



Sortieren

BubbleSort, SelectionSort,
InsertionSort, QuickSort,
MergeSort



Sortieren



- Wie können wir Elemente eines Arrays sortieren ?
 - Viele Programmierlösungen wiederholen Strategien, die aus dem täglichen Leben bekannt sind:
 - Wie sortiert ein Kartenspieler die Karten in seiner Hand ?
 - **BubbleSort**:
 - Nimm die Karten auf die Hand und vertausche zwei benachbarte Karten, wenn sie in der falschen Reihenfolge sind. Tue das bis die Karten geordnet sind
 - **InsertionSort**
 - Nimm jeweils eine Karte vom Tisch und füge sie an der richtigen Stelle in die Hand ein.
 - **SelectionSort**
 - Suche jeweils die niedrigste Karte von denen, die auf dem Tisch liegen und füge sie rechts außen in die Hand ein
 - **MergeSort**
 - Teile die Karten in zwei Teile. Sortiere die beiden Haufen einzeln und füge sie zusammen, wobei die Sortierung erhalten wird



isSorted



```
private static boolean isSorted(int[] daten, int lo, int hi){
    for(int k=lo; k<hi; k++){
        if (daten[k+1]<daten[k]) return false;
    }
    return true;
}
```

- Wir schreiben eine Invariante
 - *boolean isSorted(int [] a, int lo, int hi)*
- *isSorted(a,lo,hi)* überprüft, ob das Intervall *lo..hi* des Arrays *a* sortiert ist
- Alle Sortieralgorithmen müssen diese Invariante herstellen. Wir überprüfen dies mit einer assertion:

assert isSorted(a, 0, a.length-1) : "Nicht sortiert" ;



swap



- Beim Sortieren darf man kein Element verlieren oder hinzufügen.
 - Mathematisch: Die Elemente eines Arrays dürfen nur permutiert werden.
- Daher lassen wir zur Manipulation der Elemente nur die folgende Prozedur zu:
`void swap(int[] daten, int i, int j)`
 - Mathematisch: *swap* realisiert *Transposition* zweier Arrayelemente
 - Die Transpositionen erzeugen die *symmetrische Gruppe*, daher kann man *jedes Sortieren* mit geeigneten *swaps* erreichen

```
/** swap vertauscht die Elemente a[i] und a[j] */  
private static void swap(int[] daten, int i, int j){  
    int temp=daten[i];  
    daten[i]=daten[j];  
    daten[j]=temp;  
}
```



Sehr naives BubbleSort



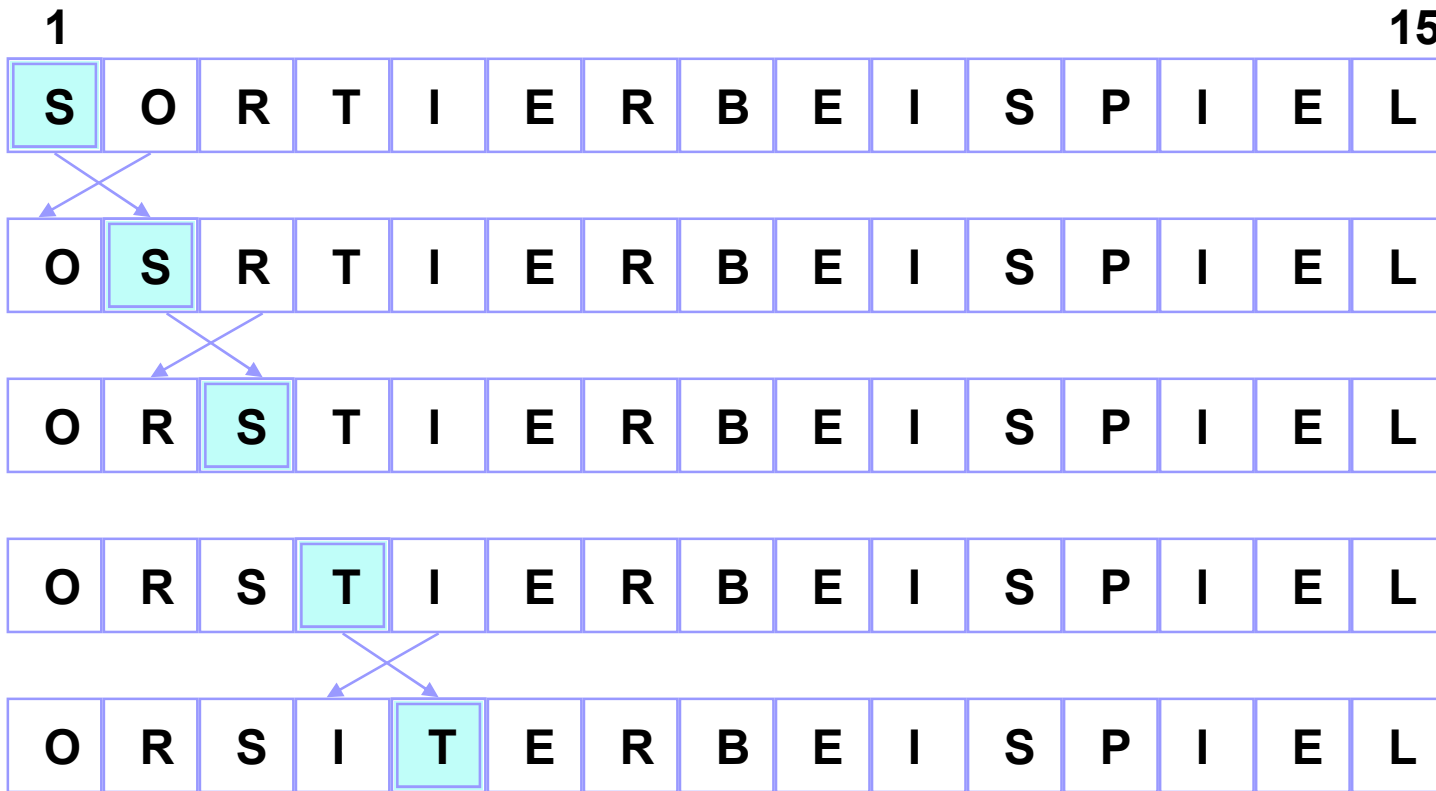
- **BubbleSort** vertauscht immer nur benachbarte Elemente – bis der Array sortiert ist:

```
static void naivesBubbleSort (int [] daten) {  
    while (!isSorted (daten, 0, daten.length-1))  
        for (int i=0; i < daten.length-1; i++) {  
            if (daten[i] > daten[i+1]) swap (daten, i, i+1);  
        }  
    // Nachbedingung - garantiert, dass Array sortiert ist  
    assert isSorted (daten, 0, daten.length-1);  
}
```

- ✓ **Nachbedingung** ist Negation der **while-Bedingung**. Daher klar, dass sie erfüllt ist, wenn **while** terminiert
- ✓ Elemente werden nur mit **swap** manipuliert, daher klar, dass Ergebnis eine **Permutation** des Ausgangsarrays ist
- ✓ **while terminiert**, weil immer weniger Fehlstellungen vorhanden sind

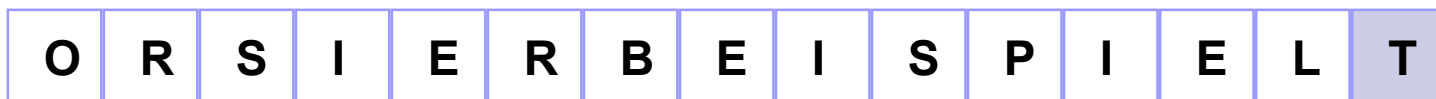


Beispiel: der komplette 1. Durchlauf



Beachte :
Das größte Element
'bubbelt' bis nach ganz oben

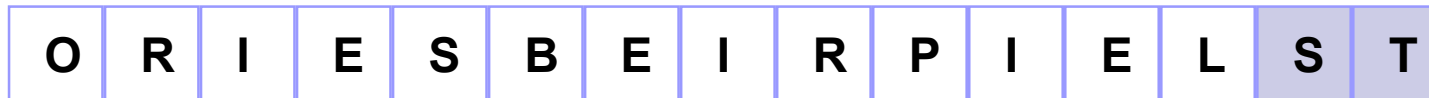
... etc. ...



