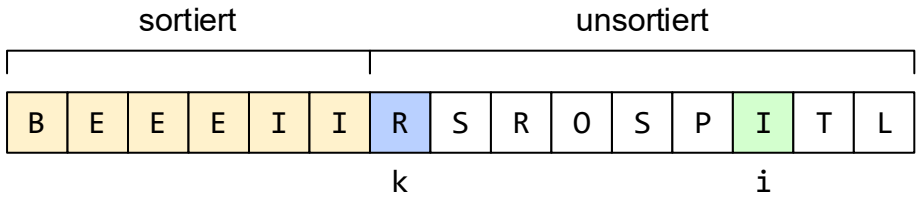
Selection-Sort

- Idee: Teile den Bereich in zwei Teile auf:
  - einen sortierten Teil
  - einen nicht sortierten Teil

- Invariante:
  Am Anfang k = 0

$$\forall n; n > 0 \wedge n < k; a[n-1] \leq a[n]$$

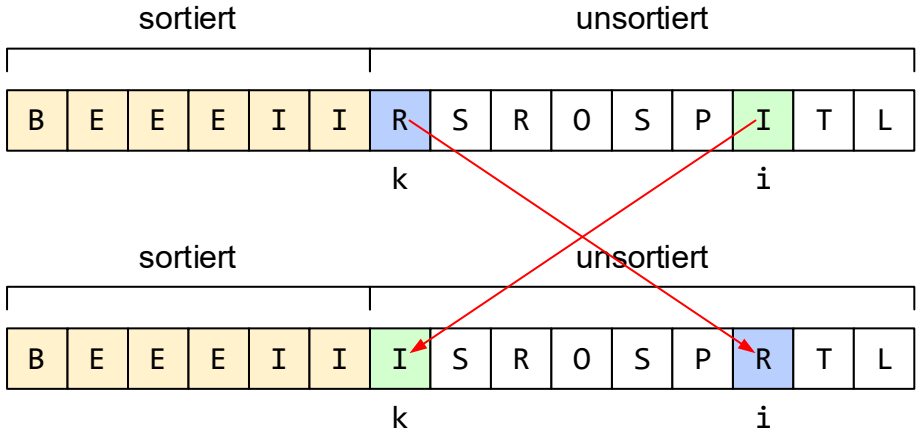
- Frage: welches Element i aus der unsortierten Restmenge muss ich auswählen, damit ich den sortierten Bereich um 1 vergrössern kann?



Selection-Sort: Algorithmus

- Suche jeweils das kleinste der verbleibenden Elemente und ordne es am Ende der bereits sortierten Elemente ein.

- In einem Array A mit dem Indexbereich 0..h sei k die Position des ersten Elements im noch nicht sortierten Bereich und i die Position des kleinsten Elementes in diesem Bereich.
- Wenn wir nun A[k] und A[i] vertauschen, dann haben wir den sortierten Bereich um ein Element vergrössert.
- Wenn wir diesen Vorgang so lange wiederholen, bis k = n gilt, ist das ganze Array sortiert.



Selection-Sort: Algorithmus

Java-Methode für den Selection-Sort:

Obere Grenze des sortierten Bereichs.

```
static void SelectionSort(char[] a) {
   for (int k = 0; k < a.length; k++) {
      int min = k;
      for (int i = k + 1; i < a.length; i ++) {
         if (a[i] < a[min]) min = i;
      }
      if (min != k) swap (a, min, k);
   }
}
```

Finde das kleinste Element.

Falls das kleinstes Element nicht schon am richtigen Platz: vertauschen.

Selection-Sort: Laufzeit

- Wenn n die Anzahl der Elemente des Arrays A sind, dann wird die innere Schleife von Selection-Sort (analog Bubble-Sort)
  - beim 1. Durchgang n-1 mal durchlaufen
  - beim 2. Durchgang n-2 mal durchlaufen
  - beim x. Durchgang n-x mal durchlaufen
- Es wir aber höchstens ein Swap pro Ausführung der äusseren Schleife ausgeführt.
- Wenn k_1 der Aufwand für den Vergleich und k_2 der Aufwand für die swap-Anweisungen in der inneren Schleife ist, ergibt sich daher als Laufzeit für den Worst-Case:

$$k_1 \cdot n \cdot (n-1) \cdot ½ + k_2 \cdot (n-1)$$

Aufwand vergleichen

Selection-Sort: Aufwand

$$k_1 \cdot n \cdot (n-1) \cdot ½ + k_2 \cdot (n-1)$$

Der minimale Aufwand, der maximale Aufwand und der mittlere Aufwand ist bei diesem naiven Verfahren gleich. Für die Grössenordnung folgt daraus:

Alle Fälle:

$O(n^2)$

Vergleich zum Bubble-Sort:

- Vorteil: Deutlich weniger Swap-Aufrufen als Bubble Sort.
- Nachteil: «Vorsortiertheit» kann nicht ausgenutzt werden.

