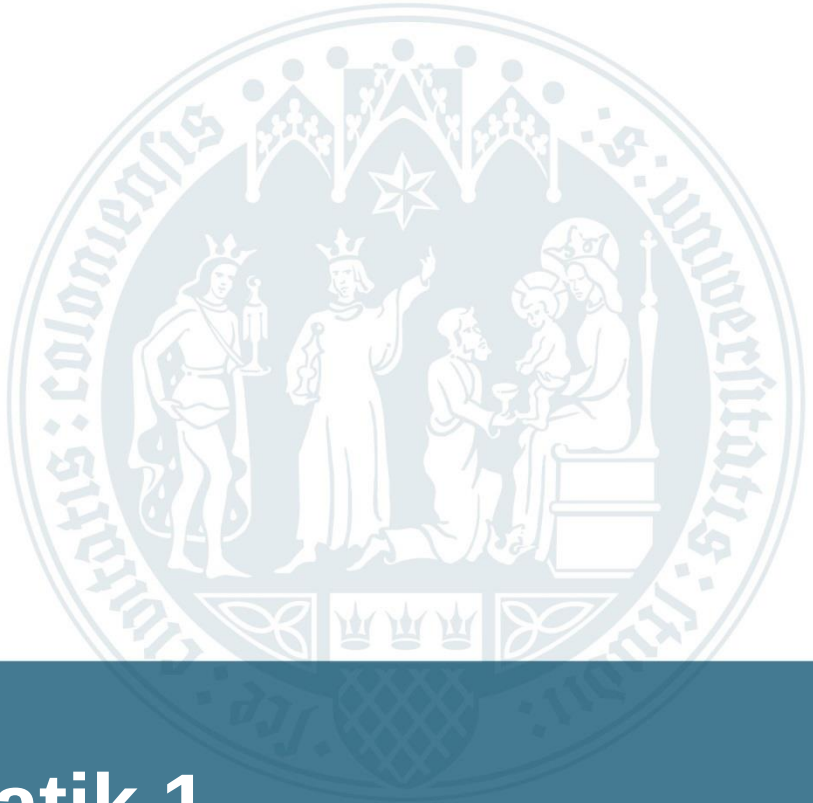
Universität zu Köln



Grundzüge der Informatik 1

Vorlesung 5

Überblick

Überblick

- Wiederholung
- Korrektheitsbeweise (Rekursionen)
- Teile & Herrsche Verfahren

Wiederholung

Korrektheitsbeweis

- Formale Argumentation, dass ein Algorithmus korrekt arbeitet

Problembeschreibung

- Definiert für eine Menge von zulässigen Eingaben die zugehörigen gewünschten Ausgaben

Korrektheit

- Wir bezeichnen einen Algorithmus für eine vorgegebene Problembeschreibung als korrekt, wenn er für jede zulässige Eingabe die in der Problembeschreibung spezifizierte Ausgabe berechnet
- Streng genommen kann man also nur von Korrektheit sprechen, wenn vorher festgelegt wurde, was der Algorithmus eigentlich tun soll

Wiederholung

Beweisprinzip der mathematischen Induktion

- Sei A(n) eine Aussage über eine natürliche Zahl $n \in \mathbb{N}=\{1, 2, 3,\ldots\}$
- Wir wollen zeigen, dass die Aussage für alle natürlichen Zahlen gilt

Mathematische Induktion besteht aus 2 Hauptkomponenten

- Induktionsanfang: Aussage A(1) stimmt
- Induktionsschritt: Wenn A(n) gilt, dann gilt auch A(n+1)

Beispiel

- A(1)
- A(2)
- A(3)
- …

Wiederholung

Schleifeninvariante

- A(n) ist eine Aussage über den Zustand des Algorithmus vor dem n-ten Durchlauf einer Schleife
- Eine Schleifeninvariante ist korrekt, wenn Sie zu Beginn jedes Schleifendurchlaufs erfüllt ist.
- A(1) wird auch als Initialisierung bezeichnet.

