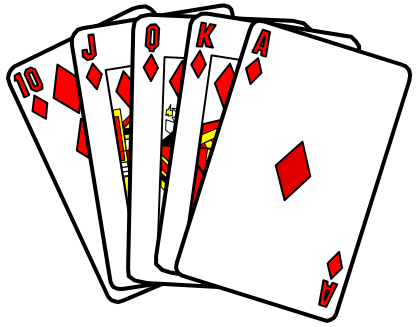
Sortierverfahren 1



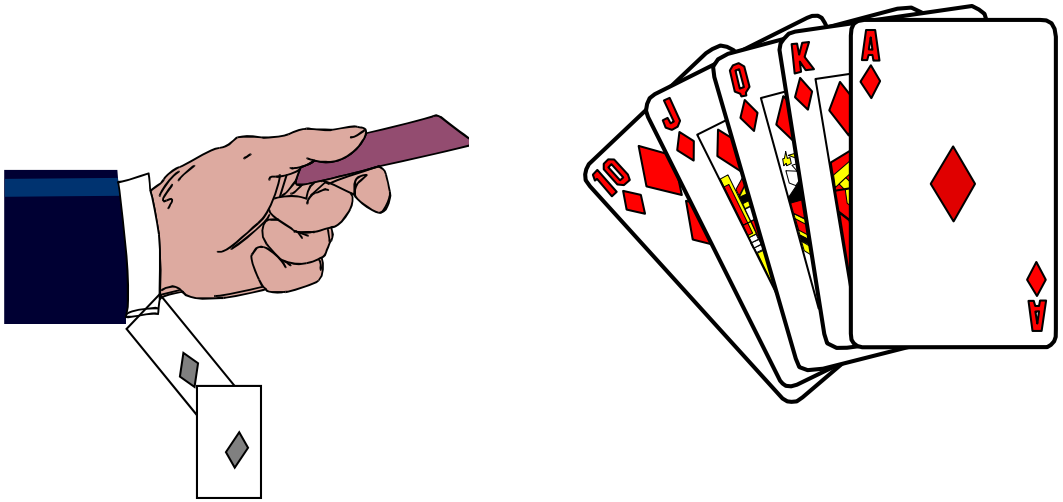
- Sie wissen, warum Sortieren wichtig ist
- Sie können den Aufwand von Sortieralgorithmen bestimmen
- Sie unterscheiden internes und externes Sortieren
- Sie kennen die drei einfachen internen Sortierverfahren:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort

Motivation

- Sortieren als (eigenständige) **Aufgabe**:
  - Wörter 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*

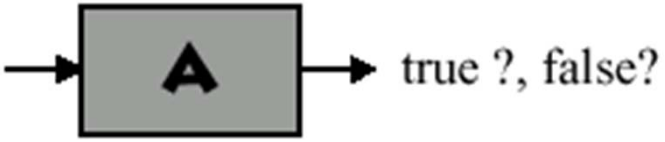
Motivation: sind zwei gleiche Karten im Spiel?

- 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:

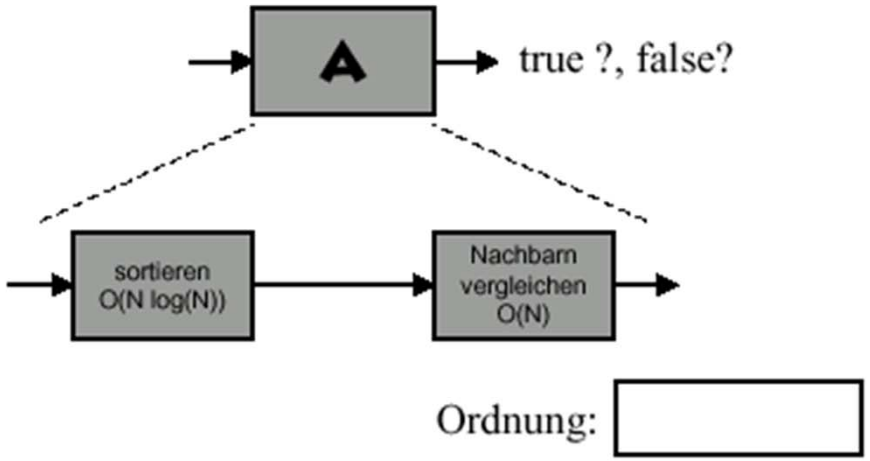



```
public boolean duplicates (Object[] a) {
  for (int i=0 ; i<a.length ; i++)
    for (int j=i+1 ; j<a.length ; j++)
      if (a[i].compareTo(a[j])==0)
        return true;
  return false;
}
```

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 *.



