



# Sortieren

bubbleSort, selectionSort, insertionSort,  
quickSort, mergeSort, sortieren  
vergleichbarer (comparable) Daten



## 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,l,h)` 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 die Transposition zweier Arrayelemente
  - Die Transpositionen erzeugen die symmetrische Gruppe, daher kann man sortieren immer 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;
}
```

changed



# Sehr naives BubbleSort



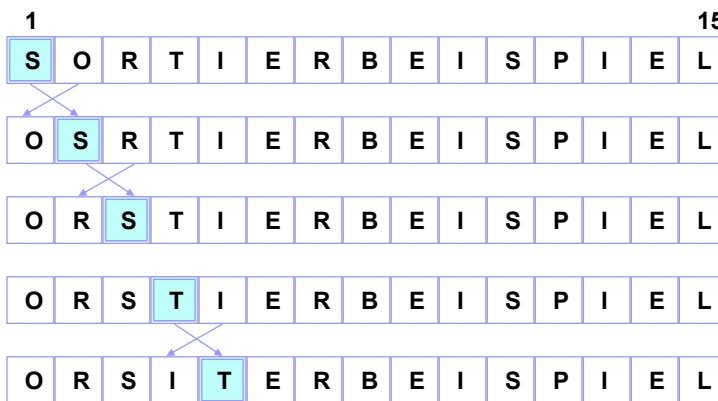
- Bubblesort vertauscht immer nur benachbarte Elemente – bis der Array sortiert ist:

```
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



# Eine Invariante von bubbleSort

Nach dem 2. Durchlauf

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

Nach dem 3. Durchlauf

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

Nach dem 4. Durchlauf

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

Ungeordnet und  $\geq$  alle Elemente in  $a[0..length-4-1]$

Geordnet und  $\geq$  Alle Elemente in  $a[0..length-4-1]$



# 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

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

nach 10. bubbleUp

Sortiert!



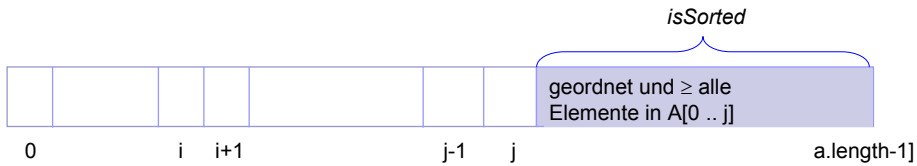
... etc. ...



# Invariante



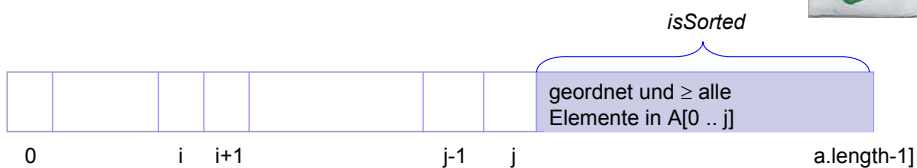
- Wie oft müssen wir durch den Array bubbeln ?
  - Bei ersten Durchlauf durch die Schleife kommt das größte Element ganz nach oben
  - Beim zweiten Durchlauf kommt das zweitgrößte Element an die zweithöchste Stelle, etc.
- Nach dem  $i$ -ten Durchlauf sind die obersten  $i$  Elemente bereits an der richtigen Stelle
  - Wir müssen maximal  $a.length-2$  mal bubbeln
  - Wir brauchen uns nur um das untere – ungeordnete Intervall zu kümmern:



- Nach dem  $i+1$ -ten Durchlauf sind die obersten  $i+1$  Elemente an der richtigen Position



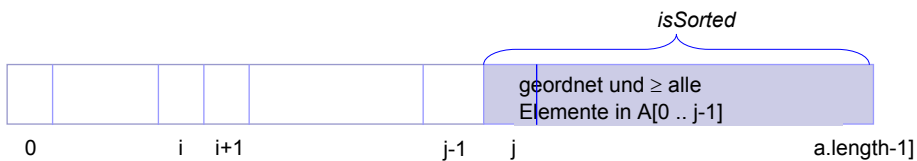
# BubbleUp – ein Durchlauf



```

for(int i=0; i< j;i++){
    if(daten[i]>daten[i+1]) swap(daten, i, i+1);
}