Korrektheitsbeweis für Invarianten

- Induktionsanfang: Die Aussage A(1) gilt
- Induktionsschluss: Gilt A(n) und ist die Eintrittsbedingung der Schleife erfüllt so gilt auch A(n+1)

Korrektheitsbeweise - Schleifeninvarianten

```
MaxSuche(A, n)     \* Array A der Länge n wird übergeben
1.    max = 1
2.    for i=2 to n do
3.        if A[i] > A[max] then max = i
4.    return max
```

Schleifeninvariante

- A(i): A[max] ist ein größtes Element aus dem Teilarray A[1..i-1]

Lemma 4.1

- A(i) ist eine korrekte Schleifeninvariante.

Korrektheitsbeweise

Satz 5.1

- Algorithmus MaxSuche(A,n) berechnet den Index eines größten Elements aus einem Feld A mit n Zahlen.

Beweis

- Der Schleifenaustritt aus der for-Schleife erfolgt für i=n+1
- Nach Lemma 4.1 ist die Schleifeninvariante erfüllt und es gilt somit, dass A[max] ein größtes Element aus dem Array A[1..i-1] = A[1..n] ist
- Der **return** Befehl in Zeile 4 gibt max zurück. Dies ist damit der Index eines größten Elements.

Korrektheitsbeweise - Schleifeninvarianten

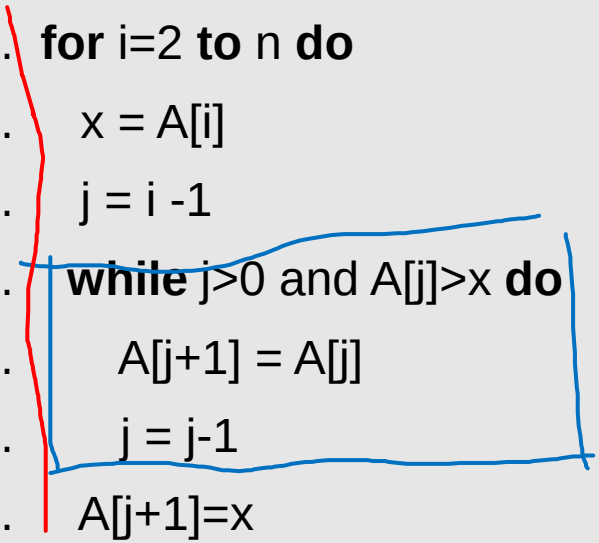
```
MaxSuche(A, n)      \* Array A der Länge n wird übergeben
1.    max = 1
2.    for i=2 to n do                    \* Invariante: A[max] ist ein größtes
3.        if A[i] > A[max] then max = i  \* Element aus dem Teilarray A[1..i-1]
4.    return max
```

Kommentare in Programmen

- Schleifeninvarianten eignen sich sehr gut, um Algorithmen und Programme zu kommentieren

Korrektheitsbeweise - InsertionSort

```
InsertionSort(A, n)
1.  for i=2 to n do
2.     x = A[i]
3.     j = i -1
4.     while j>0 and A[j]>x do
5.        A[j+1] = A[j]
6.        j = j-1
7.     A[j+1]=x
```



Korrektheitsbeweis für InsertionSort

- Beobachtung: Zwei Schleifen
- Innere Schleife ist **while**-Schleife
- Vereinfachung: Wir fassen den Rumpf der **for**-Schleife zusammen

Korrektheitsbeweise - InsertionSort

InsertionSort(A, n)

1. **for** i=2 **to** n **do**
2. Füge A[i] in das sortierte Teilarray A[1..i-1] ein

Korrektheitsbeweis für InsertionSort

- Beobachtung: Zwei Schleifen
- Innere Schleife ist **while**-Schleife
- Vereinfachung: Wir fassen den Rumpf der **for**-Schleife zusammen

Korrektheitsbeweise - InsertionSort

InsertionSort(A, n)

1. **for** i=2 **to** n **do**
2. Füge A[i] in das sortierte Teilarray A[1..i-1] ein

Korrektheitsbeweis für InsertionSort

- Beobachtung: Zwei Schleifen
- Innere Schleife ist **while**-Schleife
- Vereinfachung: Wir fassen den Rumpf der **for**-Schleife zusammen

Lemma 5.1

- Die **for**-Schleife von Algorithmus InsertionSort erfüllt folgende Invariante:
- Das Teilarray A[1..i-1] ist aufsteigend sortiert.

