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

Einführung

Motivation

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

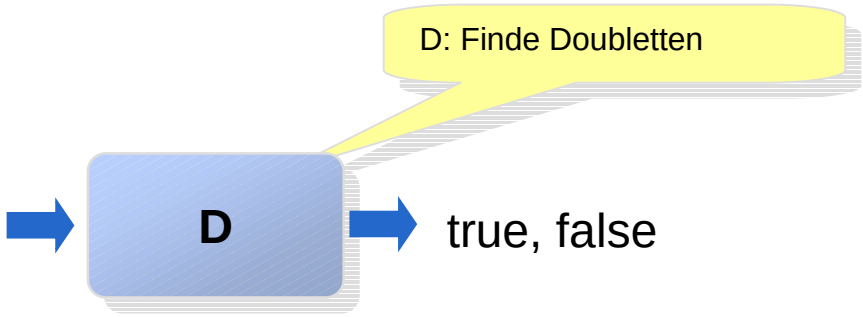
- Sortieren zur Steigerung der **Effizienz eines Algorithmus:** gewisse (schnelle) Algorithmen funktionieren nur, wenn die Daten sortiert sind, wie zum Beispiel Binäre Suche

Motivation: sind zwei gleiche Karten im Spiel?

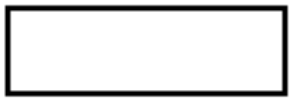
- Joe hat an diesem Abend viel verloren; er hegt den Verdacht, dass nicht alles mit rechten Dingen zugegangen ist. Sicherheitshalber will er überprüfen, ob keine zusätzliche Karten ins Spiel eingebracht wurden, d.h. keine Karte doppelt vorhanden ist.

Algorithmus 1

- im ersten Algorithmus wird jeder Karte mit jeder verglichen:

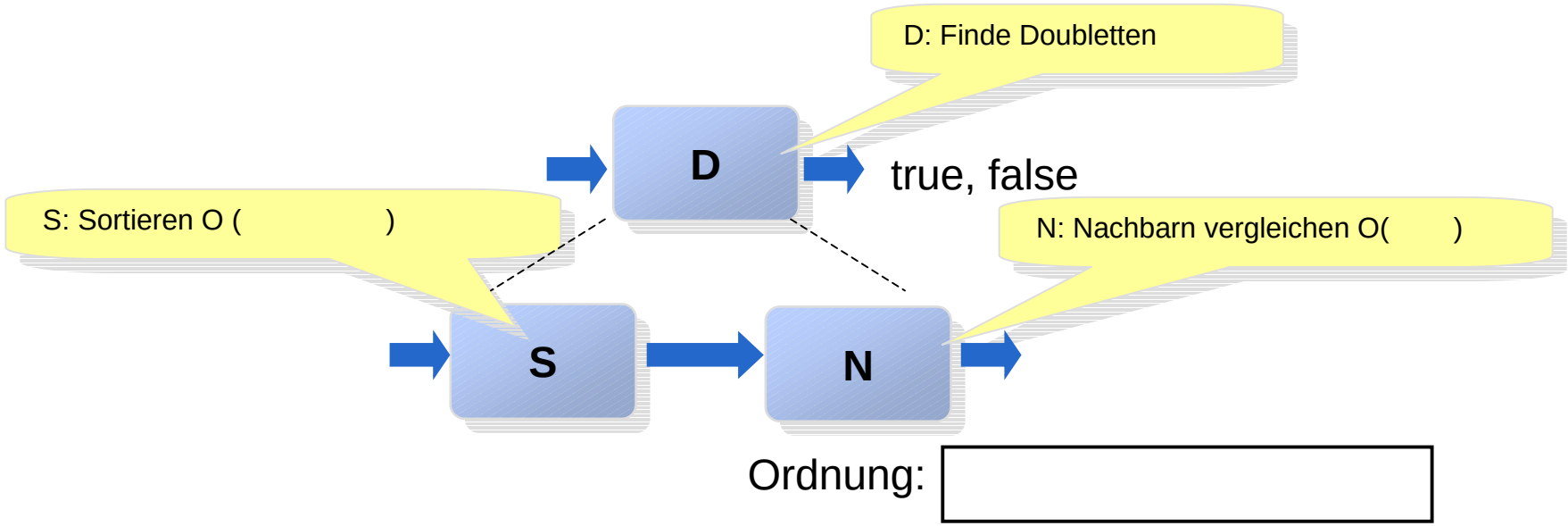



```
public static boolean duplicates(Object[] a) {
  for(int i=0; i<a.length; i++)
    for(int j=i+1; j<a.length; j++)
      if(a[i].equals(a[j]))
        return true;
  return false;
}
```

Ordnung: 

