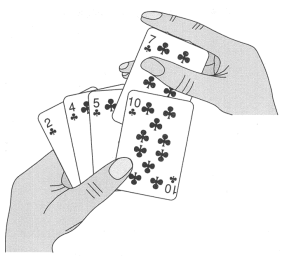


Algorithmen und Datenstrukturen

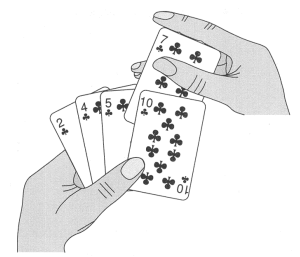
Wintersemester 2017/18

2. Vorlesung

Sortieren mit anderen Mitteln



Teile und herrsche



Idee:

- teile den Kartenstapel in zwei ungefähr gleichgroße Teile,
- sortiere die Teile (z.B. durch verschiedene Personen) und
- füge die Teilstapel zu einem sortierten Stapel zusammen.

Allgemein:

Teile... eine Instanz in kleinere Instanzen *desselben* Problems.

Herrsche... durch **rekursives** Lösen von Teilinstanzen – nur falls diese sehr klein sind, löse sie direkt.

Kombiniere... die Teillösungen zu einer Lösung der ursprünglichen Instanz.

*) Abb. aus [Corman et al. „Introduction to Algorithms“, MIT Press]

Teile und herrsche

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)

Defaultwerte –

Dadurch wird die Funktion

MergeSort(A) \equiv

MergeSort(A, 1, A.length)

definiert.

Allgemein:

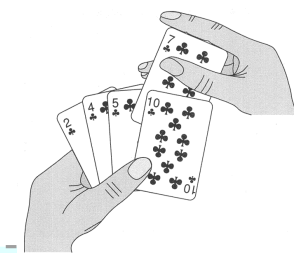
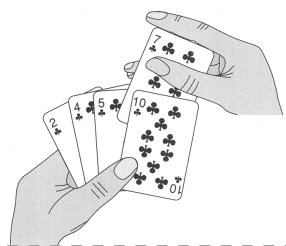
Teile... eine Instanz in kleinere Instanzen *desselben* Problems.

Herrsche... durch *rekursives* Lösen von Teilinstanzen – nur falls diese sehr klein sind, löse sie direkt.

Kombiniere... die Teillösungen zu einer Lösung der ursprünglichen Instanz.

*) Abb. aus [Corman et al. „Introduction to Algorithms“, MIT Press]

Teile und herrsche



```
MergeSort(int[] A, int  $\ell = 1$ , int  $r = A.length$ )
```

```
  if  $\ell < r$  then
```

$m = \lfloor (\ell + r) / 2 \rfloor$	}	teile
MergeSort(A, ℓ, m)	}	herrsche
MergeSort($A, m + 1, r$)	}	
Merge(A, ℓ, m, r)	}	kombiniere

To do!

Allgemein:

Teile... eine Instanz in kleinere Instanzen *desselben* Problems.

Herrsche... durch *rekursives* Lösen von Teilinstanzen – nur falls diese sehr klein sind, löse sie direkt.

Kombiniere... die Teillösungen zu einer Lösung der ursprünglichen Instanz.