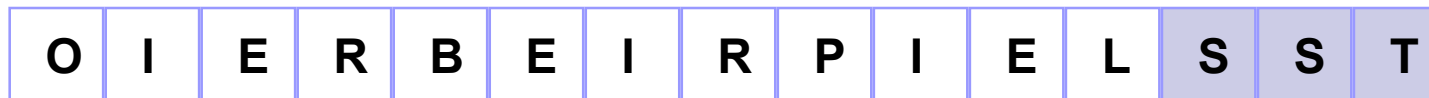


Eine Invariante von BubbleSort

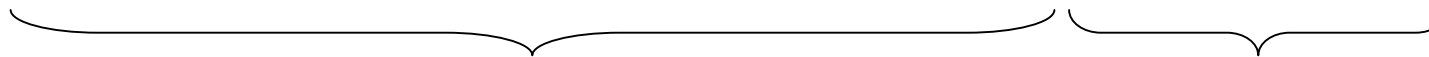
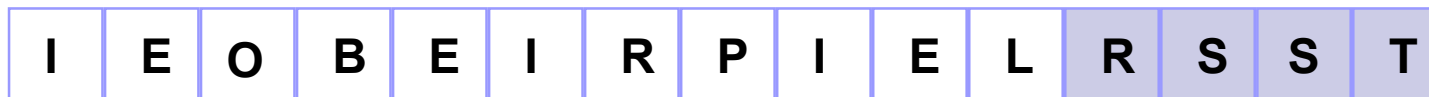
Nach dem 2. Durchlauf



Nach dem 3. Durchlauf



Nach dem 4. Durchlauf



Ungeordnet und
 \leq alle Elemente in
 $a[\text{length}-4..\text{length}-1]$

Geordnet und
 \geq alle Elemente in
 $a[0..\text{length}-4-1]$



Sortierbeispiel: bubbleSort

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
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

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
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

... etc. ...

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
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

nach 10. bubbleUp

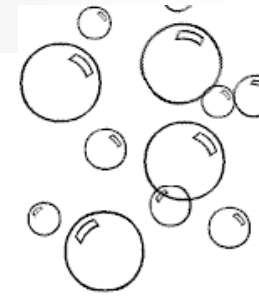
B	E	E	E	I	I	I	L	O	P	R	R	S	S	T
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Sortiert !

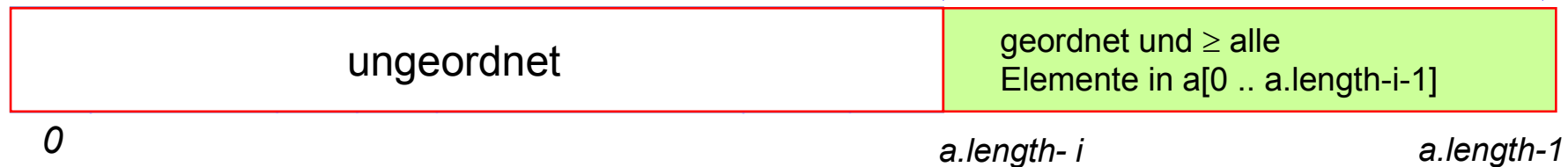
Philipps-Universität Marburg



Invariante



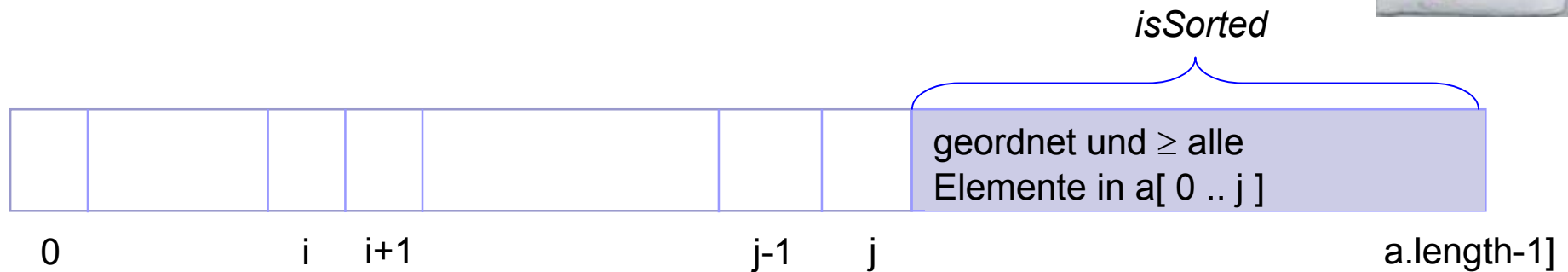
- Wie oft müssen wir durch den Array bubbeln ?
 - Beim ersten Durchlauf durch die Schleife kommt das größte Element ganz nach oben
 - Beim zweiten Durchlauf kommt das zweitgrößte Element an die zweithöchste Stelle, etc.
- Nach dem i -ten Durchlauf sind die obersten i Elemente bereits an der richtigen Stelle
 - Wir müssen maximal $(a.length-1)$ -mal bubbeln
 - Wir brauchen uns nur um das untere – ungeordnete Intervall zu kümmern:



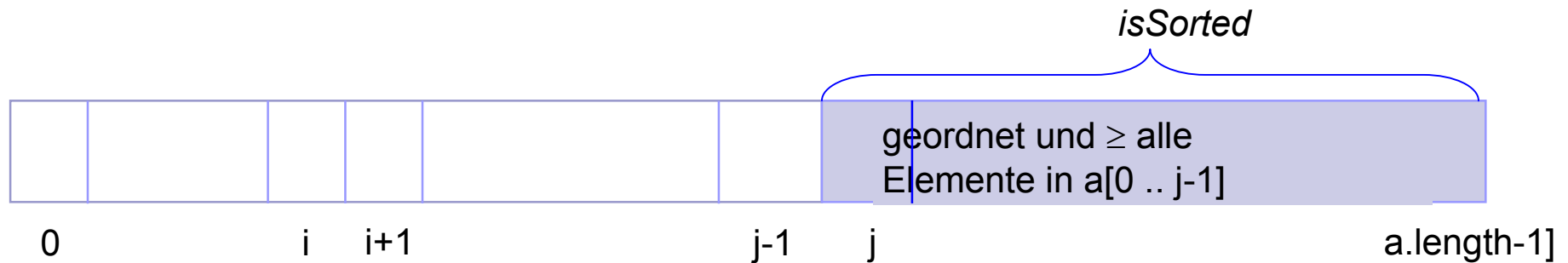
- *Nach dem i -ten Durchlauf sind die obersten i Elemente an der richtigen Position*