```



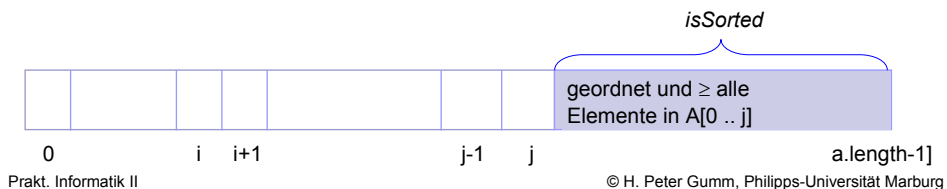


# BubbleSort komplett

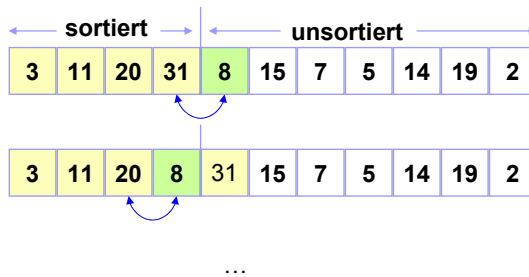
```

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

```

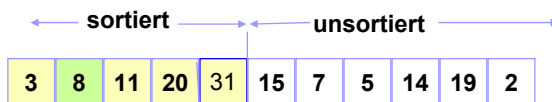


# InsertionSort – einsortieren



Die 8 wird einsortiert

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





# InsertionSort

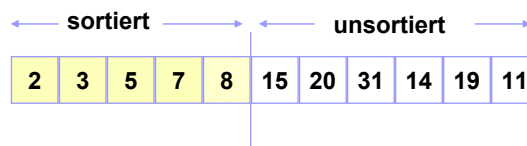
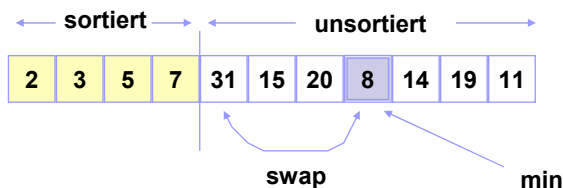
- Invariante: der erste Abschnitt ist geordnet
- Das nächste Element wird durch **swappen** eingeordnet
- Dabei werden gleichzeitig die größeren Elemente eins nach oben befördert

```
static void insertionSort(int[] daten) {
    for(int j=0; j<daten.length-1; j++){
        // Invariante
        assert isSorted(daten, 0, j);
        //Das nächste Element
        int next=daten[j+1];
        int k=j;
        // wird einsortiert
        while(k >= 0 && next < daten[k]){
            swap(daten, k, k+1);
            k--;
        }
        // Nachbedingung
        assert isSorted(daten, 0, daten.length-1);
    }
}
```



# Selectionsort

- Der untere Teil sei bereits geordnet und  $\leq$  jedes Element im ungeordneten Bereich
- Suche das nächstgrößere Element
- Befördere es durch 1 Swap an die richtige Stelle