Kombiniere

```
Merge(int[] A, int  $\ell$ , int  $m$ , int  $r$ )
```

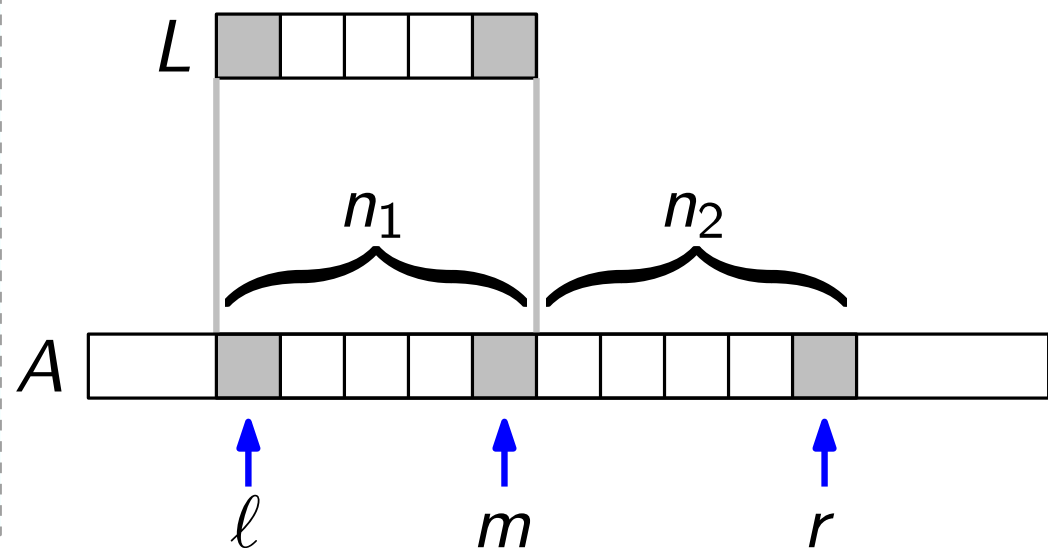
```
   $n_1 = m - \ell + 1$ ;  $n_2 = r - m$ 
```

```
  L = new int[1.. $n_1 + 1$ ]; R = new int[1.. $n_2 + 1$ ]
```

```
  L[1.. $n_1$ ] = A[ $\ell$ .. $m$ ]
```

```
  for  $i = 1$  to  $n_1$  do
```

```
    L[ $i$ ] = A[( $\ell - 1$ ) +  $i$ ]
```



Kombiniere

Merge(int[] A, int ℓ , int m , int r)