* So einfach ist das natürlich nicht. Die O-Notation stimmt für grosse n. Aber Joe muss ja nicht 1 Mio. Karten vergleichen.

Wie sortiere ich einen Kartenstapel?

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.

Bubble 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.

Selection Sort

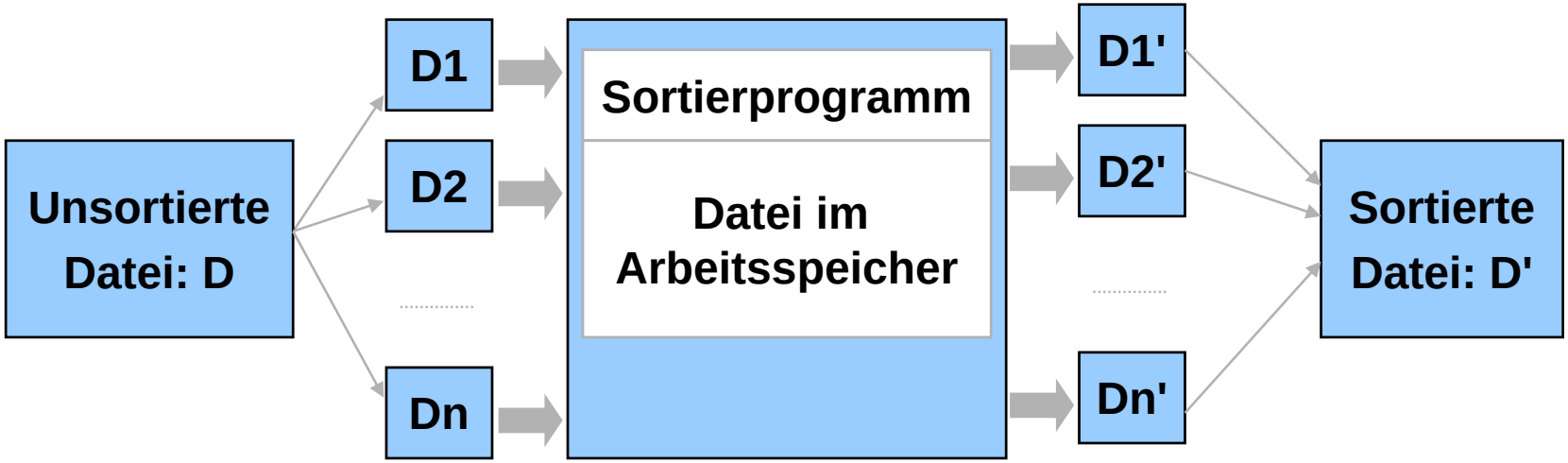
Internes Sortieren

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

- Man spricht dann von einem **internen** Sortiervorgang (**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).

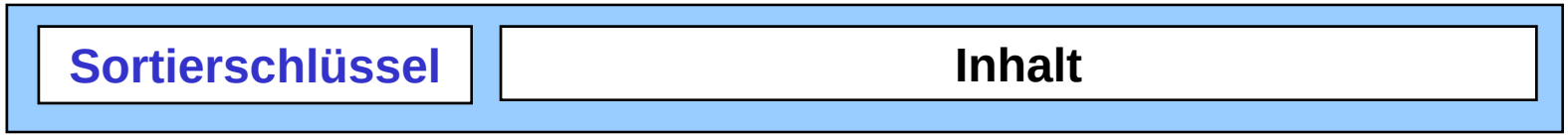
- Dateien werden in Speicher eingelesen, intern sortiert und wieder in Dateien geschrieben.