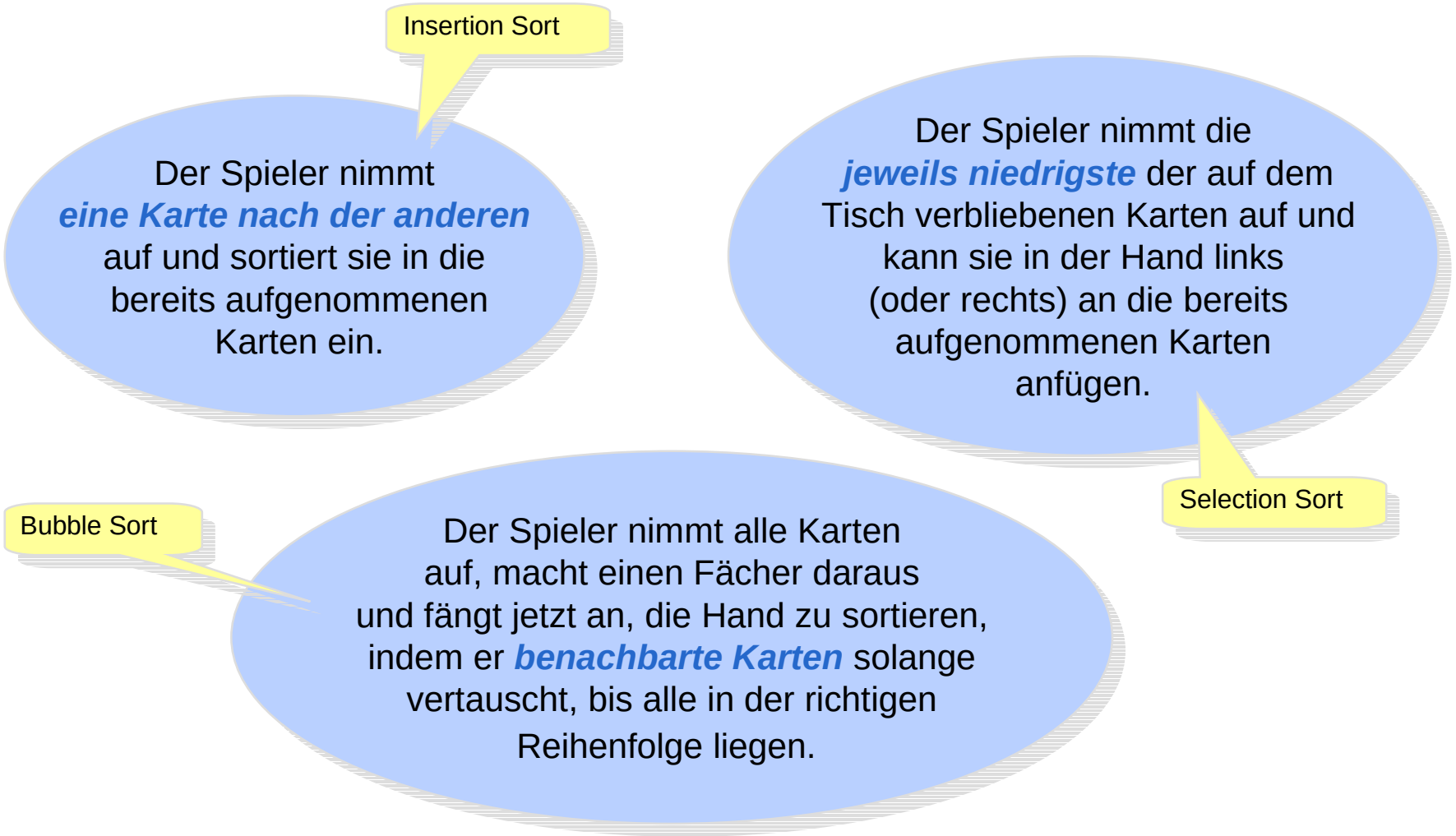
Algorithmus 2

- 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.



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

Wie sortiere ich einen Kartenstapel?



Internes Sortieren

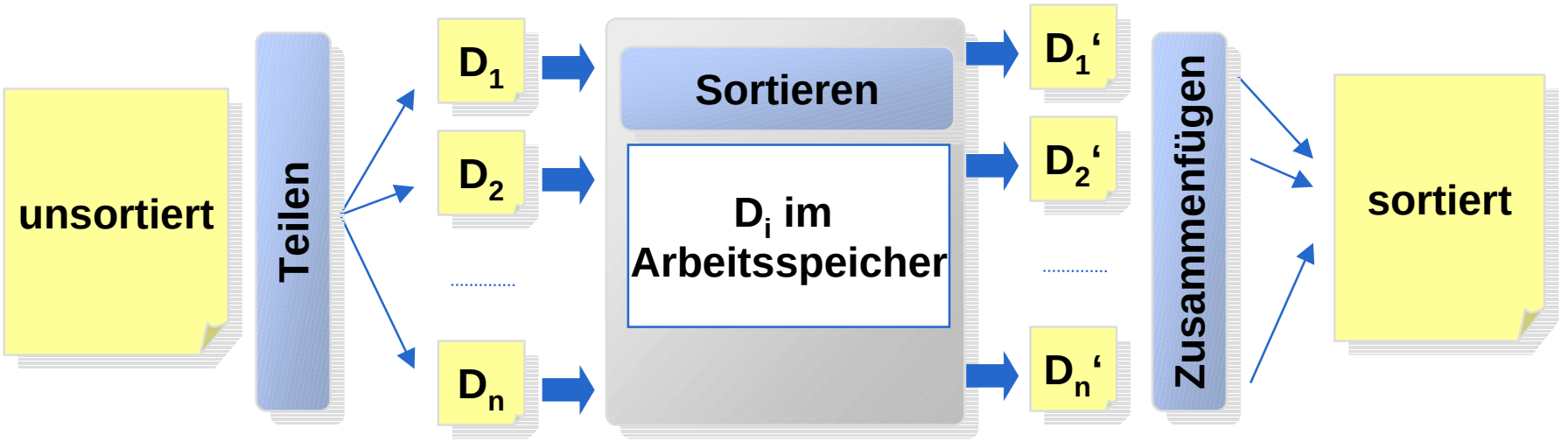
- Wenn die Anzahl der Datensätze und deren jeweiliger Umfang sich in Grenzen halten, kann man alle Datensätze im Arbeitsspeicher eines Computers sortieren.

- 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 mehr)

- **Teilen** die grosse zu sortierende Datei **D** in **n** Teile
  (klein genug, dass sie in den Hauptspeicher passen).

- Dateien werden in Speicher eingelesen, **intern sortiert** und wieder in Dateien geschrieben.

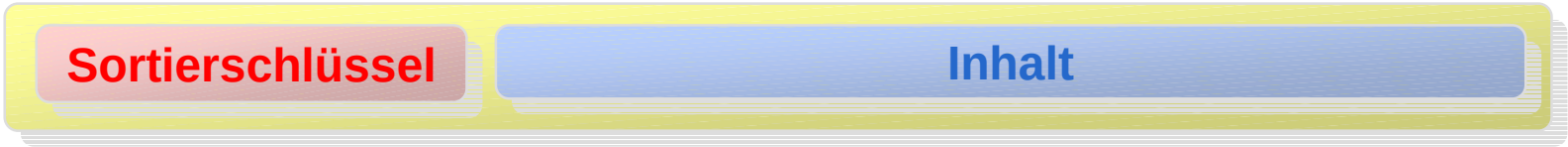
- Die sortierten Dateien werden schliesslich zu einer sortierten Datei **zusammengefügt** (merge).