BubbleUp – ein Durchlauf



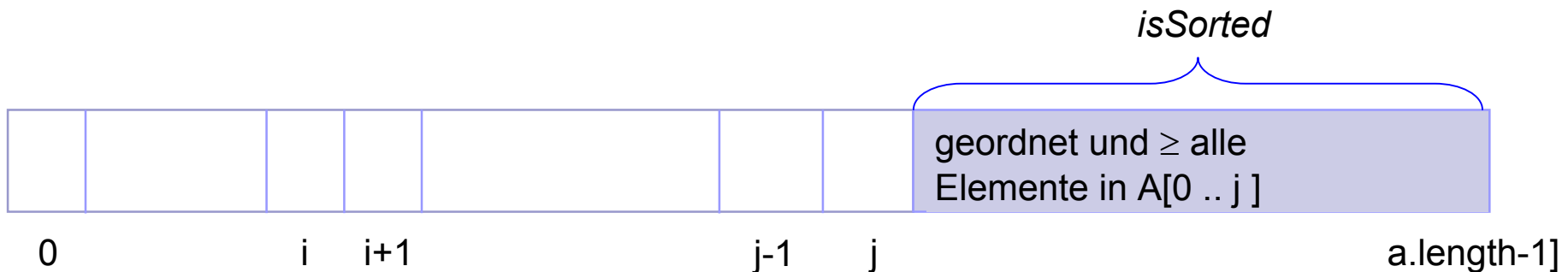
```
for (int i=0; i<j; i++){  
    // Innere Invariante  
    assert allGreaterEq(a, j+1, a.length-1, a[i]);  
    if(a[i]>a[i+1]) swap(a, i, i+1);  
}
```





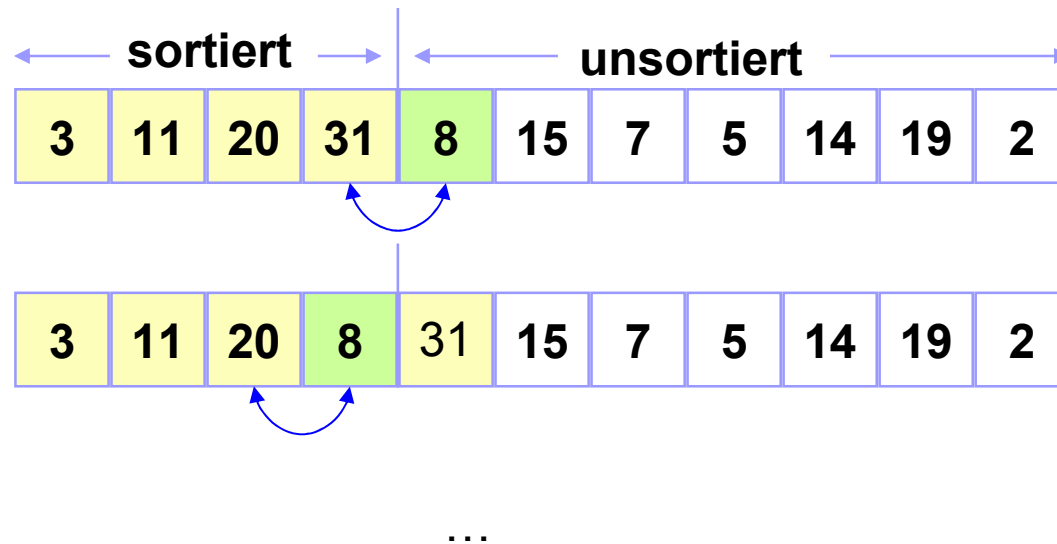
BubbleSort komplett

```
public static void bubbleSort(int[] a){
    for (int j = a.length-1; j>0; j--){
        for (int i=0; i<j; i++){
            // Innere Invariante
            assert allGreaterEq(a, j+1, a.length-1, a[i]);
            if(a[i]>a[i+1]) swap(a, i, i+1);
        }
        // Invariante
        assert isSorted(a, j, a.length-1);
    }
}
```



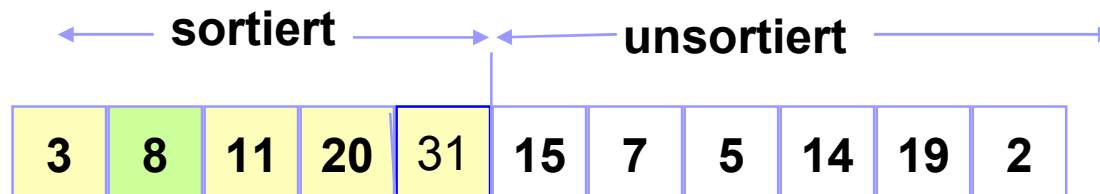


InsertionSort – einsortieren



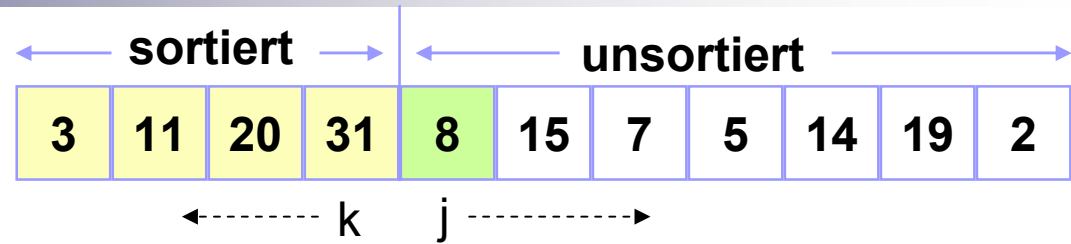
Die 8 wird einsortiert

Dabei werden alle
größeren Elemente
im sortierten Bereich
um eins nach oben
geschoben