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

→ **Das Problem des externen Sortierens lässt sich auf das des internen Sortierens zurückführen.**

Sortieren von Datensätzen: der 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  fachbereich;
}
```

Datensätze mit dieser Struktur können nach beliebigen Feldern oder Felderkombinationen sortiert werden, so etwa nach:
[Alter]
[Fachbereich]
[MatrikelNr]
[Name, Vorname, MatrikelNr]

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 vorhanden sein.

Vergleich von Sortierschlüsseln

- 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` Interface

- In Java ist folgendes Interface definiert

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

es sei `a.compareTo(b);`

- <0 falls das Object a kleiner als b
- == 0 falls das Object a gleich b
- >0 falls das Object 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);
      if (i==0) i = firstName.compareTo(s2.firstName);
      if (i==0) i = matrikelNr – s2.matrikelNr;
      return i;
  }
}
```

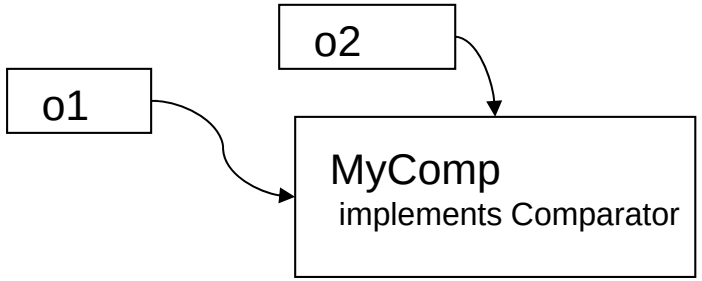
Wiederholung: Das `Comparator` Interface

- 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);
     public boolean equals(Object obj);
}
```

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



Student mit Comparator

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

```
class NameComparator implements
                  Comparator<Student> {
  public int compare(Student s1, Student s2) {
    int i = s1.name.compareTo(s2.name);
    if (i==0)
      i = firstName.compareTo(s2.firstName);
    return i;
  }
}
```

```
class NrComparator implements
                  Comparator<Student> {
  public int compare(Student s1, Student s2) {
    return s1.matrikelNr – s2.matrikelNr;
  }
}
```

```
// List of students
List<Student> list = new ArrayList<Student>();

Collections.sort(list, new NameComparator());
Collections.sort(list, new NrComparator());
```

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: Annahmen

- In den meisten Algorithmen werden Elemente vertauscht; es wird deshalb die Existenz folgender **swap** Methode angenommen.

```
void swap(char[] A, int i, int k){
   char h = A[i]; A[i] = A[k]; A[k] = h;
}
```

Bemerkung: Dieses swap ist natürlich aus verschiedenen Gründen unschön. In C++ würde man schreiben:

```
template <class Element>
void swap(Element& e1, Element& e2){
   Element h = e1; e1 = e2; e2 = h;
}
```

Einfache Sortieralgorithmen

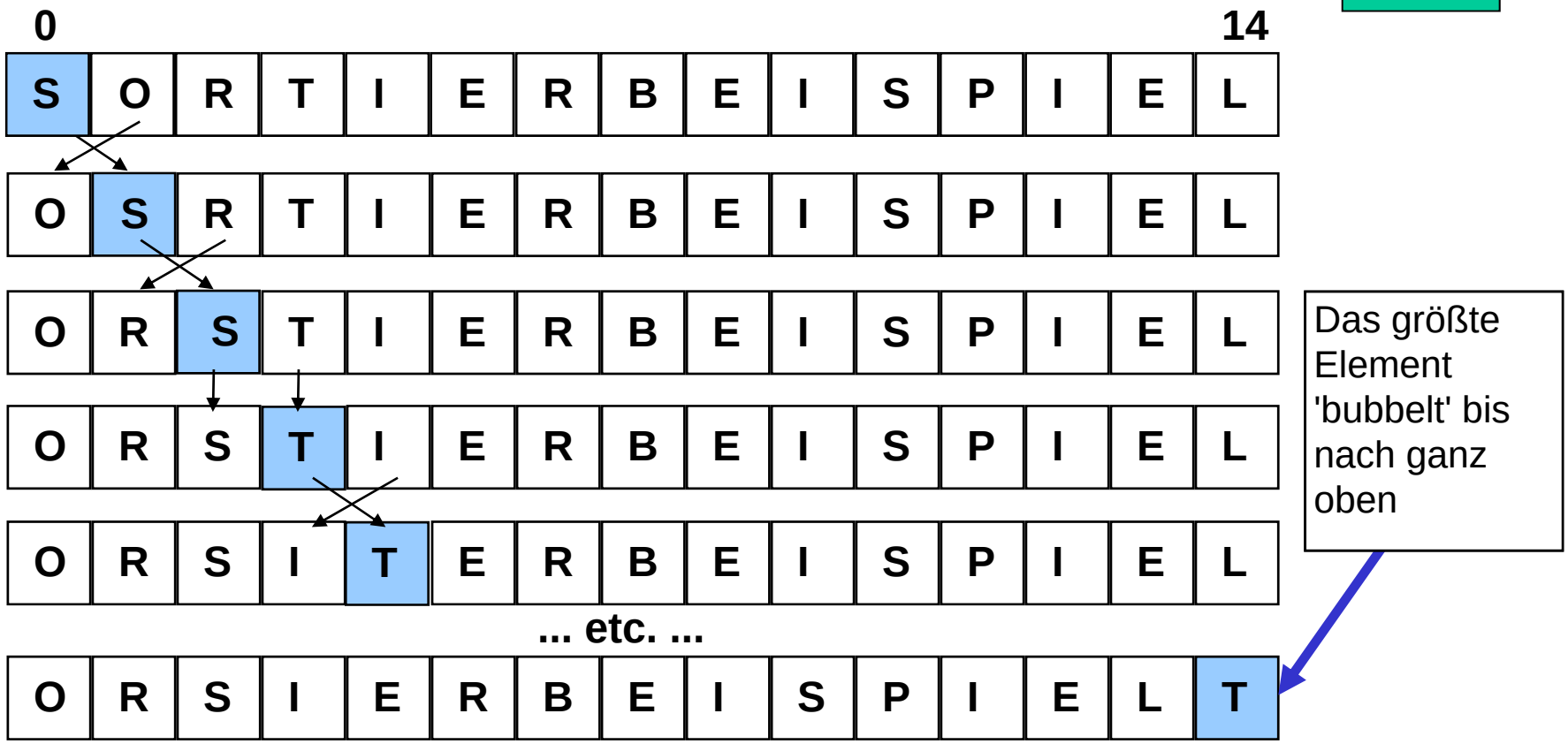
- **Bubble Sort:**
  - Sortieren durch Vertauschen von Nachbarfeldern.
- **Selection Sort:**
  - Sortieren durch Auswählen des jeweils grössten der verbleibenden Elemente und Anhängen an die Reihe der bereits sortierten Elemente.
- **Insertion Sort:**
  - Sortieren durch Einfügen eines beliebigen Elementes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

Bubble-Sort

- 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

Demo



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	Original-Array
O	R	S	I	E	R	B	E	I	S	P	I	E	L	T	nach 1. BubbleUp
O	R	I	E	R	B	E	I	S	P	I	E	L	S	T	nach 2. BubbleUp
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	... etc. ...
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	nach 10. BubbleUp: sortiert!

Bubble-Sort

Die folgende Java-Methode BubbleSort ordnet ein Array der Länge N, indem sie N mal „bubbleUp“ auf den noch ungeordneten Teil des Arrays anwendet.

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

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 alle 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.