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

Sortieren von Datensätzen: Sortierschlüssel

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;
}
```

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

MatrikelNr
Name, Vorname,
Alter,
Studiengang

Sortierschlüssel - Eigenschaften

- Die Sortierung nach [Alter] bzw. [Fachbereich] 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)*

- Für das Anlegen einer **relationalen Datenbank** mit einer Menge von Datensätzen sollte dagegen ein eindeutiger Schlüssel oder kurz Id vorhanden sein.

Vergleich von Sortierschlüsseln

School of Engineering

- 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
  - Comparable, Comparator

Wiederholung: Das Comparable<T> Interface

- In Java ist folgendes Interface definiert

```
public interface Comparable<T> {
     public int compareTo(T obj);
}
```

es sei `a.compareTo(b);`

- < 0 falls das T a kleiner als b
- == 0 falls das T a gleich b
- > 0 falls das T a grösser als b

Implementation der Klasse Student

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


   // name, vorname, 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");
```

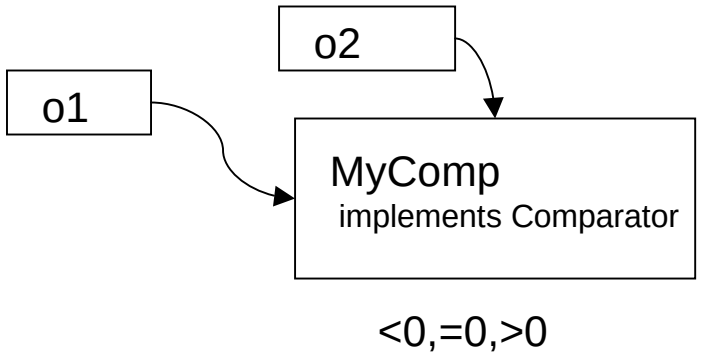
Wiederholung: Das Comparator Interface

- Nachteil des `Comparable`-Interfaces: es kann nur eine Sortier-Reihenfolge bestimmt werden.

- Lösung: ich lagere den Vergleich der Objekte in eine eigene Klasse aus -> Klasse die das `java.util.Comparator` Interface implementiert

```
public interface Comparator<T> {
     public int compare (T o1, T o2);
}
```

- o1 < o2 -> Wert kleiner 0
- o1 == o2 -> 0
- o1 > o2 -> Wert grösser 0



Klasse Student mit Comparator

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

// name, vorname, matrikelNr
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");
```

Alphabetische Sortierung

- Computer Systeme ordnen jedem Buchstaben einen Code zu, z.B. ASCII, Unicode, EBCDIC
- Resultat des (String-)Vergleichs durch diese Ordnung festgelegt: A..Z,..,a..z

- In Nachschlagewerken/Telefonbüchern wird aber eine länderspezifische Sortierung verwendet

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
```

Schweiz/Österreich

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

Finnisch

```
å kommt nach z
ä kommt nach å
ö kommt nach ä
ü und y sind gleich
```

java.text.Collator Klasse

- Implementiert das Comparator Interface
- Erzeugung über Fabrikmuster
  - getInstance() liefert eine Instanz mit den Ländereinstellungen des Systems
  - getInstance(<Locale>)
- java.util.Locale Konstruktoren
  - Locale(String language) ; language in ISO 639-1 : **de, fr, en,** .. http://www.sil.org/iso639-3/codes.asp?order=639_1
  - Locale(String language, String country); country in ISO-3166 : **DE, FR, UK, US, CH,**

    http://de.wikipedia.org/wiki/ISO-3166-1-Kodierliste

```
// Sort in default locale
Collator col = Collator.getInstance();
Collections.sort(a, col)
```

```
 // Sort in swiss german locale
Locale loc = new Locale("de","CH");
Collator col = Collator.getInstance(loc);
Collections.sort(a, col)
```

```
// Sort in German
Collator col = Collator.getInstance(Locale.GERMAN);
Collections.sort(a, col)
```

Ein paar Sprachen und Länder als 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 Buchstaben 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, 4,5);Typ wir anhand Aufruf-Typ bestimmt

Einfache Sortieralgorithmen

- **Bubble Sort:**
  - Sortieren durch Vertauschen von Nachbarfeldern.
- **Selection Sort:**
  - Sortieren durch Auswählen des jeweils kleinste der verbleibenden Elemente und Anhängen an die Reihe der bereits sortierten Elemente.
- **Insertion Sort:**
  - Sortieren durch Einfügen eines beliebigen Elementes aus den unsortierten Elementen an der richtigen Position in der Reihe der bereits sortierten Elemente.