Beweis (Teil 1):

- Induktionsanfang (i=2): Das Teilarray A[1..i-1] = A[1..1] enthält nur eine Zahl und ist damit sortiert.

Korrektheitsbeweise - InsertionSort

InsertionSort(A, n)

1. **for** i=2 **to** n **do**
2. Füge A[i] in das sortierte Teilarray A[1..i-1] ein

Korrektheitsbeweis für InsertionSort

- Beobachtung: Zwei Schleifen
- Innere Schleife ist **while**-Schleife
- Vereinfachung: Wir fassen den Rumpf der **for**-Schleife zusammen

Beweis (Teil 2):

- Induktionsannahme: Die Invariante gilt für $i \leq n$
- Induktionsschluss: Nach Induktionsannahme ist das Teilarray A[1..i-1] sortiert.
- Zeile 2 des Algorithmus fügt A[i] an die richtige Stelle im sortierten Teilarray ein. Damit ist nach dem Einfügen A[1..i] sortiert und die Invariante gilt für i+1.

Korrektheitsbeweise - InsertionSort

```
InsertionSort(A, n)                  \* Feld A der Länge n wird übergeben
1.  for i=2 to n do                  \* Invariante: A[1..i-1] ist aufsteigend sortiert
2.      x = A[i]                     \* Schleifenrumpf: A[i] wird in Teilarray A[1..i-1]
3.      j = i -1                     \* eingefügt
4.      while j>0 and A[j]>x do
5.          A[j+1] = A[j]
6.          j = j-1
7.      A[j+1]=x
```

Korrektheitsbeweise - InsertionSort

Satz 5.2 (Korrektheit von InsertionSort)

- Algorithmus InsertionSort(A,n) sortiert ein Feld A der Länge n.

Beweis

- Die **for**-Schleife endet, wenn i den Wert n+1 hat
- Nach Lemma 5.1 gilt die Schleifeninvariante, dass A[1..i-1] = A[1..n] aufsteigend sortiert ist
- Daher ist am Ende des Algorithmus das Feld aufsteigend sortiert

Korrektheitsbeweise - Rekursionen

```
Sum(A,n)
1. if n=1 then return A[1]
2. else
3.    W= Sum(A,n-1)
4.    return W+A[n]
```

Problem

- Die Anzahl rekursiver Aufrufe hängt von der Eingabe ab

Korrektheitsbeweise - Rekursionen

```
Sum(A,n)
1. if n=1 then return A[1]
2. else
3.    W= Sum(A,n-1)
4.    return W+A[n]
```

Lösung

- Rekursion ist in gewisser Hinsicht das Gegenstück zu Iteration/Induktion
- Rekursionsabbruch entspricht Induktionsanfang
- Rekursionsaufruf entspricht Induktionsschritt
- Korrektheit lässt sich daher per Induktion zeigen

Korrektheitsbeweise - Rekursionen

```
Sum(A,n)
1. if n=1 then return A[1]
2. else
3.    W= Sum(A,n-1)
4.    return W+A[n]
```

Satz 5.3

- Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge von Feld A.

Beweis (Teil 1)

- Induktionsanfang (n=1): Der Algorithmus gibt in Zeile 1 korrekt A[1] zurück
- Induktionsannahme:
  Für n>1 berechne Sum(A,n-1) die Summe der ersten n-1 Zahlen in A
- Induktionsschluss: Wir betrachten den Aufruf von Sum(A,n). Da n>1 ist, wird der **else**-Fall der ersten **if**-Anweisung aufgerufen.

Korrektheitsbeweise - Rekursionen

```
Sum(A,n)
1. if n=1 then return A[1]
2. else
3.    W= Sum(A,n-1)
4.    return W+A[n]
```

Satz 5.3

- Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge von Feld A.