$n_1 = m - \ell + 1$; $n_2 = r - m$

$L = \text{new int}[1..n_1 + 1]$; $R = \text{new int}[1..n_2 + 1]$

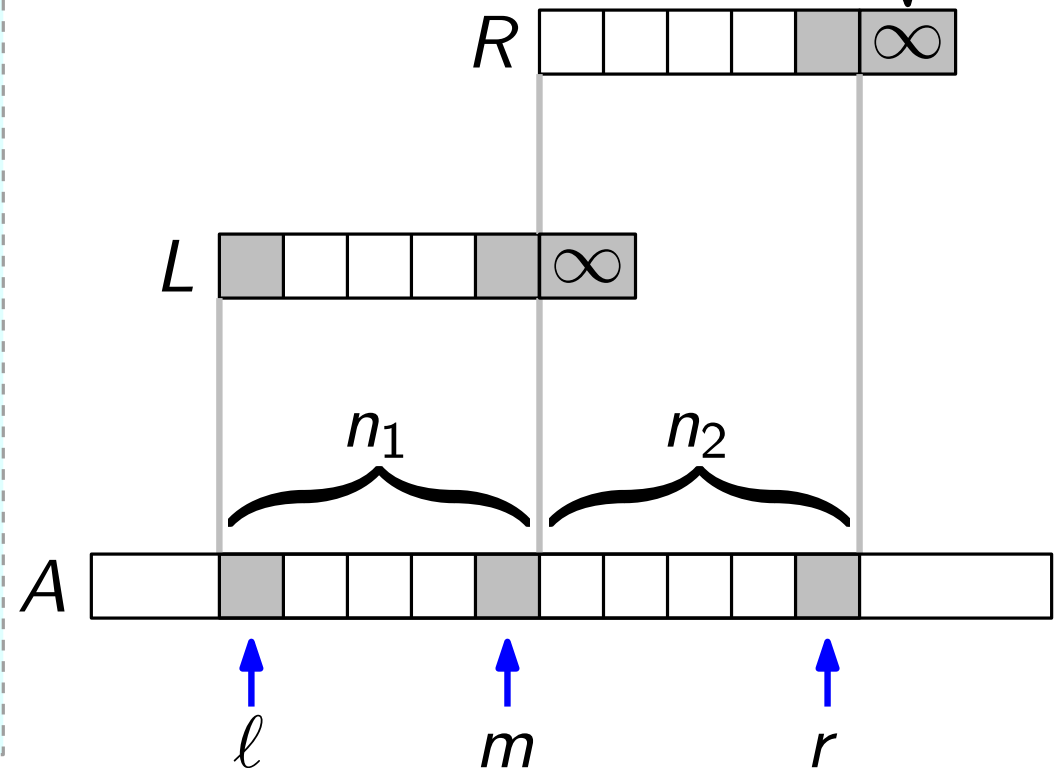
$L[1..n_1] = A[\ell..m]$

$R[1..n_2] = A[m + 1..r]$

$L[n_1 + 1] = R[n_2 + 1] = \infty$



Stopper (engl. *sentinel*)



Kombiniere

```
Merge(int[] A, int  $\ell$ , int  $m$ , int  $r$ )
```

```
   $n_1 = m - \ell + 1$ ;  $n_2 = r - m$ 
```

```
  L = new int[1.. $n_1 + 1$ ]; R = new int[1.. $n_2 + 1$ ]
```

```
  L[1.. $n_1$ ] = A[ $\ell$ .. $m$ ]
```

```
  R[1.. $n_2$ ] = A[ $m + 1$ .. $r$ ]
```

```
  L[ $n_1 + 1$ ] = R[ $n_2 + 1$ ] =  $\infty$ 
```

```
   $i = j = 1$ 
```

```
  for  $k = \ell$  to  $r$  do
```

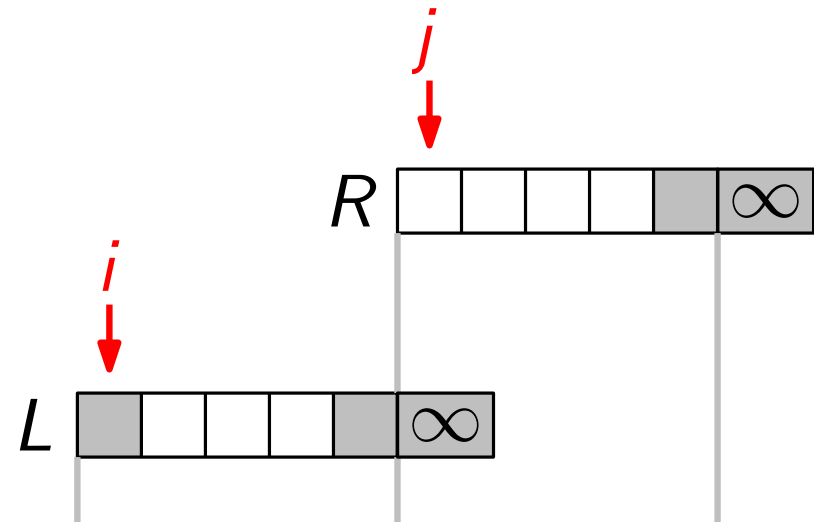
Aufgabe:

Schließen Sie Ihre Bücher und Ihren Browser!

Schreiben Sie mit Ihrer NachbarIn den Rest der Routine!

Benutzen Sie dazu die beiden neuen Felder L und R .

Sie haben **5 Minuten**.



Korrektheit von Merge

... nach Schema „F“!