Insertion Sort

Insertion-Sort

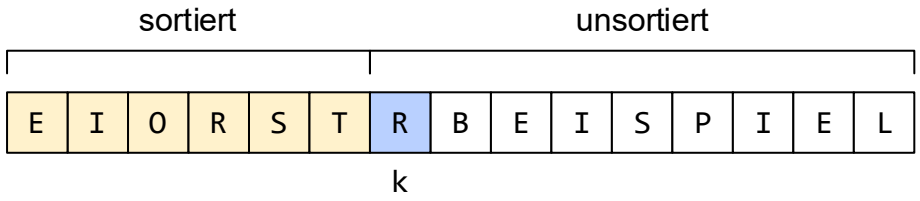
- Idee: Teile den Bereich in zwei Teile auf:
  - einen sortierten Teil
  - einen nicht sortierten Teil

Analog Selection-Sort.

- Invariante:
  Am Anfang k = 0

$$\forall n; n > 0 \wedge n < k; a[n-1] \leq a[n]$$

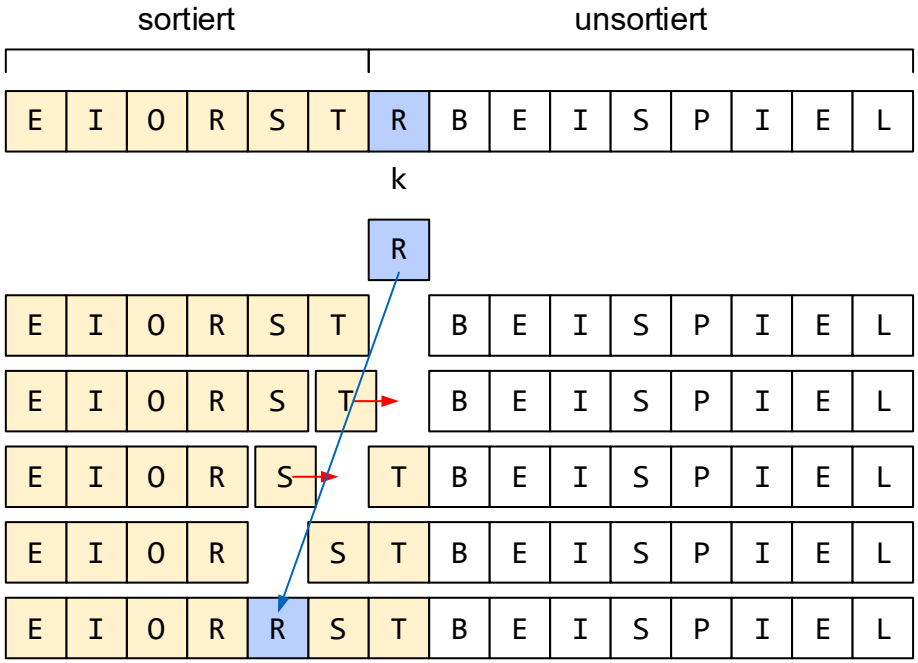
- Vorgehen: Aus dem noch unsortierten Teil entnehmen wir das Element ganz links und ordnen es in den sortierten Teil an der richtigen Stelle ein.



Insertion-Sort

- Wir entnehmen das Element ganz links vom unsortierten Teil.
- Die entstehende Lücke wird nach links verschoben, bis die korrekte Position für das Element gefunden wurde.
- Dort wird das Element eingeordnet.

Resultat: Der sortierte Bereich wurde um ein Element vergrössert.



Insertion-Sort Implementation

Java-Methode für den Insertion-Sort:

```
static void InsertionSort(char[] a) {
   for (int k = 1; k < a.length; k++) {
      char x = a[k];
      int i;
      for (i = k; ((i > 0) && (a[i-1] > x)); i--)
         a[i] = a[i-1];
      a[i] = x;
   }
}
```

Element das eingeordnet werden soll.

Lücke verschieben.

Muss die Lücke noch verschoben werden?

Element einfügen.