Bubble Sort

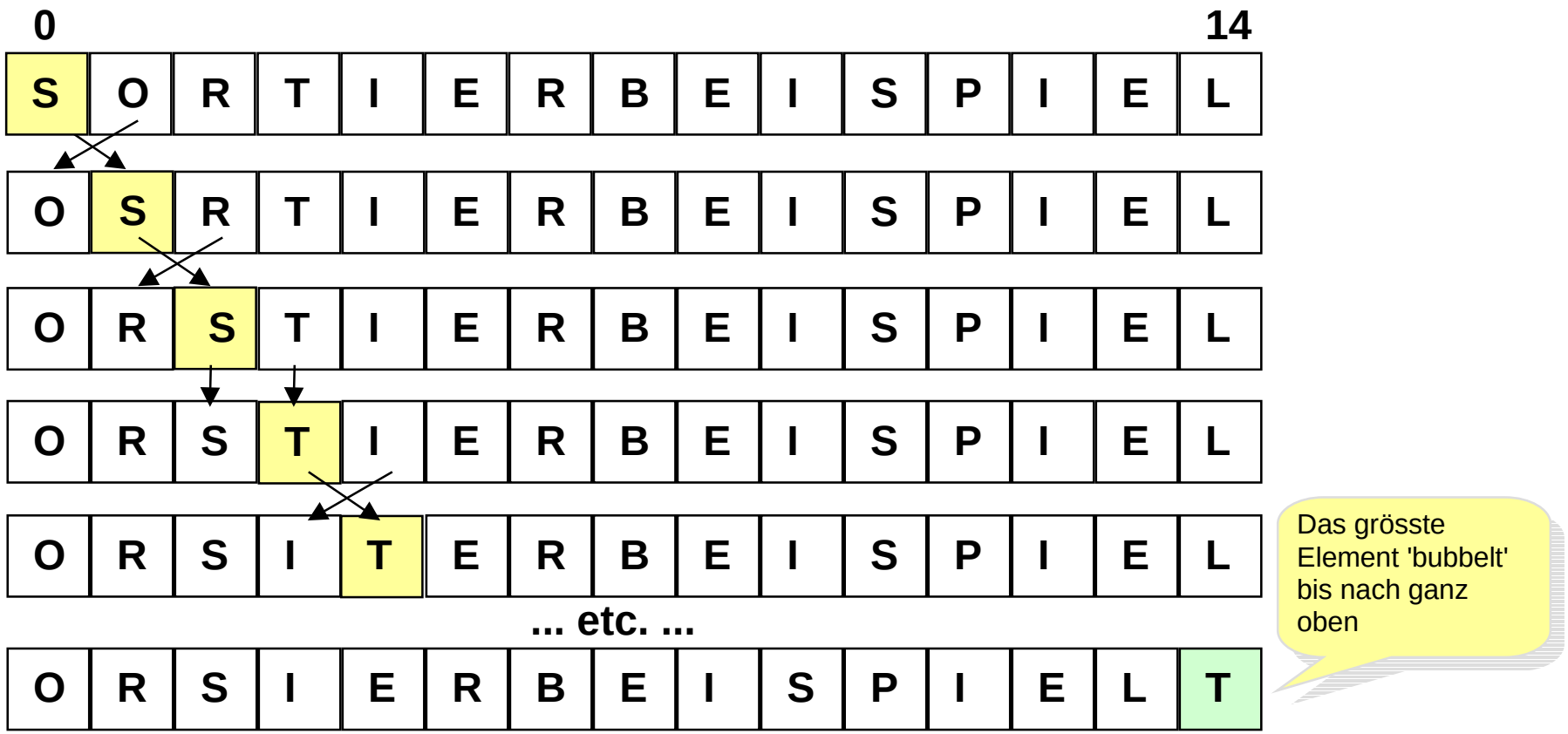
Bubble-Sort Algorithmus

- Beschreibung
  - Dieser Algorithmus sortiert ein Array von Datensätzen durch **wiederholtes Vertauschen von Nachbarfeldern**, die in falscher Reihenfolge stehen.
  - Dies **wiederholt** man so lange, **bis** der Array **vollständig sortiert** ist.

- Im Detail
  - Der Array wird in mehreren Durchgängen von links nach rechts durchwandert.
  - Bei jedem Durchgang werden alle Nachbarfelder verglichen und ggf. vertauscht.

- Nach dem 1. Durchgang hat man die folgende auf der nächsten Folie illustrierte 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**.

Bubble-Sort


0
14
S O R T I E R B E I S P I E L
O S R T I E R B E I S P I E L
O R S T I E R B E I S P I E L
O R S T I E R B E I S P I E L
O R S I T E R B E I S P I E L
... etc. ...
O R S I E R B E I S P I E L T
Das grösste Element 'bubbelt' bis nach ganz oben

Bubble-Sort

- Das Wandern des grössten Elementes ganz nach rechts kann man mit dem Aufsteigen von **Luftblasen** in einem Aquarium vergleichen:
  - Die grösste Luftblase ist soeben nach oben aufgestiegen (**BubbleUp**).
- Nach dem 1. Durchgang
  - das grösste Element also an seiner endgültigen Position.
- Für die restlichen Elemente müssen wir nun den gleichen Vorgang anwenden.
- Nach dem 2. Durchgang
  - das zweitgrösste Element an seiner endgültigen Position.
- Dies wiederholt sich für alle restlichen Elemente mit **Ausnahme des letzten**.

- In unserem Beispiel 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

Original-Array

nach 1. BubbleUp

nach 2. BubbleUp

... etc. ...

nach 10. BubbleUp: sortiert!

Bubble-Sort Implementation

Die folgende Java-Methode BubbleSort ordnet ein Array der Länge N, indem sie N 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 - Implementation Generisch

Die folgende Java-Methode BubbleSort ordnet ein Array der Länge N, indem sie N 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

- 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 Durchgänge benötigt werden, **im allgemeinen** können wir aber BubbleSort bereits nach einer geringeren Anzahl von Durchgängen **abbrechen** – im **günstigsten Fall**, sind die Daten bereits nach dem 1. Durchgang sortiert.