0. Schleifeninvariante

- $A[\ell..k-1]$ enthält die $k-\ell$ kleinsten Elemente von $L \cup R$ sortiert.
- $L[i]$ und $R[j]$ sind die kleinsten Elemente in L bzw. R , die noch nicht in A kopiert wurden.

```
Merge(int[] A, int  $\ell$ , int  $m$ , int  $r$ )
   $n_1 = m - \ell + 1$ ;  $n_2 = r - m$ 
  lege  $L[1..n_1 + 1]$  und  $R[1..n_2 + 1]$  an
   $L[1..n_1] = A[\ell..m]$ 
   $R[1..n_2] = A[m + 1..r]$ 
   $L[n_1 + 1] = R[n_2 + 1] = \infty$ 
   $i = j = 1$ 
  for  $k = \ell$  to  $r$  do
    if  $L[i] \leq R[j]$  then
       $A[k] = L[i]$ 
       $i = i + 1$ 
    else
       $A[k] = R[j]$ 
       $j = j + 1$ 
```

1. Initialisierung ✓

- Da beim ersten Schleifendurchlauf $k = \ell$ gilt, enthält $A[\ell..k-1] = \langle \rangle$ die 0 kleinsten Elem. von $L \cup R$.
- Da $i = j = 1$, sind $L[i]$ und $R[j]$ die kleinsten noch nicht kopierten Elem.

Korrektheit von Merge

... nach Schema „F“!

0. Schleifeninvariante

- $A[\ell..k-1]$ enthält die $k-\ell$ kleinsten Elemente von $L \cup R$ sortiert.
- $L[i]$ und $R[j]$ sind die kleinsten Elemente in L bzw. R , die noch nicht in A kopiert wurden.

```

Merge(int[] A, int ℓ, int m, int r)
  n1 = m - ℓ + 1; n2 = r - m
  lege L[1..n1 + 1] und R[1..n2 + 1] an
  L[1..n1] = A[ℓ..m]
  R[1..n2] = A[m + 1..r]
  L[n1 + 1] = R[n2 + 1] = ∞
  i = j = 1
  for k = ℓ to r do
    if L[i] ≤ R[j] then // Fall (a)
      A[k] = L[i]
      i = i + 1
    else // Fall (b)
      A[k] = R[j]
      j = j + 1
  
```

1. Initialisierung

2. Aufrechterhaltung

(Fall (b) symmetrisch.)

- Zwei Fälle: (a) $L[i] \leq R[j]$, (b) $R[j] < L[i]$. Betrachte Fall (a).
- Nun gilt:
 - $A[\ell..k]$ enthält die kleinsten $k-\ell+1$ Elem. sortiert
 - $L[i+1]$ ist kleinstes noch nicht kopiertes Elem. in L .

erhöhe $i \Rightarrow L[i]$ ist kleinstes noch nicht kopiertes Elem. in L .

erhöhe $k \Rightarrow A[\ell..k-1]$ enthält die kleinsten $k-\ell$ Elem. sortiert

Korrektheit von Merge

... nach Schema „F“!

0. Schleifeninvariante

- $A[\ell..k-1]$ enthält die $k-\ell$ kleinsten Elemente von $L \cup R$ sortiert.
- $L[i]$ und $R[j]$ sind die kleinsten Elemente in L bzw. R , die noch nicht in A kopiert wurden.

```

Merge(int[] A, int ℓ, int m, int r)
  n1 = m - ℓ + 1; n2 = r - m
  lege L[1..n1 + 1] und R[1..n2 + 1] an
  L[1..n1] = A[ℓ..m]
  R[1..n2] = A[m + 1..r]
  L[n1 + 1] = R[n2 + 1] = ∞
  i = j = 1
  for k = ℓ to r do
    if L[i] ≤ R[j] then
      A[k] = L[i]
      i = i + 1
    else
      A[k] = R[j]
      j = j + 1
  
```

1. Initialisierung ✓ 2. Aufrechterhaltung ✓ 3. Terminierung ✓

- Nach Abbruch der for-Schleife gilt $k = r + 1$.

⇒ $A[\ell..k-1] = A[\ell..r]$ enthält die $r - \ell + 1$ kleinsten Elem. von $L \cup R$ sortiert.

- $|L \cup R| = n_1 + n_2 + 2 = r - \ell + 3$, d.h. $A[\ell..r]$ korrekt sort. +2 Stopper

Korrektheit von Merge

... nach Schema „F“!