InsertionSort



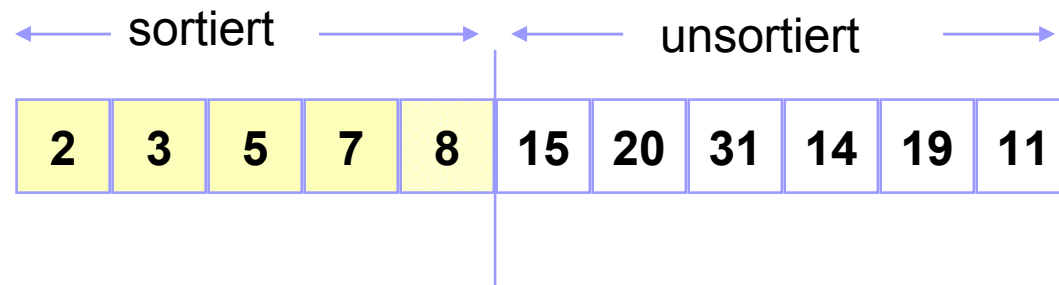
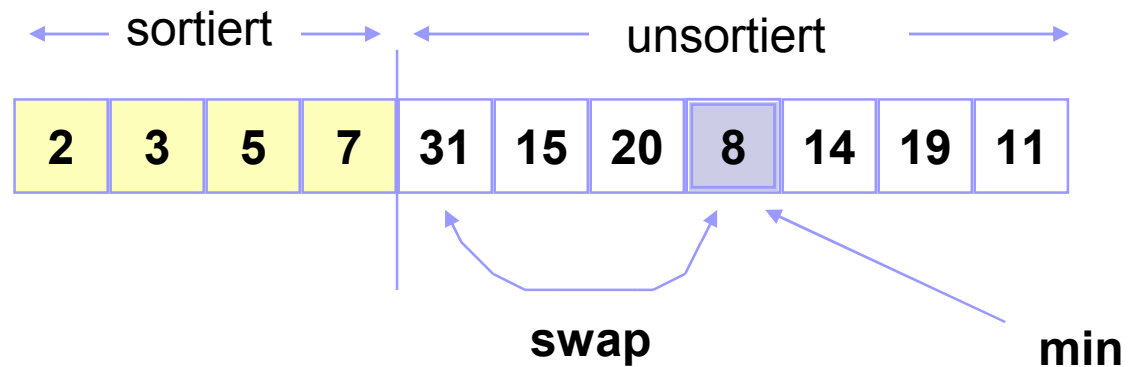
- **Invariante:** der erste Abschnitt ist geordnet
- Das nächste Element wird durch **swappen** eingeordnet
- Dabei werden gleichzeitig die größeren Elemente eins nach oben befördert

```
83 public static void insertionSort(int[] a){
84     for (int j=1; j<a.length; j++){
85         // Invariante
86         assert isSorted(a,0,j-1):j;
87         int nextVal = a[j];
88         int k=j-1;
89         // a[j] bubbelt nach unten an die richtige Position
90         while(k >= 0 && a[k] > nextVal) {
91             swap(a,k,k+1);
92             k--;
93         }
94     } // end for
95     // Nachbedingung
96     assert isSorted(a);
97 }
```



SelectionSort

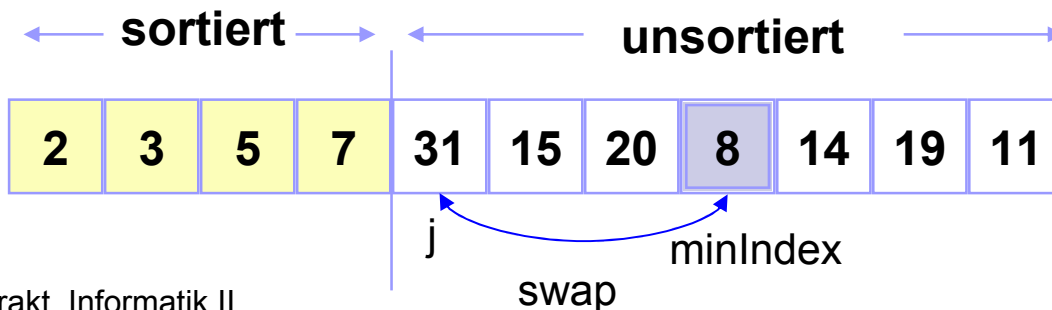
- Der untere Teil sei bereits geordnet und \leq jedem Element im ungeordneten Bereich
- Suche das nächstgrößere Element
 - das ist das kleinste Element des unsortierten Bereichs
- Befördere es durch ein *swap* an die richtige Stelle





SelectionSort

```
103 public static void selectionSort(int[] a){
104     for (int j=0; j<a.length-1; j++){
105         // Invariante
106         assert isSorted(a,0,j-1);
107         int minIndex = j;
108         for(int k=minIndex+1; k<a.length;k++)
109             if(a[k]<a[minIndex]) minIndex=k;
110         swap(a,j,minIndex);
111     }// end for
112     // Nachbedingung
113     | assert isSorted(a);
114 }
```





Sortierbeispiel: SelectionSort

1	S	O	R	T	I	E	R	B	E	I	S	P	I	E	L	15	Original-Array
	B	O	R	T	I	E	R	S	E	I	S	P	I	E	L		nach 1. Swap
	B	E	R	T	I	O	R	S	E	I	S	P	I	E	L		nach 2. Swap
	B	E	E	T	I	O	R	S	R	I	S	P	I	E	L		
	B	E	E	E	I	O	R	S	R	I	S	P	I	T	L		
	B	E	E	E	I	O	R	S	R	I	S	P	I	T	L		
	B	E	E	E	I	I	R	S	R	O	S	P	I	T	L		

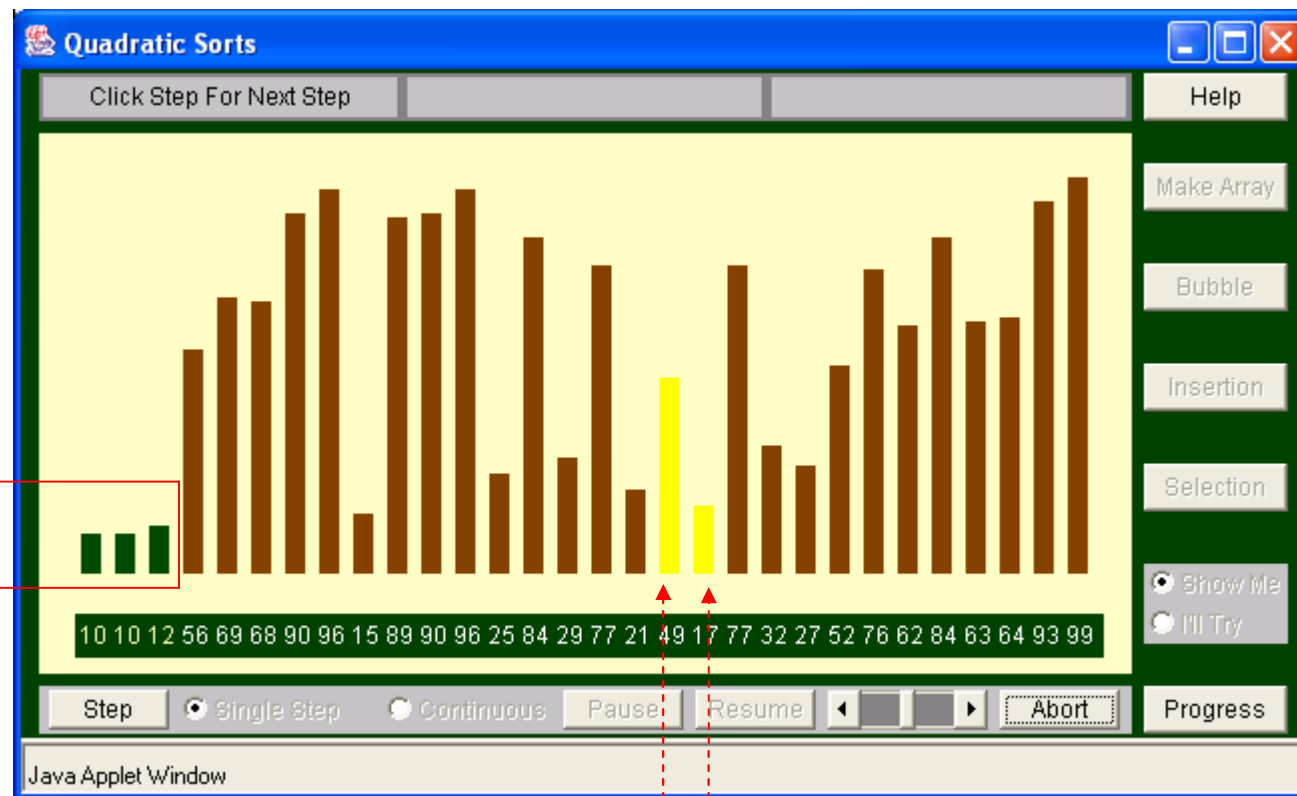
... etc. ...