Bubble-Sort - optimiert

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

```
static void BubbleSort2(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 sortiert ist.

Bubble-Sort: Aufwand

- Wenn N = A.length die Anzahl der Elemente des Arrays A sind, dann wird die innere Schleife von BubbleSort
  - 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 die (swap-)Anweisungen in der inneren Schleife ist, ergibt sich daher als Laufzeit für den worst case:

$$k \times ((N-1) + (N-2) + \ldots + 2 + 1) = k \times N \times (N-1) / 2$$

Bubble-Sort: Aufwand (2/2)

$$k \times ((N-1) + (N-2) + \ldots + 2 + 1) = k \times N \times (N-1) \times ½$$

Damit ergibt sich für BubbleSort qualitativ folgender Aufwand:

Best Case	*Average Case*	*Worst Case*
$k \times (N-1)$	$(3/8) \times k \times N \times (N-2)$	$(1/2) \times k \times N \times (N-1)$

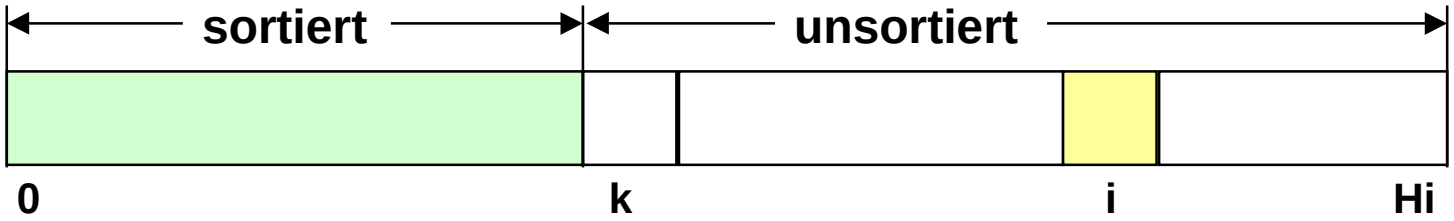
Für die Herleitung der Formel für den „average case“: Siehe weiterführende Literatur.

Für die Grössenordnung folgt daraus:

Best Case	*Average Case*	*Worst Case*
$O(N)$	$O(N^2)$	$O(N^2)$

Selection Sort

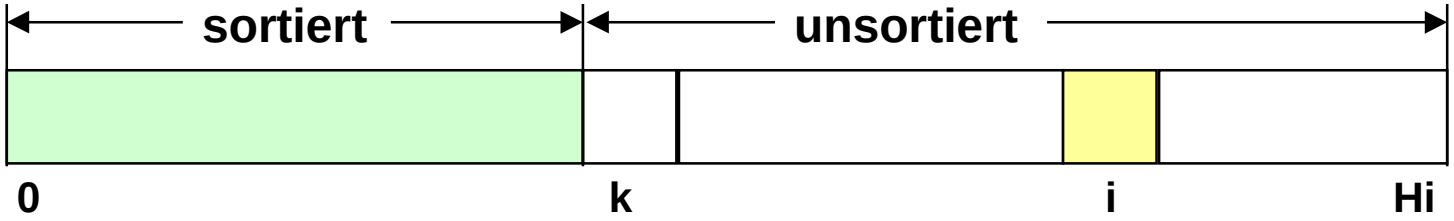
Selection-Sort



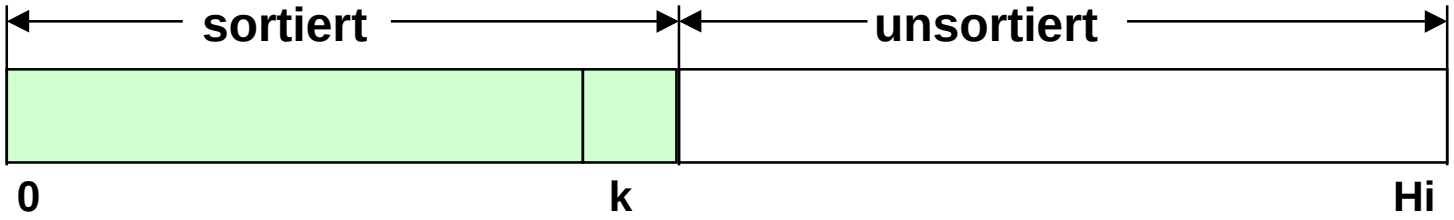
- Idee ich teile den Bereich in zwei Teile auf:
  - einen sortierten Teil
  - einen nicht sortierten Teil
- Invariante: $\forall n; n > 0 \wedge n < k; a[n-1] \leq a[n]$
- Am Anfang k = 0
- Frage: welches Element aus der (unsortierten) Restmenge muss ich auswählen,damit ich den sortierten Bereich um 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..Hi sei k die Postion des **ersten** Elements im noch nicht sortierten Bereich und i die Position des **kleinsten** Elementes in diesem Bereich

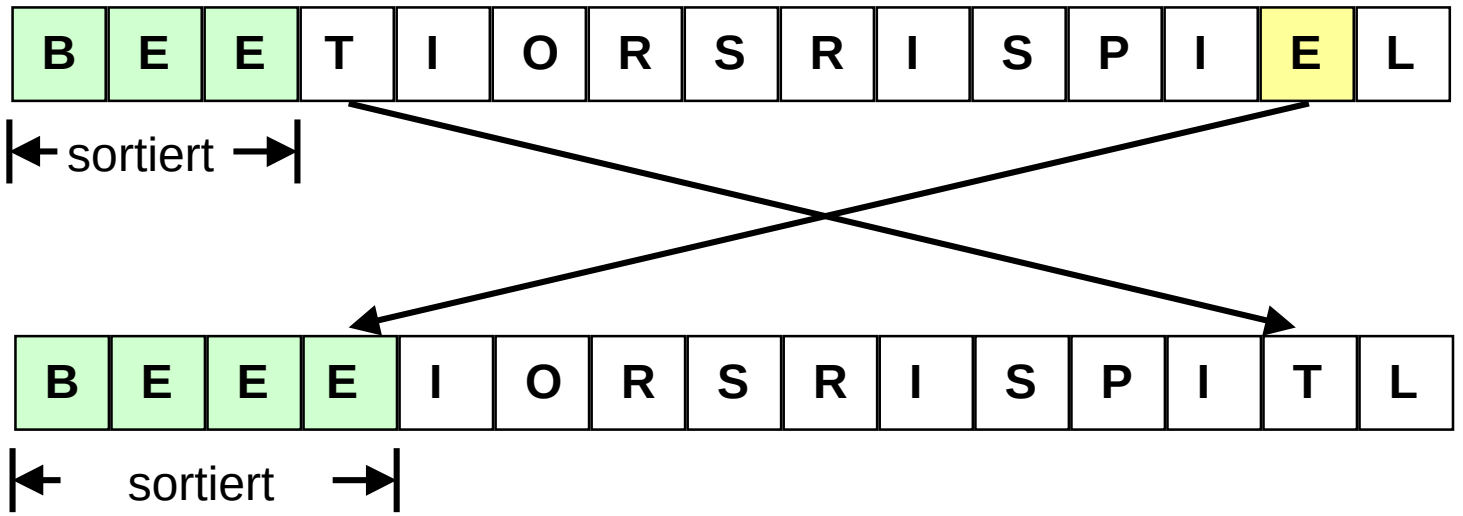


Wenn wir nun A[k] und A[i] vertauschen, dann haben wir den sortierten Bereich um **ein Element vergrössert**.



Selection-Sort

- Wenn wir diesen Vorgang so lange wiederholen, bis k == N gilt, ist das ganze Array sortiert.
- Für unser Sortierbeispiel ergibt sich für k = 3:



Selection-Sort

Java-Methode für den Selection-Sort:

```
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);
      }
}
```

Grenze des sortierten Bereichs

finde kleinstes Element

falls kleinstes Element nicht schon am richtigen Platz: vertausche

Selection-Sort: Aufwandsabschätzung

- die äussere Schleife wird (N - 1) mal durchlaufen
- die innere Schleife wird (N - k -1) mal durchlaufen
- Aufwand bestimmt durch den Vergleich in Schleife
- Aufwand von Selection Sort: $k_1*n^2 + k_2*n + k_3$: $O(N^2)$.
- Die Konstante k_1 kleiner als bei Bubblesort da weniger Vertauschungen

- Vorteil:
  - deutlich weniger Swap-Aufrufen als Bubble Sort.

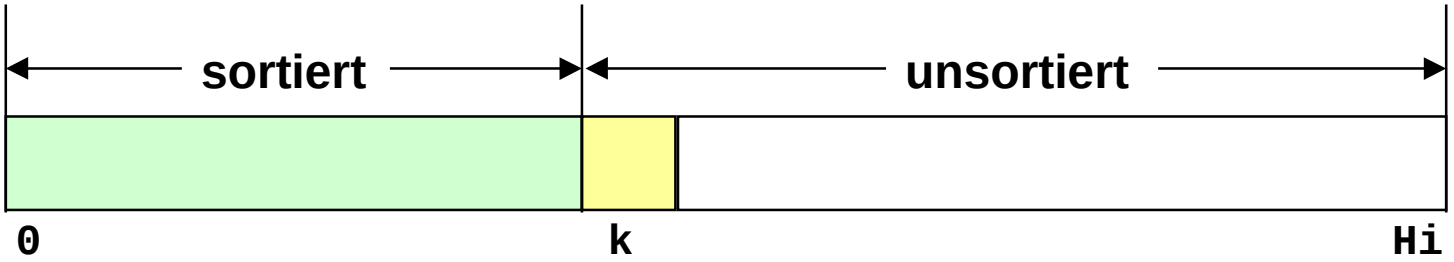
- Nachteil:
  - Vorsortiertheit kann nicht ausgenutzt werden

Insertion Sort

Sortieren durch Einsetzen: Insertion-Sort

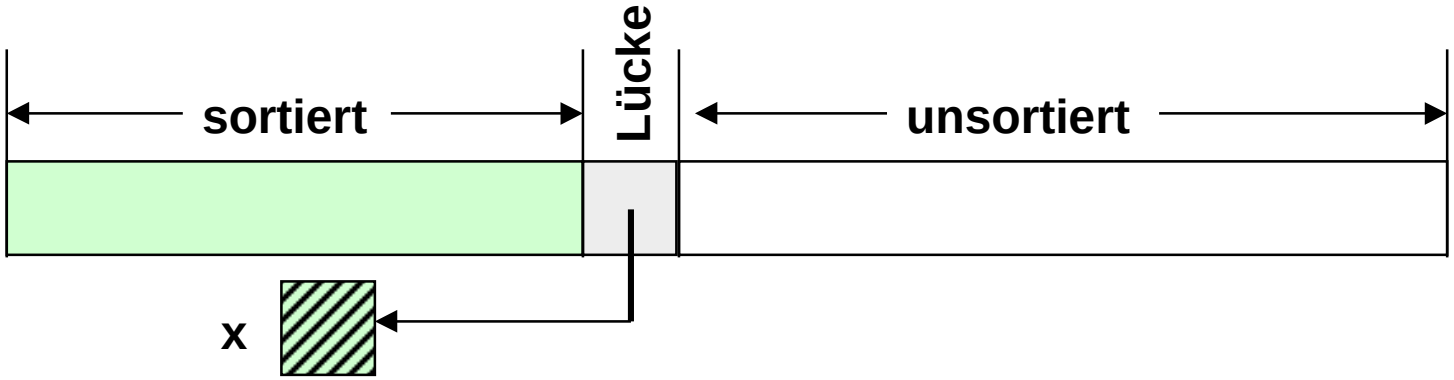
- Analogie zum Sortieren eines Kartenspieles:
  - Eine Karte nach der anderen wird aufgenommen und an der richtigen Stelle in die schon sortierten Karten eingeordnet.

- Algorithmus
  - der erste Teil eines Arrays sei bereits sortiert.
  - aus dem noch unsortierten Teil
    - *entnehmen wir ein Element*
    - *ordnen es in den sortierten Teil an der richtigen Stelle ein:*



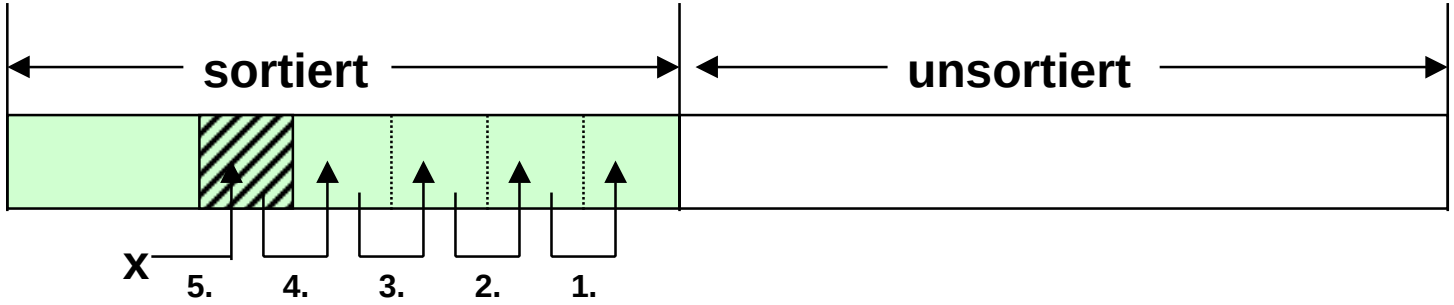
Das Element x=A[k] wird aufgenommen, also aus dem Array herausgenommen.

Insertion-Sort

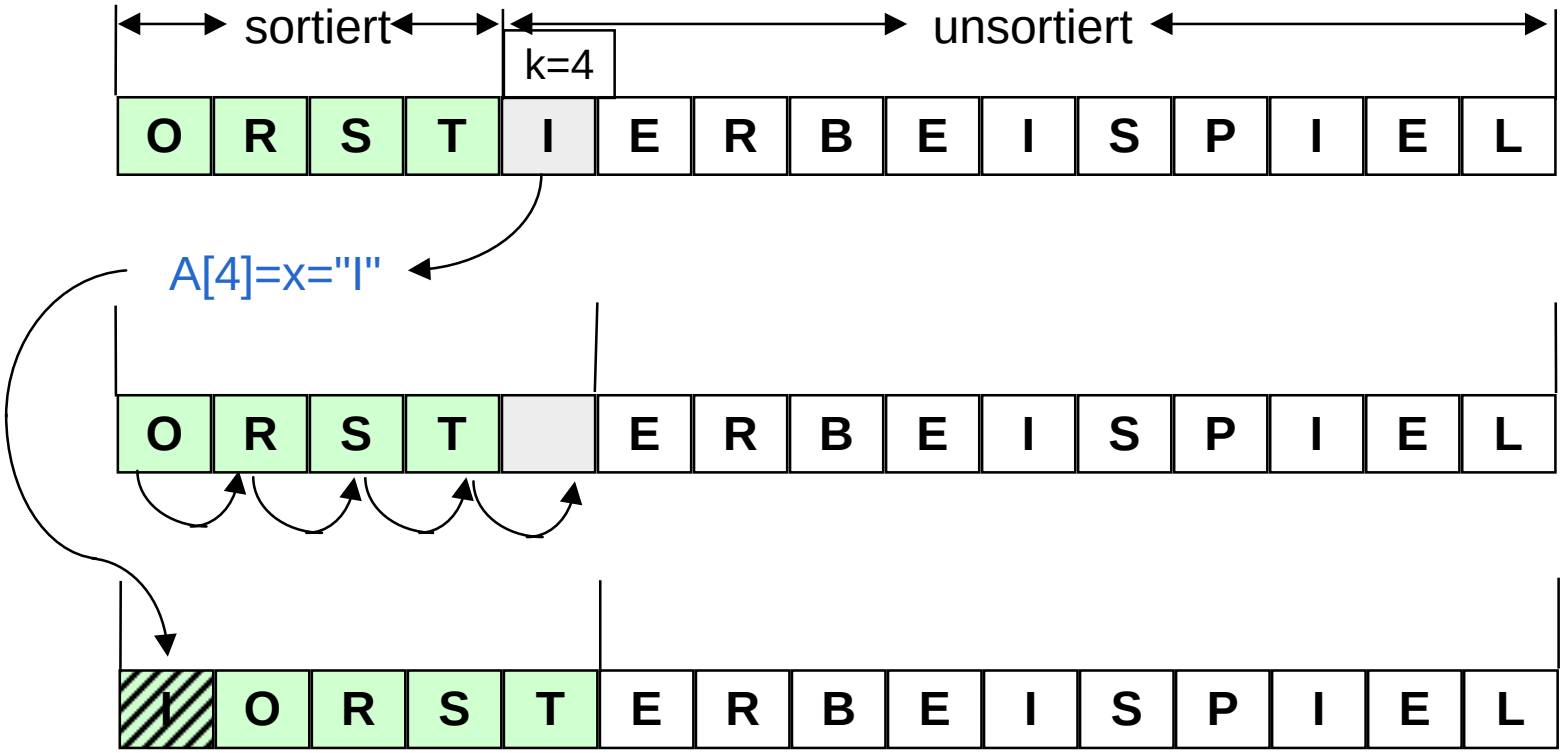


- die entstehende Lücke wird nach links verschoben,
- bis die korrekte Position für das Element x gefunden wurde.
- x wird dort eingeordnet.

Resultat: der sortierte Bereich wurde um eins vergrössert.



Insertion-Sort Beispiel



Insertion-Sort Implementation

Es folgt eine Java-Methode für den Insertion-Sort:

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

richtiger Platz?

Element das eingeordnet werden soll