0. Schleifeninvariante

- $A[\ell..k-1]$ enthält die $k - \ell$ kleinsten Elemente von $L \cup R$ sortiert.
- $L[i]$ und $R[j]$ sind die kleinsten Elemente in L bzw. R , die noch nicht in A kopiert wurden.

```
Merge(int[] A, int  $\ell$ , int  $m$ , int  $r$ )
   $n_1 = m - \ell + 1$ ;  $n_2 = r - m$ 
  lege  $L[1..n_1 + 1]$  und  $R[1..n_2 + 1]$  an
   $L[1..n_1] = A[\ell..m]$ 
   $R[1..n_2] = A[m + 1..r]$ 
   $L[n_1 + 1] = R[n_2 + 1] = \infty$ 
   $i = j = 1$ 
  for  $k = \ell$  to  $r$  do
    if  $L[i] \leq R[j]$  then
       $A[k] = L[i]$ 
       $i = i + 1$ 
    else
       $A[k] = R[j]$ 
       $j = j + 1$ 
```

1. Initialisierung ✓

2. Aufrechterhaltung ✓

3. Terminierung ✓

Also ist Merge korrekt!

q.e.d.

Laufzeit?

Merge macht genau $r - \ell + 1$ Vergleiche.

Und MergeSort?

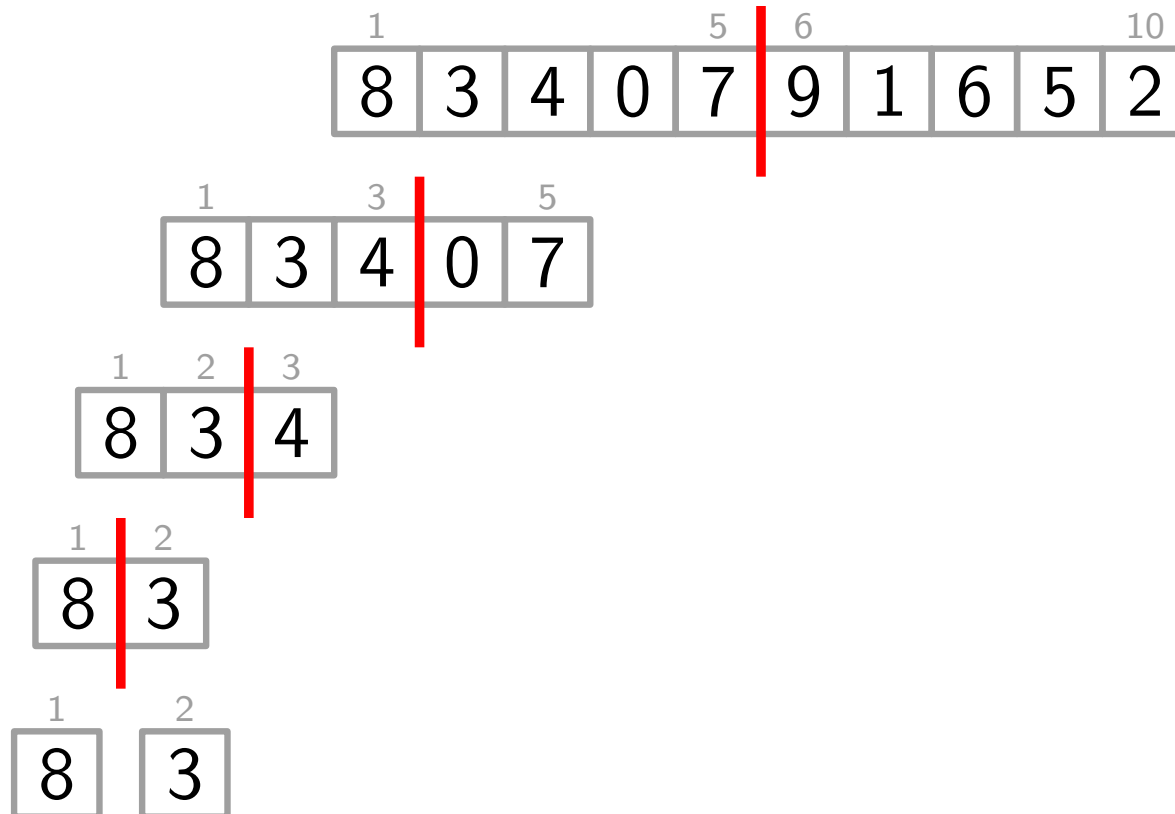
Korrekt? Effizient?