	B	E	E	E	I	I	I	L	O	P	R	R	S	S	T		nach 14. Swap
																	Sortiert !



Animationen

- Schöne Animationen grundlegender Algorithmen befinden sich auf
 - <http://nova.umuc.edu/~jarc/idsv/>



Bereits
sortiert
und an der
endgültigen
Position

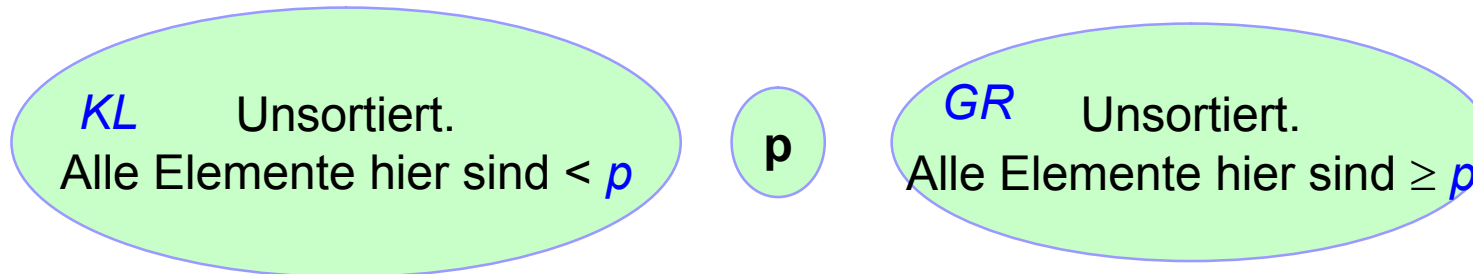
Vertauschung notwendig



QuickSort

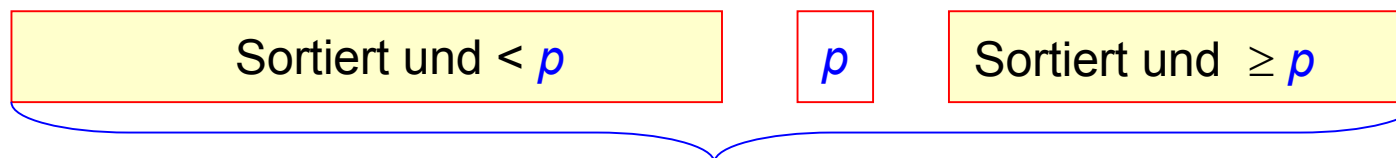


- QuickSort ist ein *Divide-and-Conquer*-Algorithmus
 - Greife ein beliebiges Element p (**pivot**) aus dem zu sortierenden Haufen
 - Zerlege ("partitioniere") den Rest in
 - KL : die Elemente $< p$, und $\{p\}$ und GR : die Elemente $\geq p$ (ohne p)



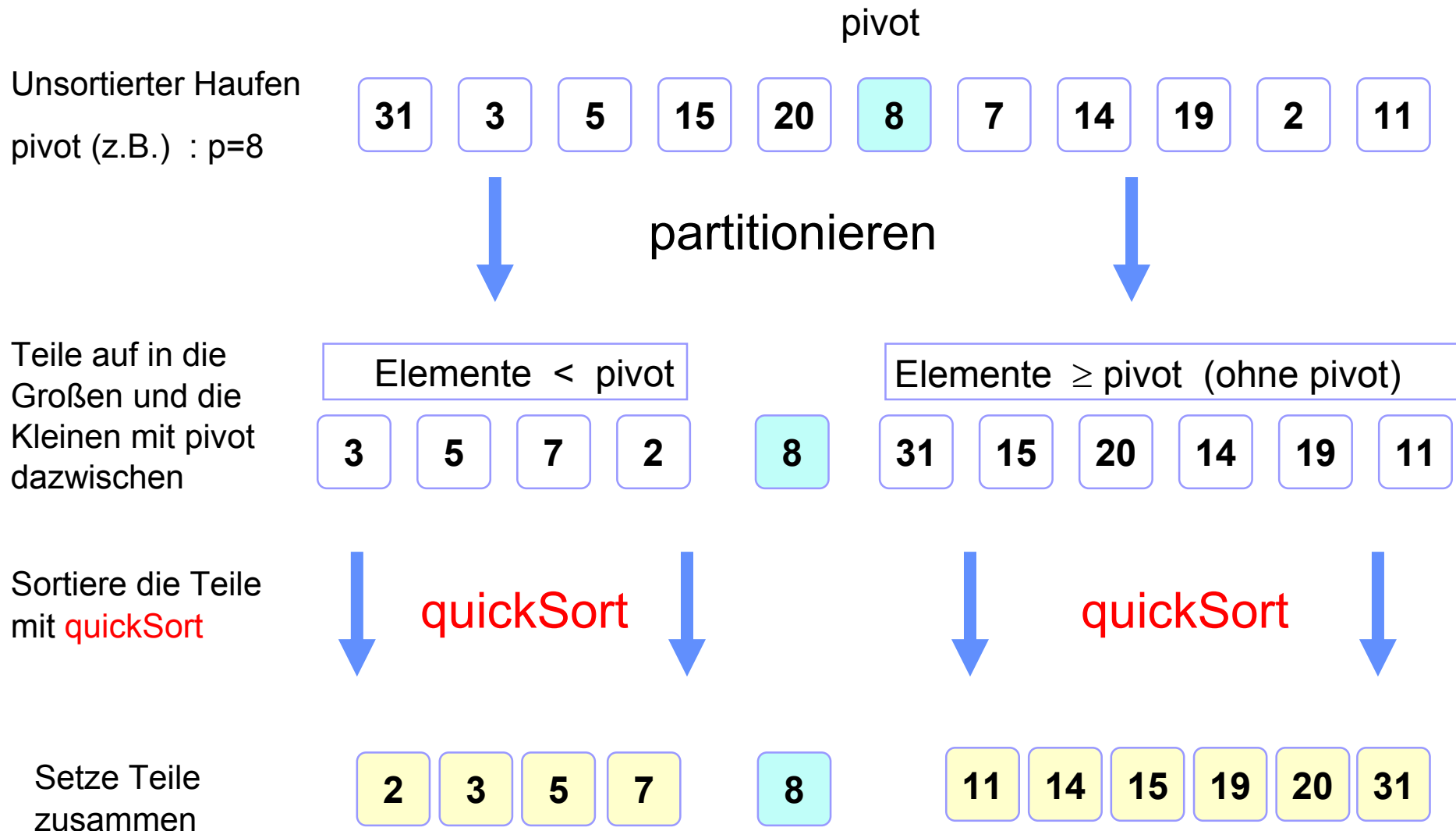
- Sortiere KL (mit QuickSort) , Sortiere GR (mit QuickSort), dann setze die sortierten Teile zusammen:

$$KL + \{p\} + GR.$$





quickSort – schematisches Beispiel





QuickSort

- Verallgemeinern:

`qSort` sortiert ein *Slice*.