Insertion-Sort: Laufzeit

- Wenn n die Anzahl der Elemente des Arrays sind, dann wird die innere Schleife von Selection-Sort:
  - beim 1. Durchgang 1 mal durchlaufen
  - beim 2. Durchgang 2 mal durchlaufen
  - beim n-1 Durchgang n-1 mal durchlaufen
- Es werden analog Anzahl Swaps ausgeführt.
- Wenn k der Aufwand für den Vergleich und für die Verschiebe-Anweisungen ist, ergibt sich daher als Laufzeit für den Worst-Case (analog Bubble-Sort):

$$k \cdot n \cdot (n-1) \cdot ½$$

Insertion-Sort: Aufwand

$k \cdot n \cdot (n-1) \cdot ½$

Damit ergibt sich für Insertion-Sort qualitativ folgender Aufwand:

Best-Case[1)]:	Average-Case[2)]:	Worst-Case:
$k \cdot ((n-1)$	$k \cdot n \cdot (n-1) \cdot ¼$	$k \cdot n \cdot (n-1) \cdot ½$

Für die Grössenordnung folgt daraus:

Best-Case:	Average-Case:	Worst-Case:
$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$

Laufzeit und Ordnung

Sortierung: Vergleich Laufzeit und Ordnung

	Best-Case:	Average-Case:	Worst-Case:
Bubble-Sort:	$k \cdot ((n-1)$	$(3/8) \cdot k \cdot n \cdot (n-2)$	$k \cdot n \cdot (n-1) \cdot ½$
Selection-Sort:	$k_1 \cdot n \cdot (n-1) \cdot ½ + k_2 \cdot (n-1)$		
Insertion-Sort:	$k \cdot ((n-1)$	$k \cdot n \cdot (n-1) \cdot ¼$	$k \cdot n \cdot (n-1) \cdot ½$
Bubble-Sort:	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection-Sort:	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Insertion-Sort:	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$

k = Vergleichs- UND Swap-Operation
k_1 = Vergleichsoperation
k_2 = Swap-Operation

Sortierung: Laufzeitvergleich

- Testbedingungen
  - Array mit N=10000, N=20000, N=30000 und N=40000 ganzer positiver Zahlen.
  - erst dann fallen nennenswerte Laufzeiten an.
  - mit Hilfe eines Zufallszahlengenerators wird ein Array mit der gewünschten Zahl von Elementen erzeugt (der Startwert immer gleich).

- Die Daten werden jeweils zweimal sortiert:
  - in einer zufälligen, unsortierten Reihenfolge
  - sortiert
  - (umgekehrt sortiert)

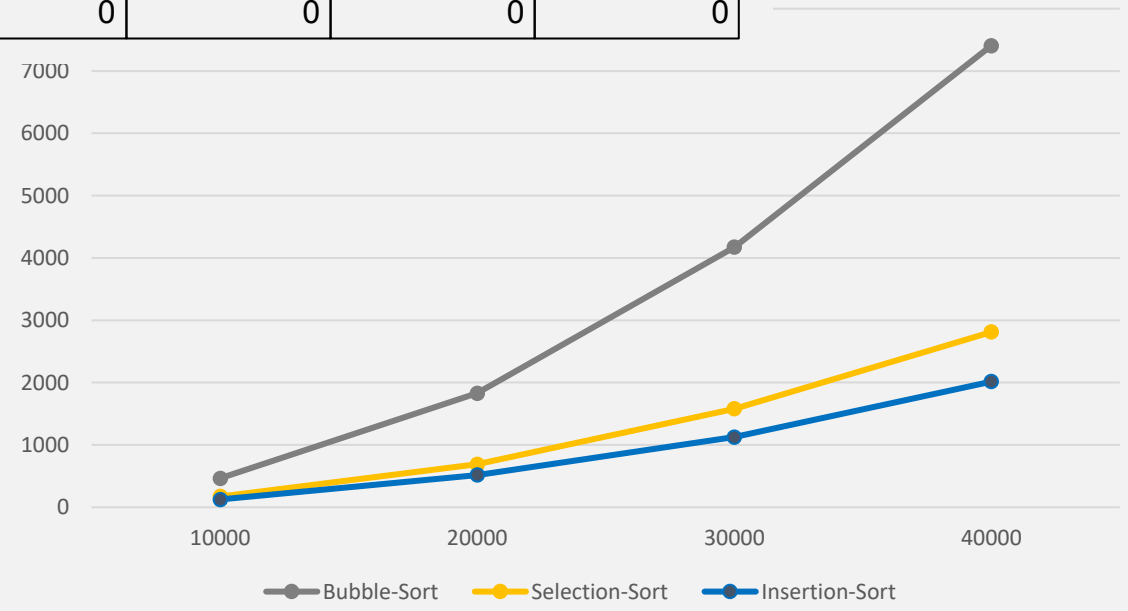
Sortierung: Laufzeitvergleich

unsortiert	N = 10'000	N = 20'000	N = 30'000	N = 40'000
Bubble-Sort	463	1'828	4'172	7'406
Selection-Sort	172	688	1'578	2'812
Insertion-Sort	122	516	1'125	2'016

sortiert	N = 10'000	N = 20'000	N = 30'000	N = 40'000
Bubble-Sort	0	0	0	0
Selection-Sort	175	711	1'578	2'812
Insertion-Sort	0	0	0	0

Ergebnisse der Laufzeitmessungen in Millisekunden.