MergeSort – ein Beispiel

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)

if $\ell < r$ **then**

$m = \lfloor (\ell + r) / 2 \rfloor$	}	teile
MergeSort(A, ℓ , m)	}	herrsche
MergeSort(A, $m + 1$, r)	}	
Merge(A, ℓ , m , r)	}	kombiniere



MergeSort – ein Beispiel

MergeSort(int[] A, int $l = 1$, int $r = A.length$)

if $l < r$ **then**

$m = \lfloor (l + r) / 2 \rfloor$

} teile

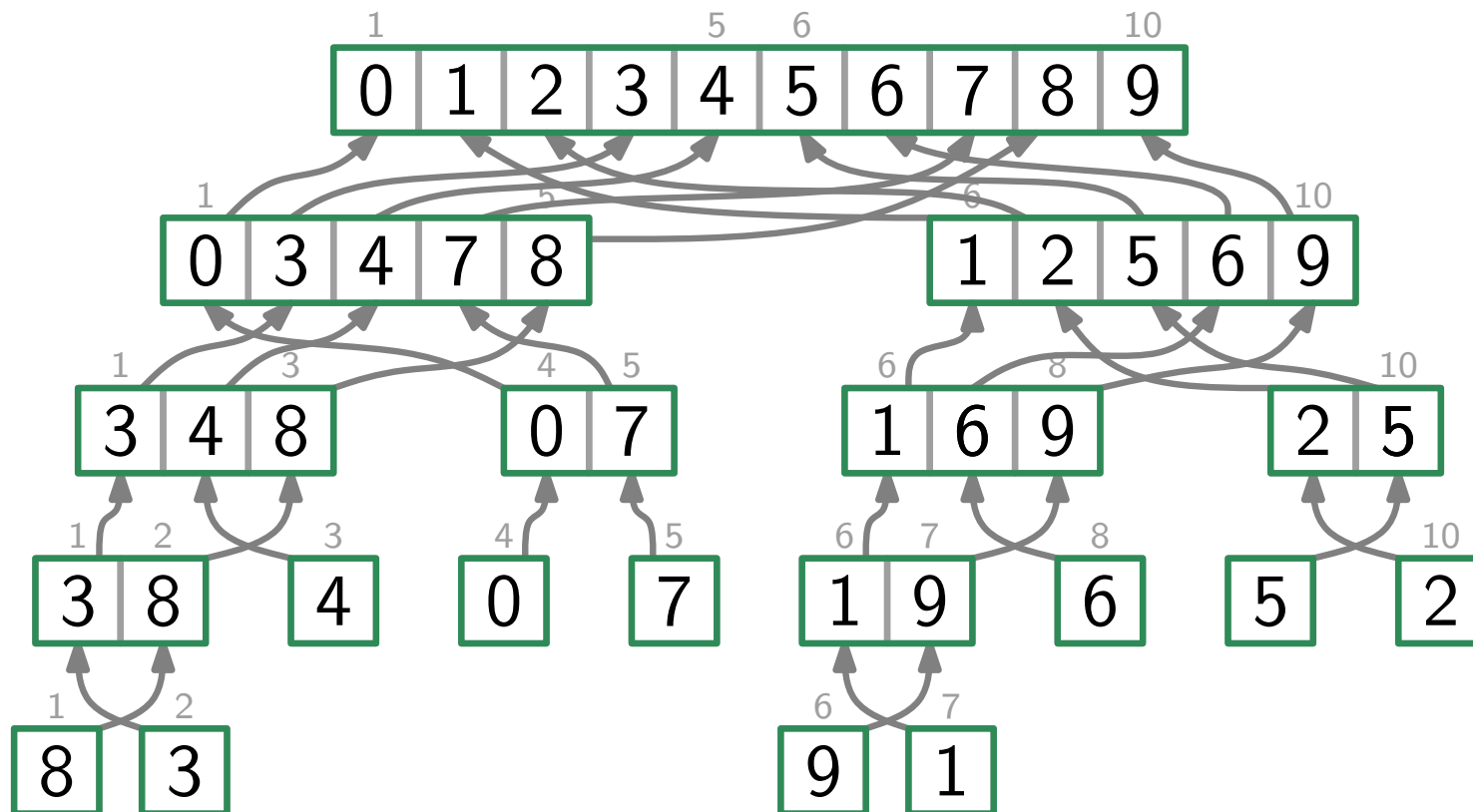
MergeSort(A, l , m)