`qSort`:

- Partitioniere und gebe endgültige Position des Pivots zurück
- Sortiere untere Hälfte
- Sortiere obere Hälfte

```
3 public static void quickSort(int[] a){
4     qSort(a,0,a.length-1);
5 }
6
7 /** qSort kann ein beliebiges
8  * slice a[lo .. hi] sortieren
9  */
10 public static void qSort(int[] a, int lo, int hi){
11     if (lo < hi){
12         int q = partition(a,lo,hi); // Partitionieren
13         qSort(a,lo,q-1); // Unteren Teil sortieren
14         qSort(a,q+1,hi); // Oberen Teil sortieren
15     }
16 }
```



Partitionieren

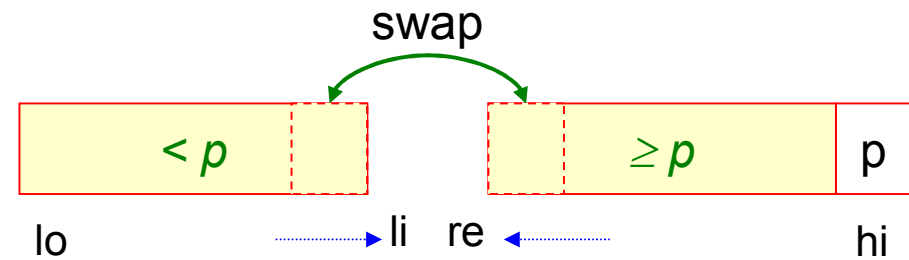
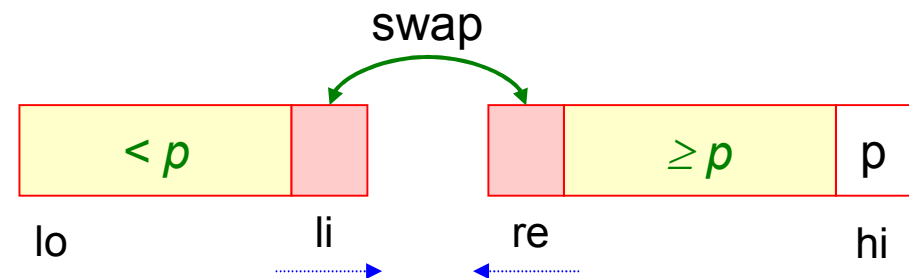
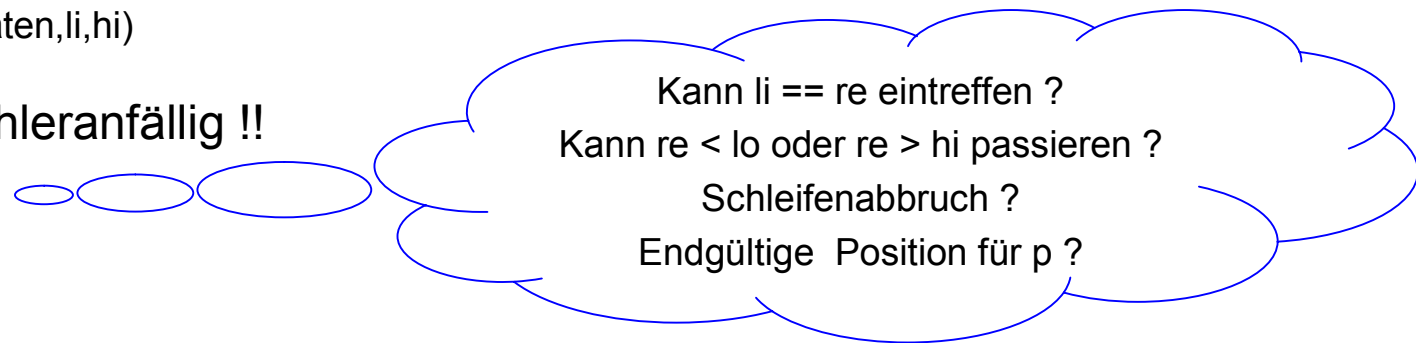
■ Idee einfach

- Bringe Pivot an rechtes Ende
 - `swap(daten,pivIndex,hi)`

- Schleife:
 - Schiebe Index `li` nach rechts bis `daten[li] ≥ pivot`
 - Schiebe Index `re` nach links bis `daten[re] < pivot`
 - Falls `li < re` vertausche `swap(daten,li,re)`

- Setze Pivot in die Mitte
 - `swap(daten,li,hi)`

■ Ausführung fehleranfällig !!





Partitionieren in Java



```
private static int partition(int[] a, int lo, int hi) {
    int pivIndex = (lo+hi)/2;    // Wähle Pivot-Index
    swap(a, pivIndex, hi);      // Pivot ans rechte Ende
    int pivot = a[hi];          // Wert des Pivots

    int li=lo;
    int re = hi;

    while(li < re) {
        // Invariante: a[lo...li-1] < pivot <= a[re...hi]
        assert allLess(a, lo, li-1, pivot);
        assert allGreaterEq(a, re, hi, pivot);
        while(a[li] < pivot) li++;
        while(a[re] >= pivot && re>li) re--;
        if(li < re) swap(a, li, re);
    }
    swap(a, li, hi);
    // Nachbedingung
    assert allLess(a, lo, li-1, a[li]);
    assert allGreaterEq(a, li+1, hi, a[li]);
    return li;
}
```

Überlegen Sie sich:

re=hi-1;
wäre falsch

warum muss man hier
re>li testen ?

warum nicht
swap(a, re, hi) ?



Einfachere Partitionierung

- Idee

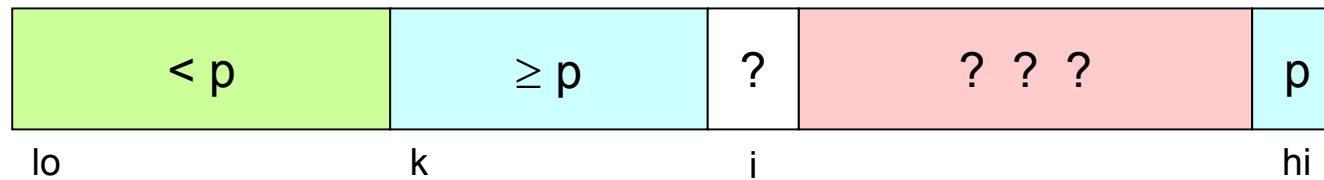
- Pivot ans rechte Ende bringen:

`swap(a,pivIndex,hi)`


- `a[lo .. hi]` aufteilen

`a[lo .. k-1] < pivot`

`a[k .. hi] ≥ pivot`



- beginne mit `k=lo`
- für `i = lo` bis `hi - 1`:

- wenn `a[i] < p`  \Rightarrow `swap(a,i,k); k++; i++;`

- wenn `a[i] ≥ p`  \Rightarrow `i++;`

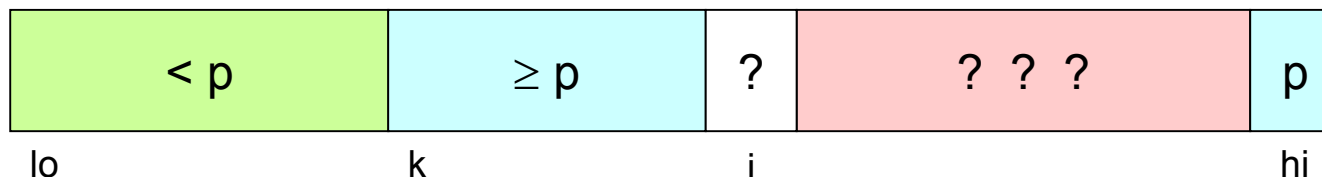
- `a[hi]` mit `a[k]` vertauschen.