```
void BubbleSort2(char[] A){
  for (int k = 0; k < A.length-1; k++){
      boolean noSwap = true;
      for (int i = 0; i < A.length-k-1; i++)
        if ( A[i] > A[i+1]) {
            swap (A, i, i+1);
            noSwap = false;
        }
      if (noSwap) break;
  }
}
```

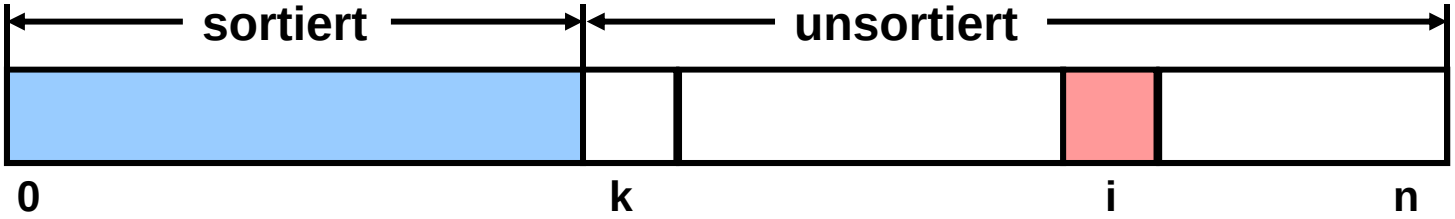
Bubble-Sort: Aufwand

- Wenn n = A.length-1 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 Anweisungen in der inneren Schleife ist, ergibt sich daher als Laufzeit für den worst case:

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

Selection-Sort



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

- Invariante: $\forall i; (i > 0) \wedge (i < k) : a[i-1] \leq a[i]$

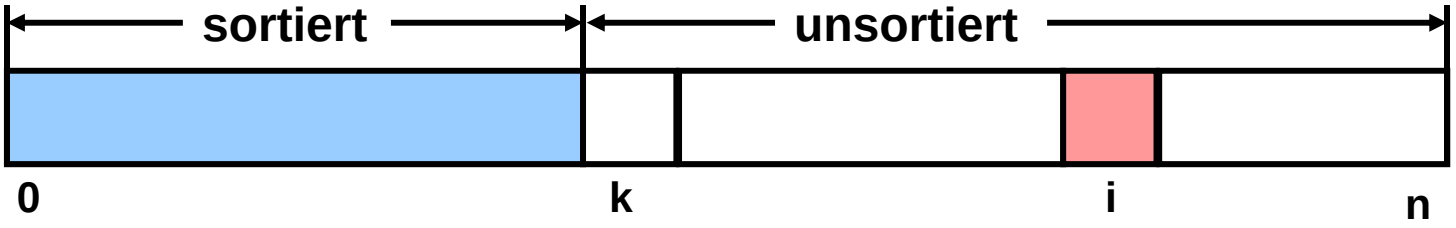
  $\forall i; (i > 0) \wedge (i < k); \forall j;\ j \geq k : a[i] \leq a[j]$

- Am Anfang k = 0

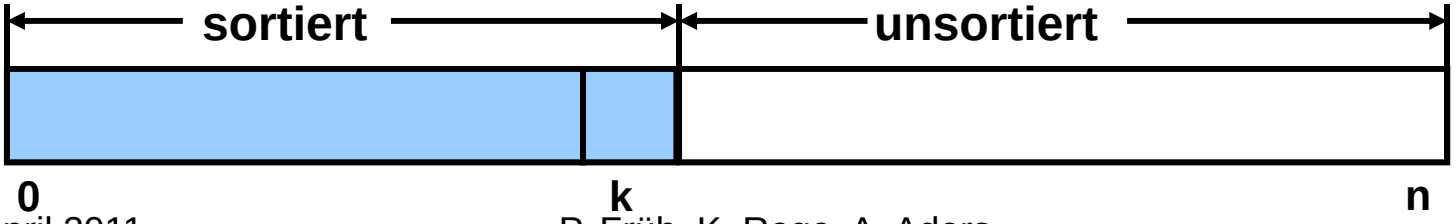
- Frage: welches Element aus der (unsortierten) Restmenge muss ich auswählen, damit ich den sortierten Bereich um 1 vergrössern kann?

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..n sei k die Postion des **ersten** Elements im noch nicht sortierten Bereich und i die Position des **kleinsten** Elementes in diesem Bereich



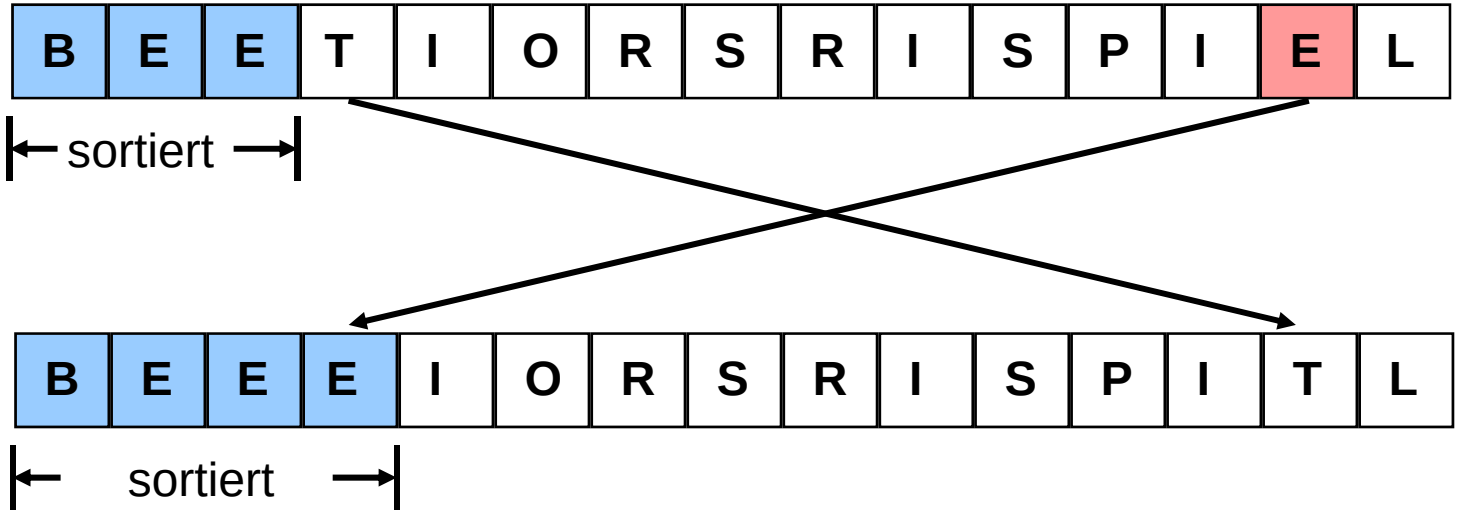
Wenn wir nun A[k] und A[i] vertauschen, dann haben wir den sortierten Bereich um **ein Element vergrößert**.



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:

Demo



Selection-Sort

Java-Methode für den Selection-Sort:

```
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

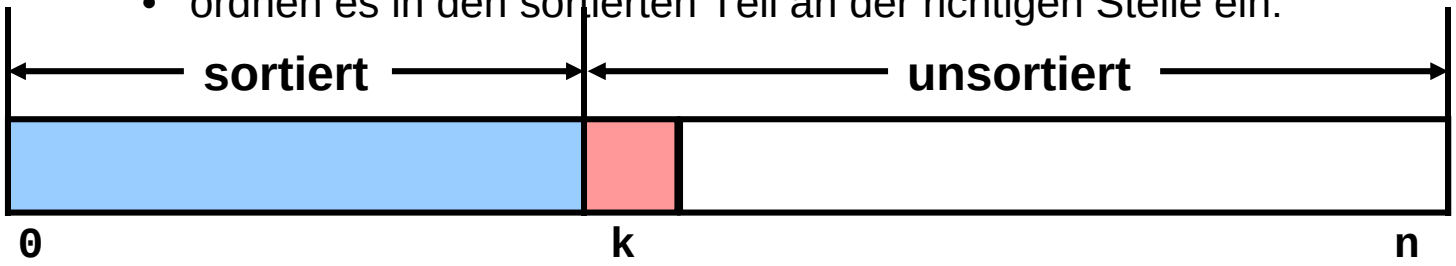
- Vorteil:
  - deutlich weniger Swap-Aufrufe als Bubble Sort.