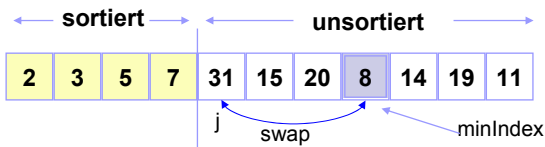


# Selectionsort

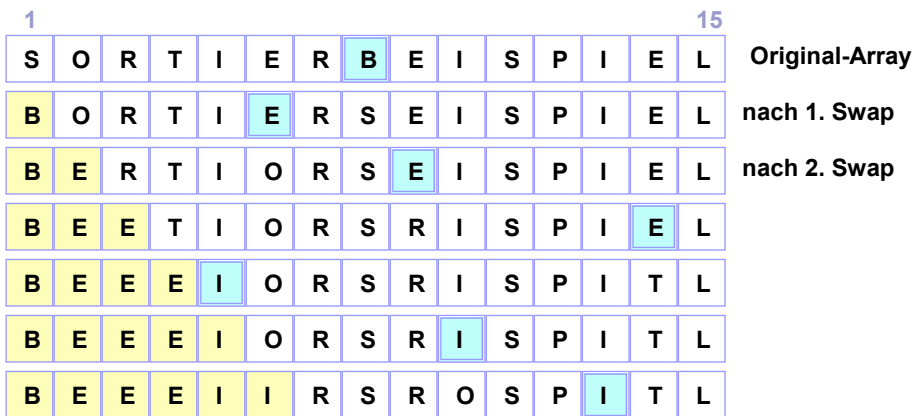
```

static void selectionSort(int[] daten){
    for(int j=0; j<daten.length-1; j++){
        //Invariante
        assert isSorted(daten, 0, j-1);
        int minIndex = j;
        // Suche das nächstgrößere Element
        for (int k=minIndex+1; k<daten.length; k++)
            if (daten[k]<daten[minIndex])minIndex=k;
        //Bringe es an die richtige Stelle
        swap(daten, minIndex, j);
    }
    // Nachbedingung
    assert isSorted(daten, 0, daten.length-1);
}

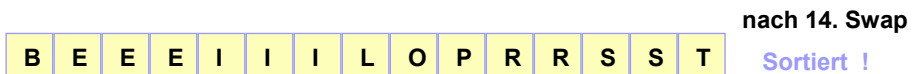
```



# Sortierbeispiel: Selectionsort



... etc. ...





# Animationen

- Schöne Animationen grundlegender Algorithmen befinden sich auf
  - <http://student.seas.gwu.edu/~idsv/idsv.html>

Bereits sortiert und an der endgültigen Position

Vertauschung notwendig

Quadratic Sorts

Click Step For Next Step

Help

Make Array

Bubble

Insertion

Selection

Show Me

Quit

Step

Single Step

Continuous

Pause

Resume

Abort

Progress

Java Apple Window

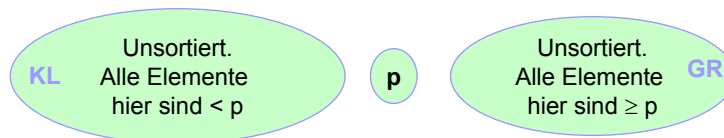
10 10 12 56 69 68 90 96 15 89 90 96 25 84 29 77 21 49 17 77 32 27 52 76 62 84 63 64 93 98



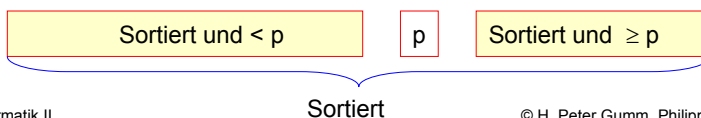
# QuickSort



- 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  $p$ )

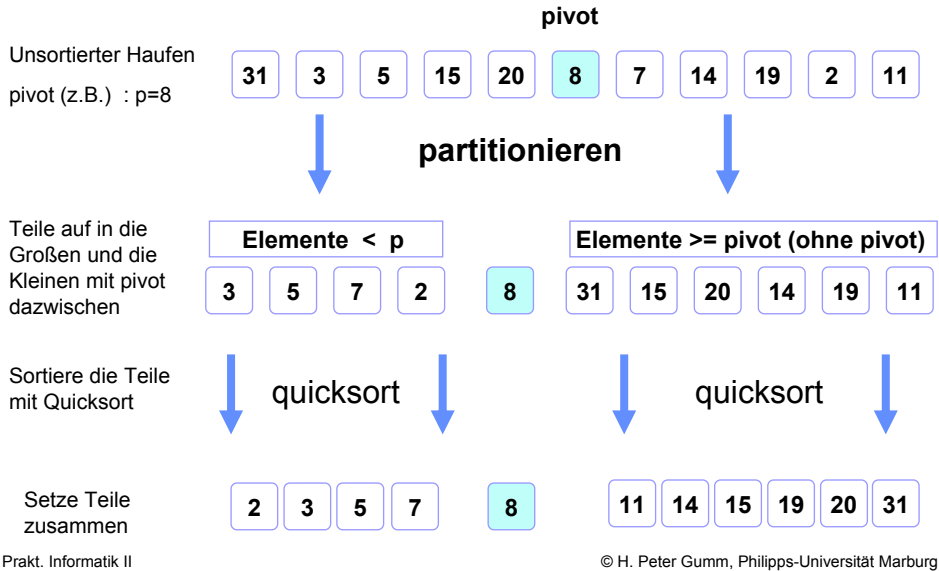


- Sortiere KL (mit QuickSort), Sortiere GR (mit QuickSort), dann setze die sortierten Teile zusammen, mit  $p$  dazwischen.