} herrsche

MergeSort(A, $m + 1$, r)

} kombiniere

Merge(A, l , m , r)



MergeSort – ein Beispiel

```
MergeSort(int[] A, int  $\ell = 1$ , int  $r = A.length$ )
```

```
  if  $\ell < r$  then
```

```
     $m = \lfloor (\ell + r) / 2 \rfloor$ 
```

```
  } teile
```

```
    MergeSort(A,  $\ell$ ,  $m$ )
```

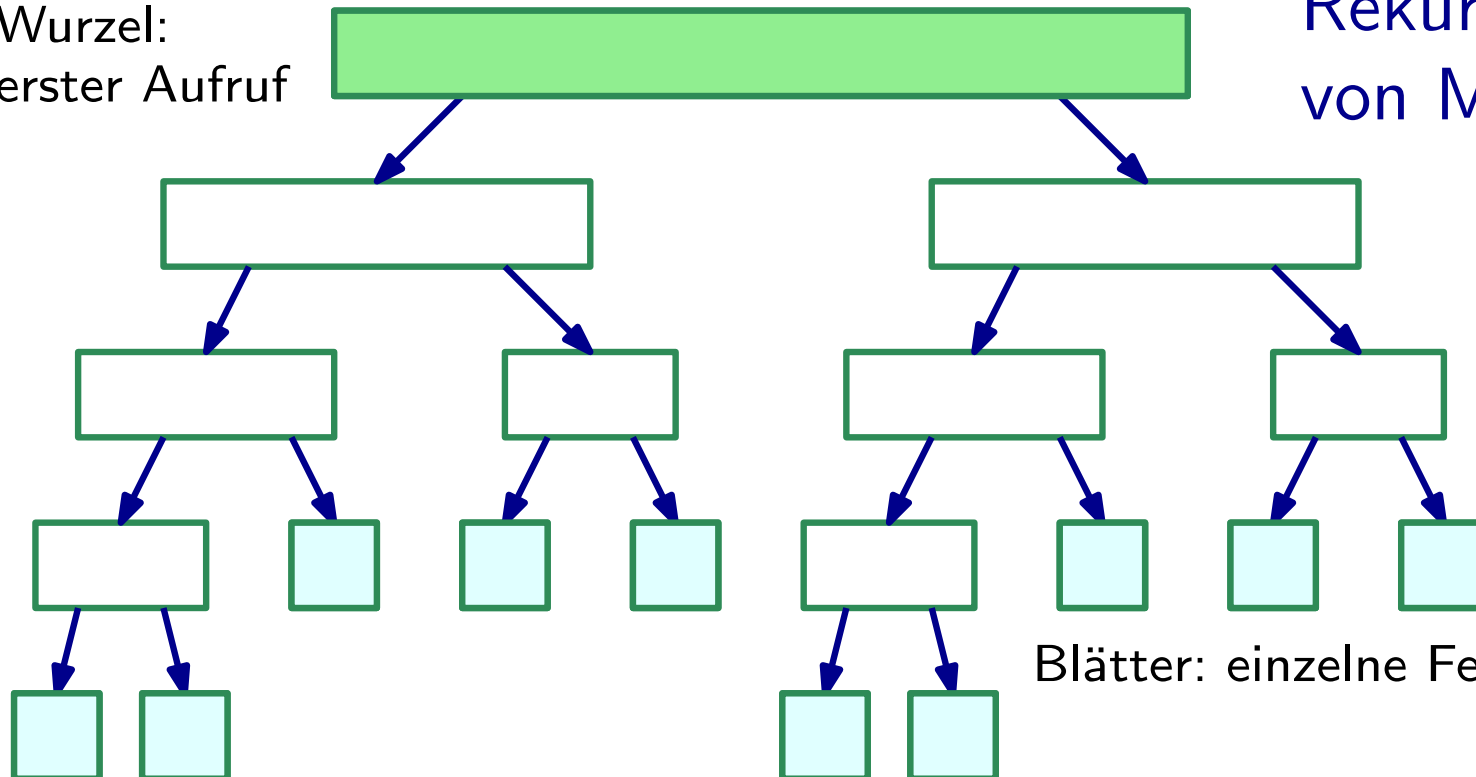
```
  } herrsche
```

```
    MergeSort(A,  $m + 1$ ,  $r$ )
```

```
  } kombiniere
```

```
    Merge(A,  $\ell$ ,  $m$ ,  $r$ )
```