Beweis (Teil 2)

- In Zeile 3 wird W auf Sum(A,n-1) gesetzt
- Nach I.V. ist dies die Summe der ersten n-1 Einträge von A
- Nun wird in Zeile 4 A[n]+W, also die Summe der ersten n Einträge von A zurückgegeben
- Es folgt, dass Sum(A,n) die Summe der ersten n Zahlen berechnet

Korrektheitsbeweise

Zusammenfassung

- Ohne Schleifen: Nachvollziehen der Abfolge der Befehle
- Schleifen: Korrektheit mit Hilfe von Invarianten und Induktion
- Rekursion: Korrektheit mit Hilfe von Induktion

Motivation

- Vertieftes Verständnis des Algorithmus
- „Sprache“, um über die Funktionsweise von Algorithmen zu reden
- Invarianten helfen bei Kommentaren

Algorithmenentwurf durch Rekursion

Sortieren

- Eines der wichtigsten Probleme in der Informatik
- Sortieren erlaubt schnelles Suchen
- Beispiel: Telefonbuch

Bisher

- Sortieralgorithmus OurSort mit Hilfe von Rekursion
- Wir haben das Sortieren von n Zahlen auf n-1 Zahlen zurückgeführt
- Umwandlung in einen iterativen Algorithmus -> InsertionSort