- Nachteil:
  - Vorsortiertheit kann nicht ausgenutzt werden

Sortieren durch Einsetzen: Insertion-Sort

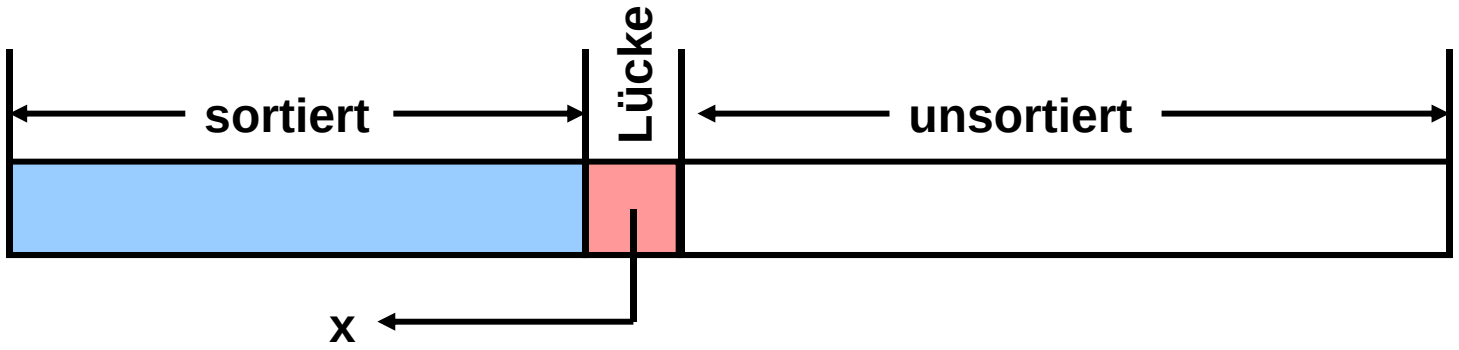
- Analogie zum Sortieren eines Kartenspieles:
  - Eine Karte nach der anderen wird aufgenommen und an der richtigen Stelle in die schon sortierten Karten eingeordnet.

- 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:



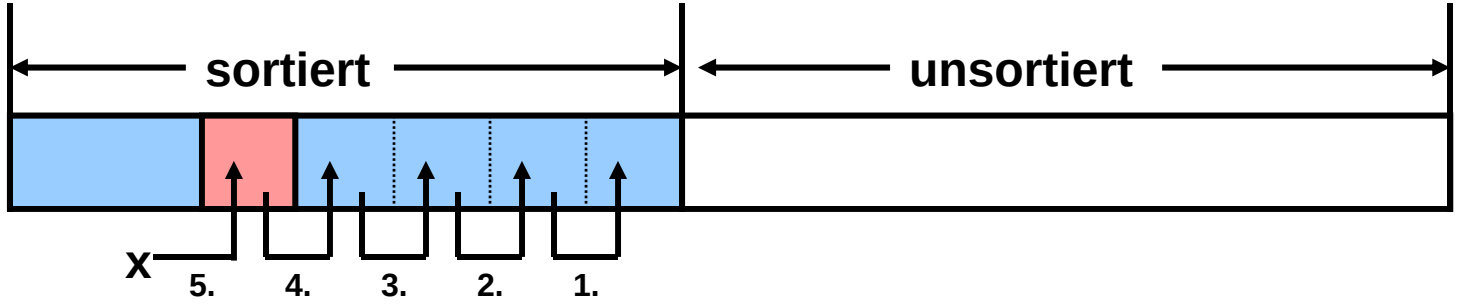
Das Element x=A[k] wird aufgenommen, also aus dem Array herausgenommen.