# QuickSort – schematisches Beispiel



# Quicksort

- Behandle Trivialfälle
- Wähle Pivot
- Partitioniere und bestimme endgültige Position des Pivot
- Sortiere untere Hälfte
- Sortiere obere Hälfte

```
private static void quSort(int[] daten, int i, int j){
    // Trivialfälle: länge <= 1:
    if (j==i+1 && daten[j]< daten[i]) swap(daten,i,j);
    if (j-i<=1) return;

    // Wähle Index für einen pivot mit i < pivIndex < j:
    int pivIndex=(i+j)/2;

    // Partitioniere und gebe Index für Pivot zurück:
    pivIndex = partition(daten,i,j,pivIndex);

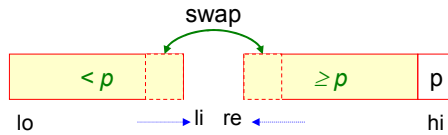
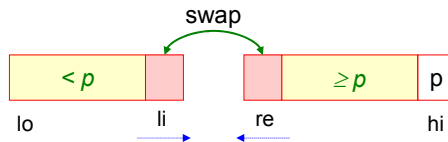
    // Sortiere untere Hälfte daten[i..pivIndex-1]
    quSort(daten,i,pivIndex-1);

    // Sortiere obere Hälfte daten[pivIndex..j]
    quSort(daten,pivIndex+1,j);
}
```

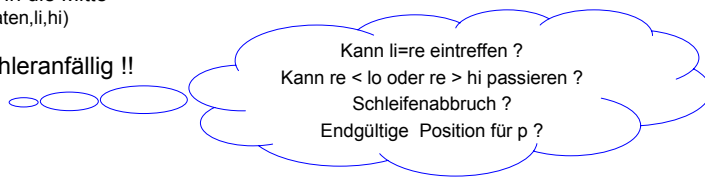


# Partitionieren

- 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 !!



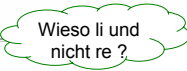
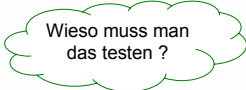
# Partitionieren in Java



```
private static int partition(int[] daten, int lo, int hi, int pivIndex) {
    int pivot = daten[pivIndex];
    //pivot ans Ende des Arrays befördern
    swap(daten, pivIndex, hi);

    int li=lo;
    int re=hi-1;
    while (li<re) { // Invariante:
        assert allBelow(daten, lo, li-1, pivot)
            && allAboveOrEqual(daten, re+1, hi, pivot)
            && li <= re+1 && li != re;

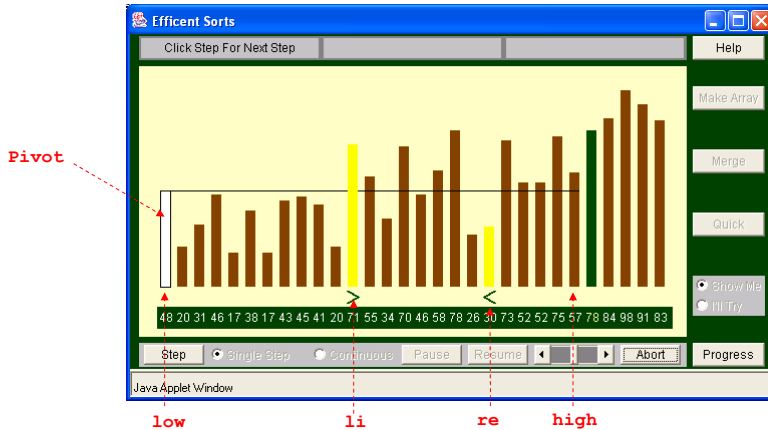
        while(daten[li] < pivot) li++;
        while(daten[re] >= pivot && re > lo) re--;
        if (li < re) swap(daten, li, re);
    }
    // Pivot an endgültige Position befördern:
    swap(daten, li, hi);
    assert allBelow(daten, lo, li-1, daten[li])
        && allAboveOrEqual(daten, li+1, hi, daten[li]);
    return li; // Pivot-Position
}
```