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile

15	7	6	13	25	4	9	12

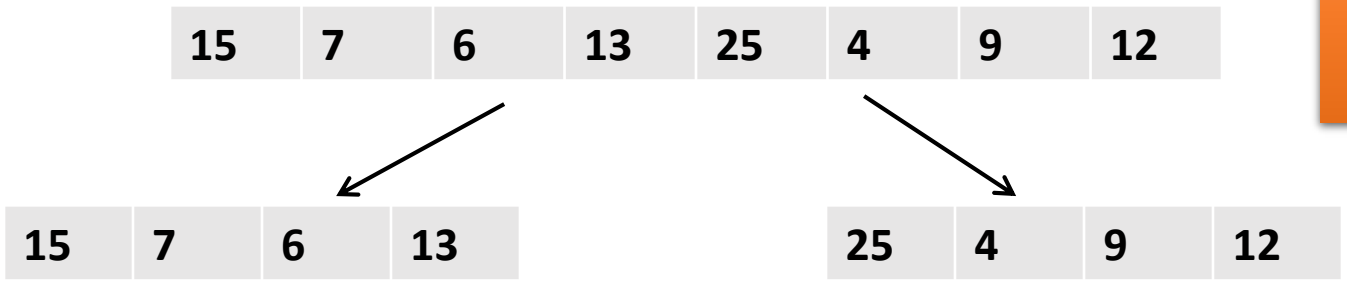
Schritt 1:
Aufteilen der
Eingabe

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



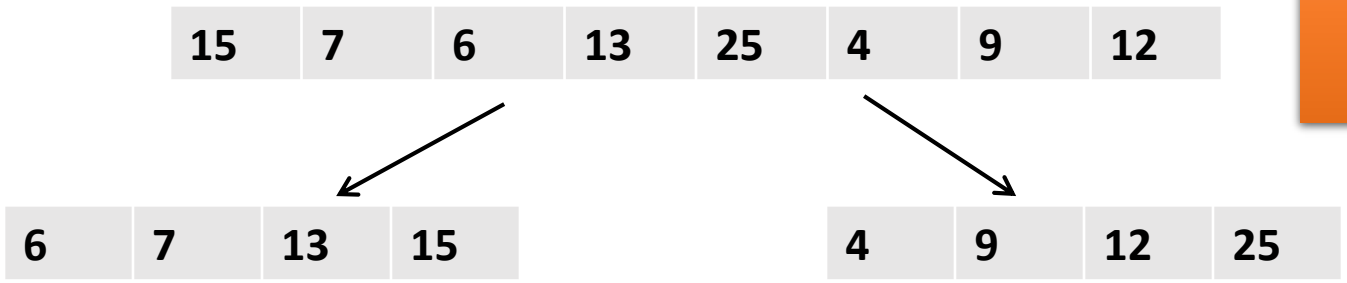
Schritt 1:
Aufteilen der
Eingabe

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



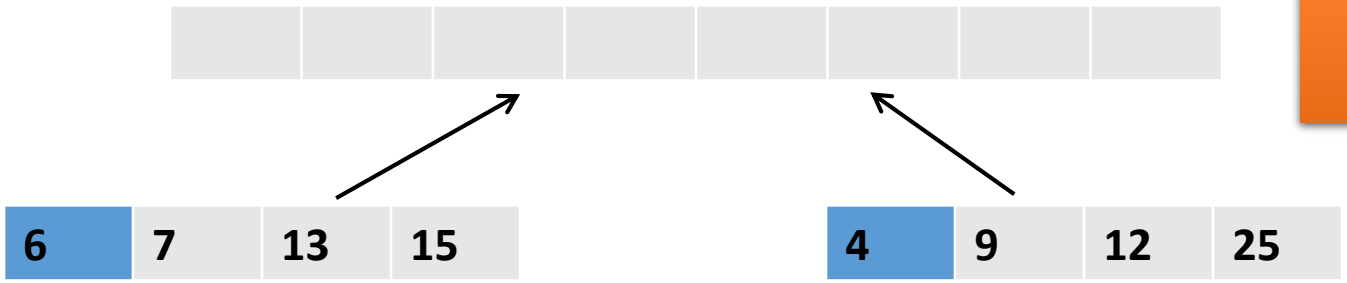
Schritt 2:
Rekursiv Sortieren

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



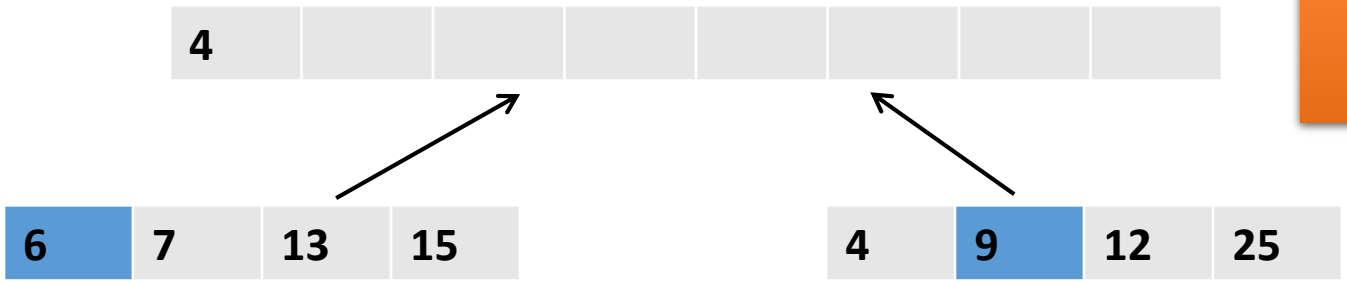
Schritt 3:
Zusammenfügen

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



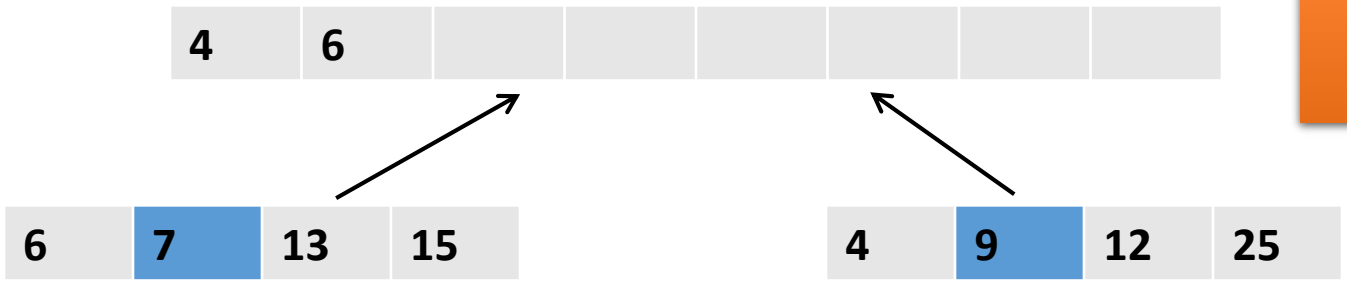
Schritt 3:
Zusammenfügen

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



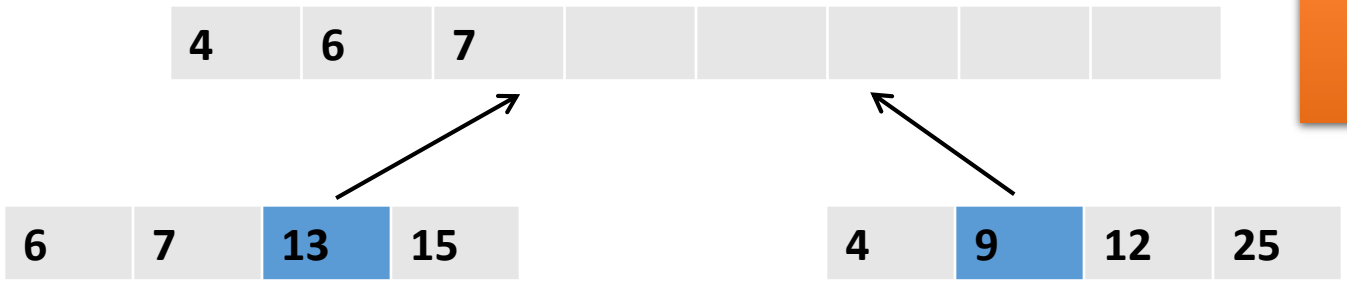
Schritt 3:
Zusammenfügen

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



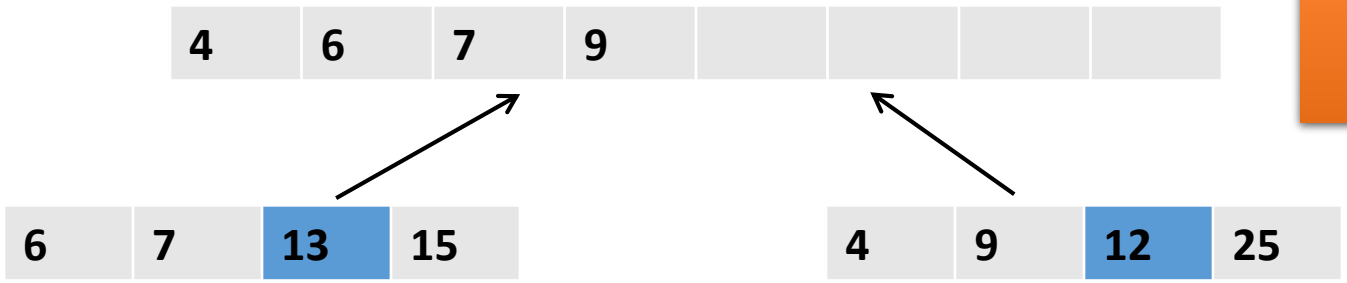
Schritt 3:
Zusammenfügen