Laufzeiten unsortiert.



Sortierung: Ordnung vs. Laufzeit

- Die Ordnung besagt, wie stark sich der Aufwand bei einer Veränderung der (Anzahl der) Eingangsdaten verändern.
  $O(n^2)$: Verdoppelung von n → Aufwand wird 4-mal grösser

- Die Laufzeit besagt, wie lange das Programm benötigt.
  Ist von vielen Faktoren abhängig, wie Rechnergeschwindigkeit, verwendeter Programmiersprache, Cache, Compilereinstellungen, bei Mehrprozess-Betriebssystemen: auch Auslastung der Maschine und Priorität des Prozesses.

- Laufzeit kann für unbekannte n folgendermassen extrapoliert werden
  - Ansatz nach Aufwand, Bsp.: $O(n^2)$ → z.B. Polynom 2-ten Grades: $k_1 * n^2 + k_2 * n + k_3$
  - Messen von verschiedenen Laufzeiten (Selection-Sort-Beispiel):
    1. $0.8 = k_1 \cdot 10'000^2 + k_2 \cdot 10'000 + k_3$
    2. $3.1 = k_1 \cdot 20'000^2 + k_2 \cdot 20'000 + k_3$
    3. $7.1 = k_1 \cdot 30'000^2 + k_2 \cdot 30'000 + k_3$
  - Nach k_1, k_2 und k_3 auflösen → $k_1 = 8.5 \cdot 10^{-9}$, $k_2 = -2.5 \cdot 10^{-3}$, $k_3 = 0.2$

Sortierung: Ordnung vs. Laufzeit

- Für grosse n fallen die Terme niedriger Ordnung nicht ins Gewicht.
- Oftmals wird daher nur der erste Koeffizient bestimmt.
  - 7.1 = $k_1 \cdot 30'000^2$, nach k1 auflösen → $k_1 = 7.1 / (9 \cdot 10^{-8})$

Übung

Extrapolieren Sie die Laufzeit des Selection-Sort bei N = 1'000'000 unter der Annahme, dass sich diese mit einem Polynom 2. Grades beschreiben lässt.

6000.2

Stabilität

Sortieralgorithmen: Stabilität

Ein wichtiger Punkt bei Sortieralgorithmen ist die Art wie Elemente mit gleichem Schlüssel behandelt werden.

Sei S = ((k0, e0), … , (kn-1, en-1)) eine Sequenz von Elementen:
Ein Sortieralgorithmus heisst **stabil (stable)**, wenn für zwei beliebige Elemente (ki, ei) und (kj, ej) mit gleichem Schlüssel ki = kj und i < j (d.h. Element i kommt vor Element j), i < j auch noch nach dem Sortieren gilt (Element i kommt immer noch vor Element j).

Sortieralgorithmen: Stabilität

Vreni	IT2b
Max	IT2b
Moni	IT2a
Sepp	IT2a
Köbi	IT2b
Fritz	IT2b
Jenny	IT2a

1. Sortiere nach Namen

Fritz	IT2b
Jenny	IT2a
Köbi	IT2b
Max	IT2b
Moni	IT2a
Sepp	IT2a
Vreni	IT2b

Fritz	IT2b
Jenny	IT2a
Köbi	IT2b
Max	IT2b
Moni	IT2a
Sepp	IT2a
Vreni	IT2b

Sortiert nach Namen

2. Sortiere nach Klassen

Jenny	IT2a
Moni	IT2a
Sepp	IT2a
Fritz	IT2b
Köbi	IT2b
Max	IT2b
Vreni	IT2b

Stabil: Die Sortierung der Namen innerhalb der Klassen bleibt erhalten.

Sortieralgorithmen: Stabilität

Bubble-Sort:	$O(n^2)$	stabil
Selection-Sort:	$O(n^2)$	instabil
Insertion-Sort:	$O(n^2)$	stabil
Quick-Sort:	O(n log n)	instabil
Merge-Sort:	O(n log n)	stabil

Quick- und Merge-Sort kommen später.

Zusammenfassung

- Anwendungen des Sortierens
- Sortierschlüssel
  - Comparable
  - Comparator
- Einfache Sortieralgorithmen
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- Laufzeitverhalten und Ordnung
- Stabilität