finde Einfügestelle, verschiebe Elemente

verschiebe Lücke

Insertion-Sort: Aufwandsabschätzung

- Die Unterschiede zu Selection Sort :
  - im **Mittel** nur die Hälfte aller maximal notwendigen Vergleiche (bei **Selection Sort** müssen immer alle Vergleiche gemacht werden)
  - beim Verschieben der Lücke müssen mehr **Swap**-Aufrufe vorgenommen werden,
  - **InsertionSort** hat eine um so geringere Laufzeit, je besser das Array vorsortiert ist.

Anwendung von Insertion- oder Selection-Sort

- InsertionSort ist besser wenn:
  - Datensätze **relativ kurz** sind (Aufwand für die zusätzlich erforderlichen Swap-Aufrufe gering)
  - Daten relativ **gut vorsortiert** sind -> in diesen Fällen der effizienteste Sort-Algorithmus überhaupt

- SelectionSort ist besser wenn:
  - Datensätze relativ lang sind
  - Daten **völlig unsortiert** sind

Laufzeit und Ordnung

Laufzeitvergleich von Sortieralgorithmen

- 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)

Laufzeitvergleich von Sortieralgorithmen

Die Ergebnisse in Millisekunden der Laufzeitmessung bei zufällig geordneten Daten:

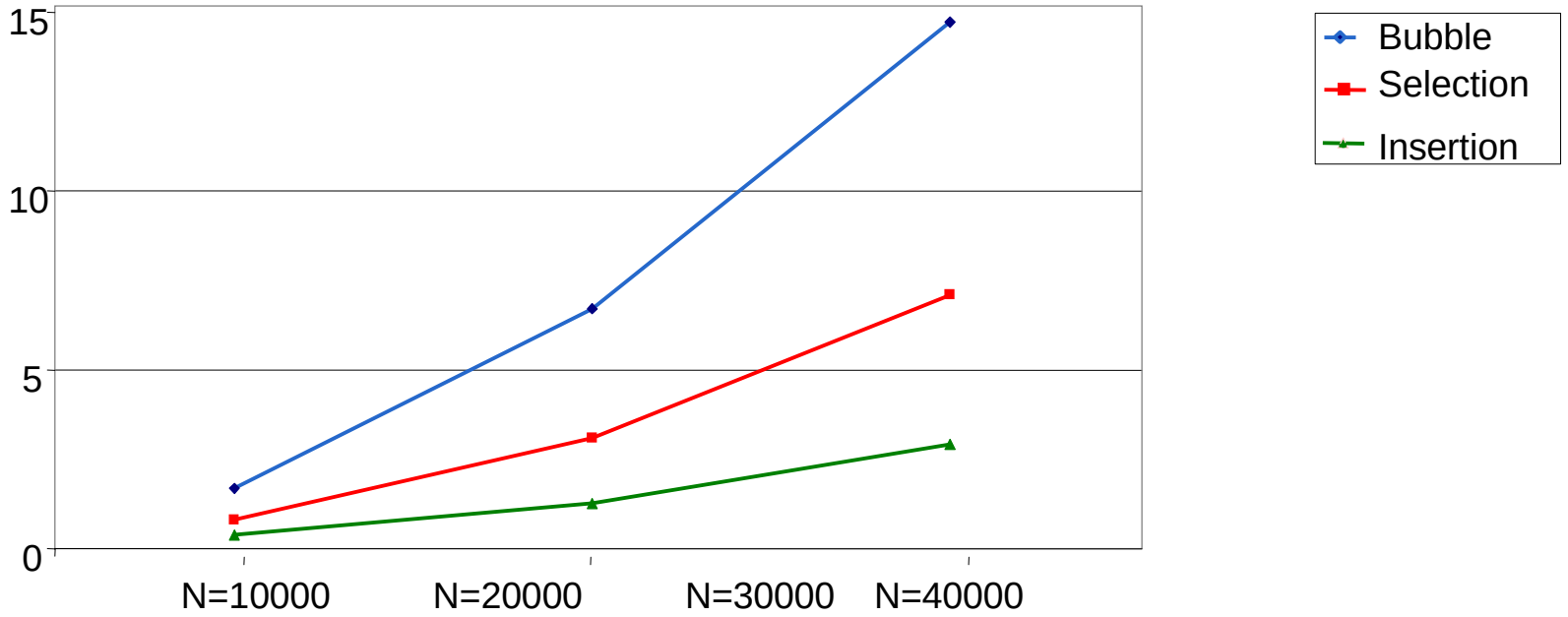
	N=10000	N=20000	N=30000	N=40000
Bubble	463	1828	4172	7406
Selection	172	688	1578	2812
Insertion	122	516	1125	2016

... und bei vorsortierten Daten:

	N=10000	N=20000	N=30000	N=40000
Bubble	0	0	0	0
Selection	175	711	1578	2812
Insertion	0	0	0	0

Laufzeitvergleich von Sortieralgorithmen

Laufzeitverhalten bei zufällig geordneten Daten:



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, Compilereinstellungen, bei Mehrprozess-Betriebssystemen: auch Auslastung der Maschine und Priorität des Prozesses.

- Laufzeit kann für unbekannte n folgendermassen extrapoliert werden
  - Ansatz nach Aufwand:
    - *Bsp: $O(n^2)$ ->Polynom 2-ten Grades: $k_1 * n^2 + k_2 * n + k_3$*