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



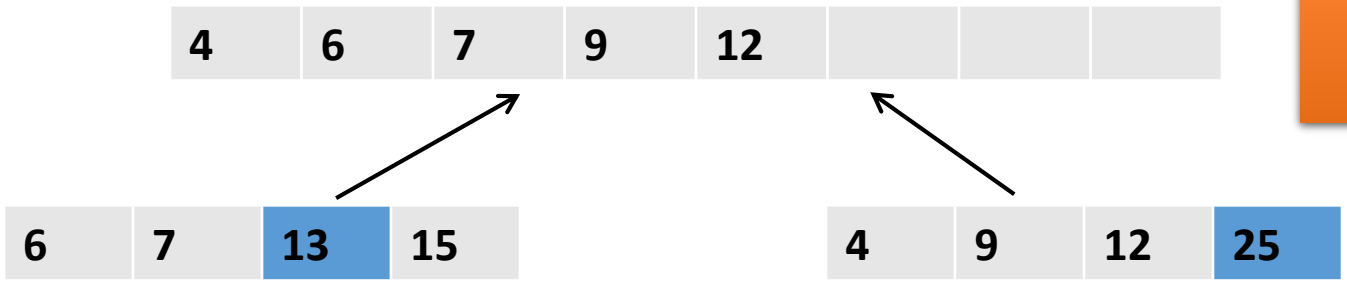
Schritt 3:
Zusammenfügen

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



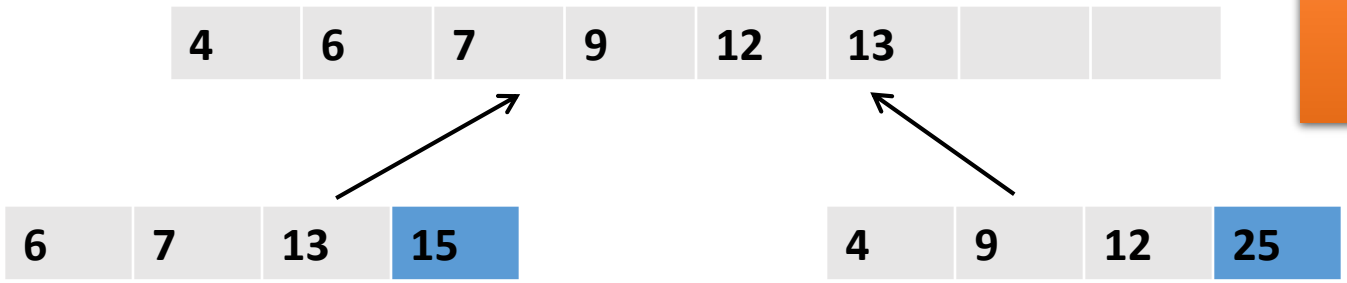
Schritt 3:
Zusammenfügen

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



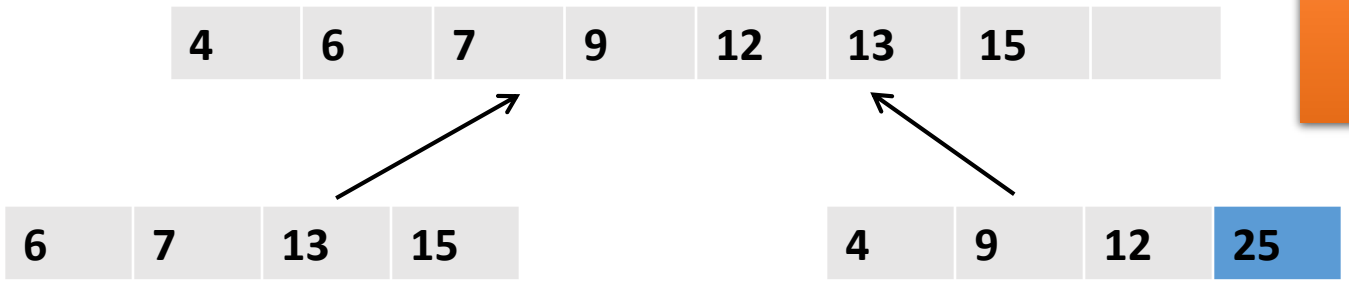
Schritt 3:
Zusammenfügen

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



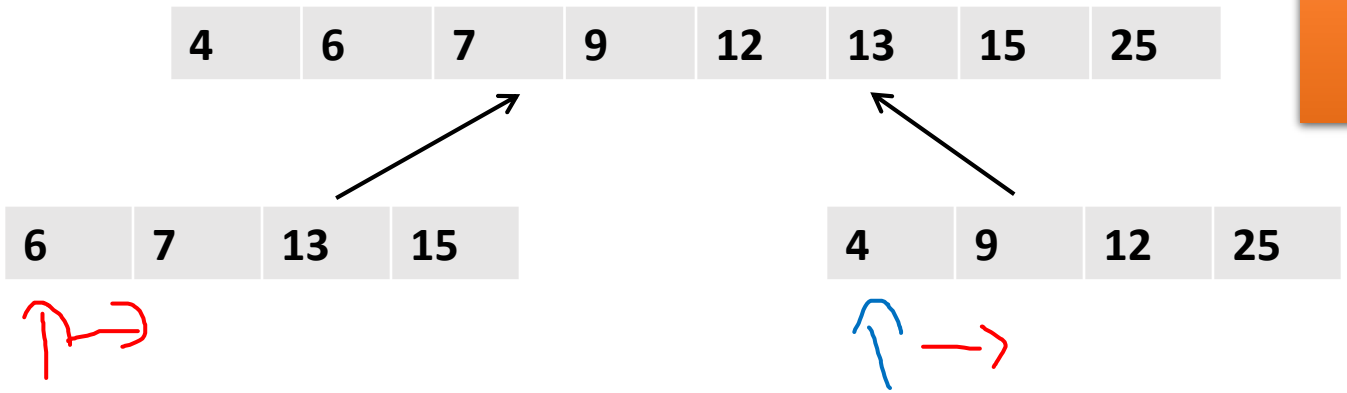
Schritt 3:
Zusammenfügen

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

- Beispiel: Sortieren durch Aufteilen in zwei Teile



Schritt 3:
Zusammenfügen

Algorithmenentwurf durch Rekursion

Teile & Herrsche Verfahren

- Idee: Teile die Eingabe in mehrere gleich große Teile auf
- Löse das Problem rekursiv auf den einzelnen Teilen
- Füge die Teile zu einer Lösung des Gesamtproblems zusammen

Wichtig

- Wir benötigen Rekursionsabbruch
- Beim Sortieren: Folgen der Länge 1