Wurzel:
erster Aufruf



Rekursionsbaum
von MergeSort:

Baum der
rekursiven
Aufrufe

Blätter: einzelne Feldelemente

Korrektheit von Mergesort

```

MergeSort(int[] A, int  $\ell = 1$ , int  $r = A.length$ )
  if  $\ell < r$  then
     $m = \lfloor (\ell + r) / 2 \rfloor$  } teile
    MergeSort(A,  $\ell$ ,  $m$ ) } herrsche
    MergeSort(A,  $m + 1$ ,  $r$ ) }
    Merge(A,  $\ell$ ,  $m$ ,  $r$ ) } kombiniere
  
```

Korrekt? Welche Beweistechnik? Hm, MergeSort ist *rekursiv*...

Vollständige Induktion über $n = r - \ell + 1$ ($= A[\ell..r].length$):

$n = 1$: *Induktionsanfang*

Dann ist $\ell = r$.

\Rightarrow if-Block wird nicht betreten.

D.h. nichts passiert.

OK, da $A[\ell..r]$ schon sortiert.



Korrektheit von Mergesort

```

MergeSort(int[] A, int  $\ell = 1$ , int  $r = A.length$ )
  if  $\ell < r$  then
    {  $m = \lfloor (\ell + r)/2 \rfloor$  } teile
    { MergeSort(A,  $\ell$ ,  $m$ ) } herrsche
    { MergeSort(A,  $m + 1$ ,  $r$ ) } herrsche
    { Merge(A,  $\ell$ ,  $m$ ,  $r$ ) } kombiniere
  
```

$n > 1$: *Induktionsschritt*

Induktionsannahme: MergeSort korrekt für Felder d. Länge $< n$.

Wegen $n > 1$ ist $\ell < r$. \Rightarrow if-Block wird betreten.

Nach Wahl von m gilt $\ell \leq m < r$.

$\Rightarrow A[\ell..m]$ und $A[m + 1..r]$ sind *kürzer* als $A[\ell..r]$.

$\stackrel{\text{i.A.}}{\Rightarrow}$ MergeSort(A, ℓ , m) ist korrekt und MergeSort(A, $m + 1$, r) ist korrekt. } MergeSort(A, ℓ , r) ist korrekt, d.h. MS für Felder d. Länge n .
 Schon bewiesen: Merge ist korrekt. } für Felder d. Länge n . \square

Übersicht

Techniken für Korrektheitsbeweise

- iterative Algorithmen (à la InsertionSort)

Schleifeninvariante (Schema „F“)

- rekursive Algorithmen (à la MergeSort)

Induktion