# Animation

- Auf
  - <http://student.seas.gwu.edu/~idsv/idsv.html>
- unter der Rubrik
  - Efficient Sorts
- können Sie QuickSort und MergeSort animieren und ausprobieren



Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



# MergeSort



- Der Aufbau von MergeSort ist
  - Divide
    - Teile den Array in zwei etwa gleich große Abschnitte
  - Sort
    - Sortieren den linken Abschnitt
    - Sortieren den rechten Abschnitt
  - Merge
    - Füge die Abschnitte unter Beibehaltung der Ordnung zusammen

```

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

private static void mSort(int [] daten, int lo, int hi){
    if (hi - lo >= 1){
        int mitte = (lo+hi+1)/2;
        mSort(daten,lo,mitte-1);
        mSort(daten,mitte,hi);
        merge(daten,lo,mitte,hi);
    }
}

```

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

Prakt. Informatik II

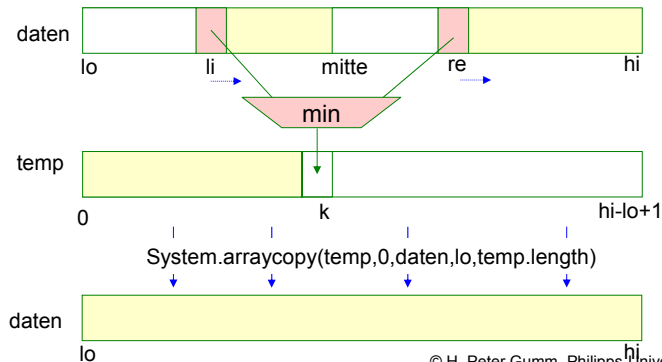
© H. Peter Gumm, Philipps-Universität Marburg



# Merge – die Idee



- `daten[lo..mitte-1]` und `daten[mitte..hi]` sind sortiert
- Mische sie unter Beibehaltung der Reihenfolge in einen temporären Array `temp`
- Kopiere `temp[0..hi-lo+1]` in `daten[lo..hi]`
- Vorsicht: Für das Kopieren wird **nicht** `swap` benutzt. Man muss überlegen, dass sortierter Array eine Permutation des ursprünglichen bleibt !



Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



# Merge – der Code

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

Kurze Auswertung verhindert Bereichsüberschreitung !!!

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

Prakt. Informatik II

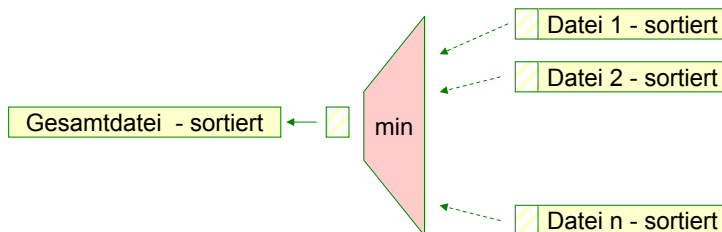
© H. Peter Gumm, Philipps-Universität Marburg



# Externes mergeSort



- 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.



Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



# Animation

- Wieder von
  - <http://student.seas.gwu.edu/~idsv/idsv.html>
- die Animation von mergeSort

Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



# Verallgemeinerung

- Alle Sortieralgorithmen funktionieren nicht nur auf Arrays von Zahlen, sondern
  - auf geordneten Mengen
  - auf Datentypen die Comparable implementieren

swap muss reimplementiert werden

Es kann mit dem alten swap koexistieren

In der Klasse existieren gleichzeitig

void swap(int[], int, int)

void swap(Comparable[], int, int)

```
/** Allgemeines SelectionSort */
/** Neue (zusätzliche) Methode swap */
private static void swap(Comparable[] daten, int i, int j){
    Comparable temp=daten[i];
    daten[i]=daten[j];
    daten[j]=temp;
}
/** Allgemeineres selectionSort */
public static void selectionSort(Comparable[] daten){
    for(int j=0; j<daten.length-1; j++){
        // Invariante - daten[0..j-1] ist sortiert:
        int minIndex = j;
        // Suche das nächstgrößere Element
        for (int k=minIndex+1; k<daten.length; k++)
            if (daten[k].compareTo(daten[minIndex])<1)
                minIndex=k;
        // Bringe es an die richtige Stelle
        swap(daten, minIndex, j);
    }
}
```