Alternative Methode der Partitionierung

- Programmier-technisch einfacher
- Ohne *assertions* nochmal kürzer

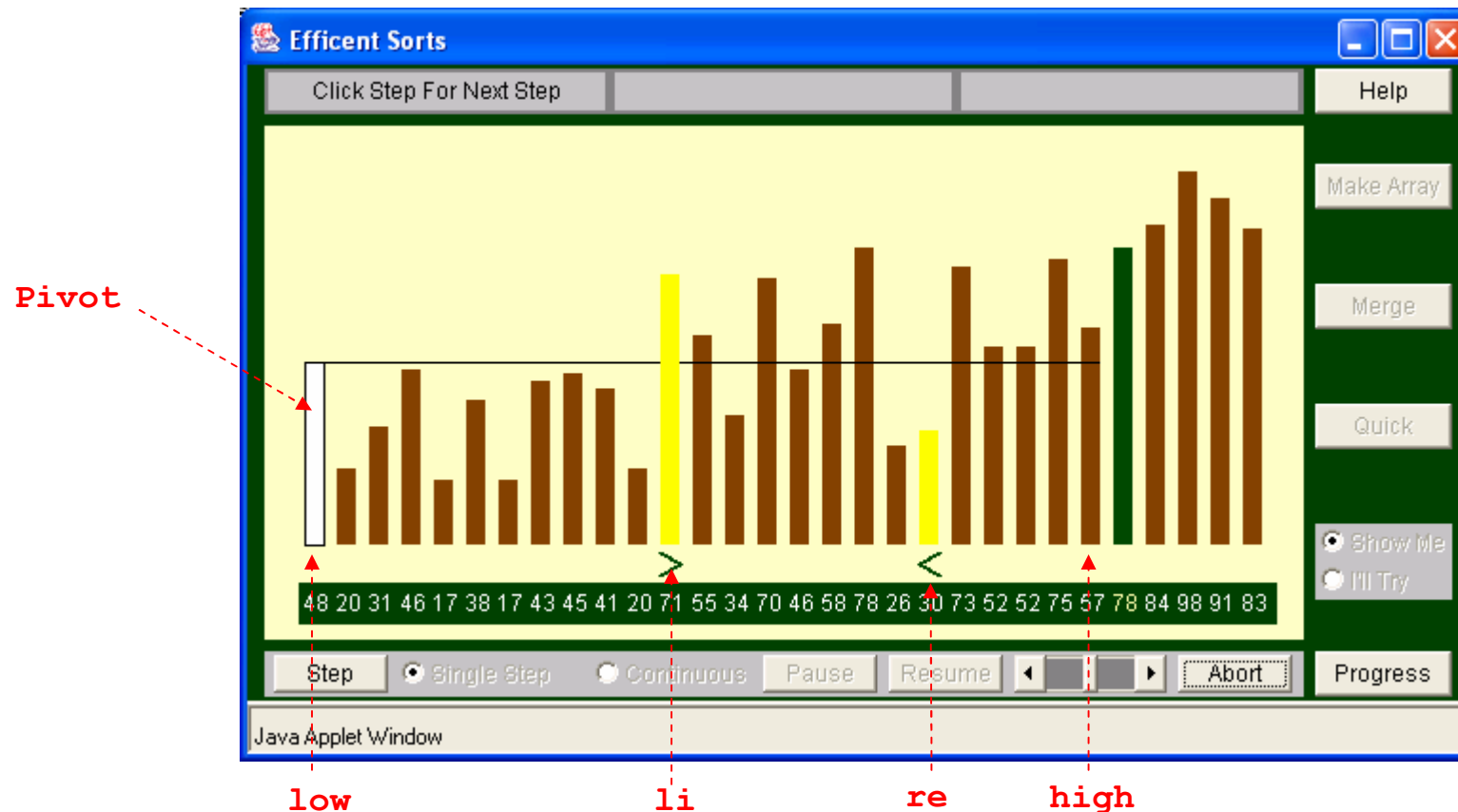
```
private static int altPartition(int[] a, int lo, int hi) {
    int pivIndex = (lo+hi)/2;        // Waehle Pivot-Index
    swap(a, pivIndex, hi);          // Pivot ans rechte Ende
    int pivot = a[hi];
    int k=lo;
    for(int i=lo; i<hi; i++){
        // Invariante: a[lo .. k-1] << pivot <<= a[k .. i-1]
        assert allLess(a, lo, k-1, pivot);
        assert allGreaterEq(a, k, i-1, pivot);
        if(a[i] < pivot) {swap(a, i, k); k++;}
    }
    swap(a, k, hi);
    // Nachbedingung: a[lo..k-1] << a[k] <<= a[k+1..hi]
    assert allLess(a, lo, k-1, a[k]);
    assert allGreaterEq(a, k+1, hi, a[k]);
    return k;
}
```





Animation

- Auf
 - <http://nova.umuc.edu/~jarc/idsv/>
- unter der Rubrik
 - Efficient Sorts
- können Sie QuickSort animieren und ausprobieren





MergeSort



- Divide
 - Teile den Array in zwei etwa gleich große Abschnitte

- Sort
 - Sortieren den linken Abschnitt
 - Sortiere den rechten Abschnitt

- Merge
 - Füge die Abschnitte unter Beibehaltung der Ordnung zusammen

```
public static void mergeSort(int[] a){
    mSort(a,0,a.length-1);
}

public static void mSort(int[] a, int lo, int hi){
    if (hi-lo >= 1){
        int mid = (hi+lo+1)/2;
        mSort(a,lo,mid-1);
        mSort(a,mid,hi);
        merge(a,lo,mid,hi);
    }
}
```

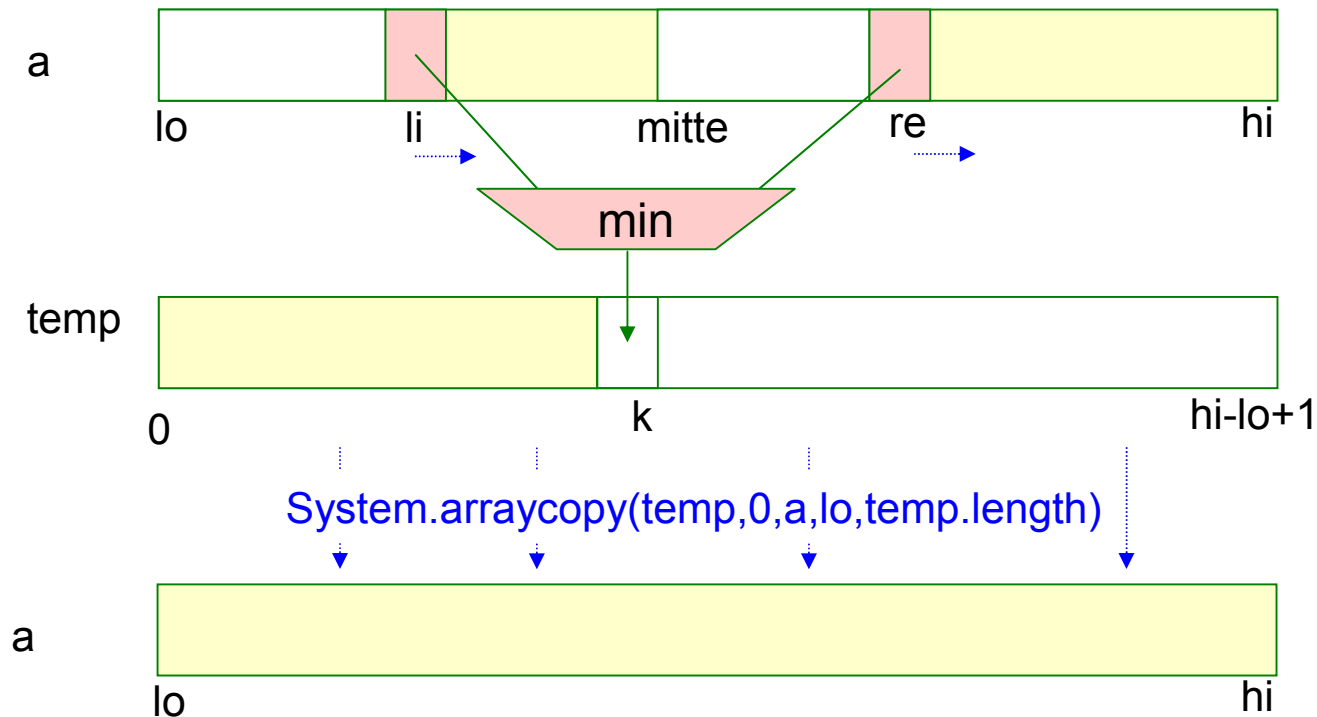
Linker Abschnitt: a[lo..mitte-1]
Rechter Abschnitt: a[mitte..hi]