Insertion-Sort



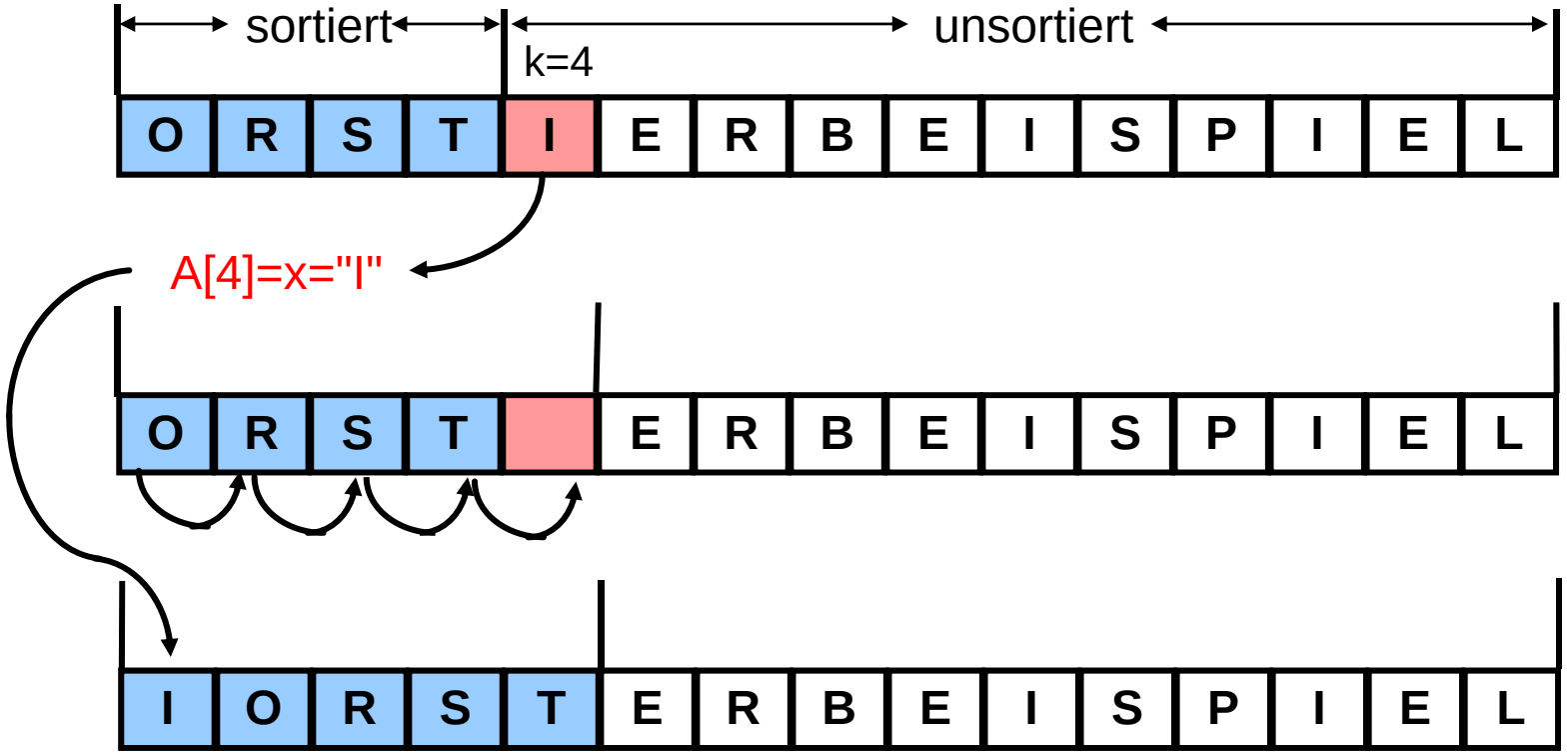
- die entstehende Lücke wird nach links verschoben,
- bis die korrekte Position für das Element x gefunden wurde.
- x wird dort eingeordnet.

Resultat: der sortierte Bereich wurde um eins vergrößert.



Insertion-Sort Beispiel

Demo



Insertion-Sort Algorithmus

```
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

verschiebe Lücke

finde Einfügestelle, verschiebe Elemente

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.

Vergleich Insertion- mit Selection-Sort

- InsertionSort ist besser wenn:
  - Datensätze relativ kurz sind (Aufwand für die zusätzlich erforderlichen Swap-Aufrufe gering)*
  - Daten relativ gut vorsortiert sind.
- SelectionSort ist besser wenn:
  - Datensätze relativ lang sind*
  - Daten völlig unsortiert sind

*) Da in Java nur Zeiger (Objektreferenzen) vertauscht werden, sticht dieses Argument nicht

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