  - Messen von verschiedenen Laufzeiten
    - $0.8 = k_1 * 10'000^2 + k_2 * 10'000 + k_3$
    - $3.1 = k_1 * 20'000^2 + k_2 * 20'000 + k_3$
    - $7.1 = k_1 * 30'000^2 + k_2 * 30'000 + k_3$
  - nach k_1 , k_2 und k_3 auflösen

Ordnung vs. Laufzeit

- Für grosse n fallen die Terme niedriger Ordnung nicht ins Gewicht

- Oftmals wird nur der erste Koeffizient bestimmt.
  - 7.1 = k_1 * 30'000^2 -> nach k_1 auflösen

Übung

- Extrapolieren Sie die Laufzeit von Selection-Sort bei N = 1'000'000

  - exakt

  - bei Berücksichtigung nur eines Koeffizienten

Stabilität

Stabile Sortieralgorithmen

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

- Sei S = ((k_0, e_0), ... , (k_{n-1}, e_{n-1})) eine Sequenz von Elementen: Ein Sortieralgorithmus heisst *stabil (stable)*, wenn für zwei beliebige Elemente (k_i, e_i) und (k_j, e_j) mit gleichem Schlüssel
  $k_i = k_j$ 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).

Stabile Sortieralgorithmen Beispiel

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

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

Sortiere nach Klassen

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

Innerhalb Klasse
nach Namen sortiert

Stabile Sortieralgorithmen

Algorithmus	Effizienz	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*

Zusammenfassung

- Anwendungen des Sortierens
- Sortierschlüssel
  - Comparable
  - Comparator
- Einfache Sortieralgorithmen
  - Bubble Sort
    - *Algorithmus*
    - *optimierter Algorithmus*
    - *Aufwandsbetrachtung*
  - Selection Sort
    - *Idee: den sortierten Bereich erweitern*
    - *Algorithmus*
    - *Aufwandsbetrachtung*
  - Insertion Sort
    - *Idee: den sortierten Bereich erweitern*
    - *Algorithmus*
    - *Aufwandsbetrachtung*