Merge – die Idee



- $a[lo..mitte-1]$ und $a[mitte..hi]$ sind sortiert
- Mische sie unter Beibehaltung der Reihenfolge in einen temporären Array `temp`
- Kopiere `temp[0..hi-lo+1]` zurück in $a[lo..hi]$



Vorsicht: Für das Kopieren wird **nicht** `swap` benutzt.
Warum ist der sortierte Array eine Permutation des ursprünglichen ?



Merge – der Code

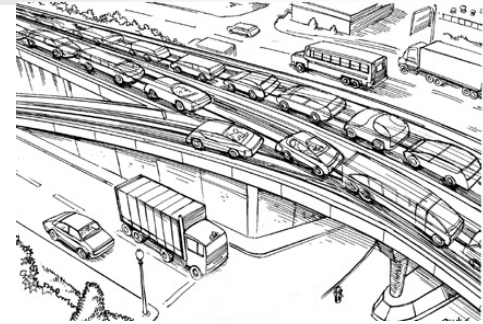
```
private static void merge(int[] a, int lo, int mid, int hi) {
    // Precondition
    assert lo < mid && mid <= hi;
    assert isSorted(a, lo, mid-1) & isSorted(a, mid, hi);
    // Sortiere Daten in einen Hilfsarray;
    int[] temp = new int[hi-lo+1];
    for(int k=0, li=lo, re=mid; k<temp.length; k++)
        if((li < mid) && (re>hi || a[li] < a[re])) //links gewinnt
            temp[k]=a[li++];
        else temp[k]=a[re++];
    System.arraycopy(temp, 0, a, lo, temp.length);
}
```

Kurze Auswertung verhindert Bereichsüberschreitung !!!

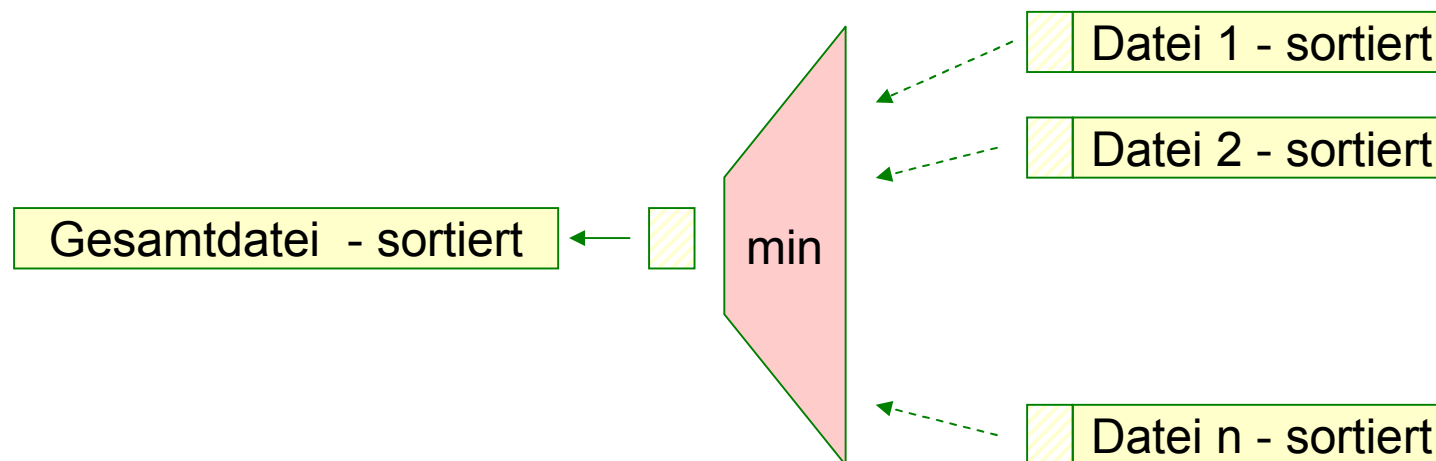
`System.arraycopy(a,i,b,j,len)` kopiert `a[i..i+len]` nach `b[j..j+len]`



Externes Sortieren



- MergeSort eignet sich zum Sortieren großer Datenmengen, die evtl. nicht in den Hauptspeicher passen
- Zerlege die Daten in mehrere **Dateien**
- Sortiere die Dateien einzeln
- **Merge:**
 - Wähle unter den ersten Elementen der Dateien das Minimum
 - Entferne dieses und füge es an die Gesamtdatei an.





Animation

- Wieder von
 - <http://nova.umuc.edu/~jarc/idsv/>
- die Animation von mergeSort

Array temp

$temp[k] = daten[re]$

jeweils für sich sortiert

Noch unsortierte Daten

li re $daten[re] < daten[li]$



Verallgemeinerung

- Alle Sortialgorithmen funktionieren nicht nur auf Arrays von Zahlen oder Strings, sondern
 - (begrifflich) auf geordneten Mengen
 - (technisch) auf Datentypen die `Comparable` implementieren

- `swap` muss re-
implementiert werden
- kann mit dem alten `swap`
koexistieren
- In der Klasse koexistieren

void `swap`(`int`[], int, int)

void `swap`(`Comparable`[], int, int)

```
249 /** Allgemeines SelectionSort */
250 /** Neue (zusätzliche) Methode swap */
251 private static void swap(Comparable[] daten, int i, int j){
252     Comparable temp=daten[i];
253     daten[i]=daten[j];
254     daten[j]=temp;
255 }
256 public static void selectionSort(Comparable[] daten){
257     for(int j=0; j<daten.length-1; j++){
258         // Invariante - daten[0..j-1] ist sortiert:
259         int minIndex = j;
260         // Suche das nächstgrößere Element
261         for (int k=minIndex+1; k<daten.length; k++)
262             if (daten[k].compareTo(daten[minIndex])<0)
263                 minIndex=k;
264         // Bringe es an die richtige Stelle
265         swap(daten,minIndex,j);
266     }
267 }
```