Algorithmenentwurf durch Rekursion

```
MergeSort(A,p,r)                    \* Sortiert A[p..r]
1. if p<r then                      \* Rekursionsabbruch, wenn p=r
2.    q=⌊(p+r)/2⌋                   \* Berechne die Mitte
3.    MergeSort(A,p,q)              \* Sortiere linke Teilhälfte
4.    MergeSort(A,q+1,r)            \* Sortiere rechte Teilhälfte
5.    Merge(A,p,q,r)                \* Füge die Teile zusammen
```

Aufruf des Algorithmus

- MergeSort(A,1,r), wobei r die Länge des Feldes A ist

Korrektheitsbeweis - Erweiterte Induktion

Induktion

- Sei A(n) eine Aussage über eine natürliche Zahl $n \in \mathbb{N}=\{1, 2, 3,\ldots\}$
- Wir wollen zeigen, dass die Aussage für alle natürlichen Zahlen gilt

Erweiterte Induktion

- Induktionsanfang: Aussage A(1) stimmt
- Induktionsschritt: Wenn A(1), A(2),…, A(n-1) gelten, dann gilt auch A(n)

Korrektheitsbeweis - MergeSort

Satz 5.4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis (1. Teil)

- Induktion über die Größe n des zu sortierenden Bereichs (d.h. n=r-p+1)
- Induktionsanfang (n=1):
- In diesem Fall enthält der zu sortierende Bereich A[p..r] ein Element (p=r)
- Dann ist der Bereich bereits sortiert
- Der Algorithmus tritt in Zeile 1 nicht in den **then-Fall** ein und endet ohne das Feld zu verändern
- Damit ist der Induktionsanfang korrekt

Korrektheitsbeweis - MergeSort

Satz 5.4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis (2. Teil)

- Induktionsannahme: MergeSort sortiert Bereiche der Größe m mit $1 \leq m < n$ korrekt
- Induktionsschluss:
- Wir betrachten MergeSort(A,p,r) mit n=r-p+1
- Da n>1 folgt p<r und der Algorithmus führt den **then**-Fall aus
- Hier wird q auf $\lfloor (p+r)/2 \rfloor$ gesetzt
- Es gilt $q \geq p$ und $q < r$

Korrektheitsbeweis - MergeSort

Satz 5.4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

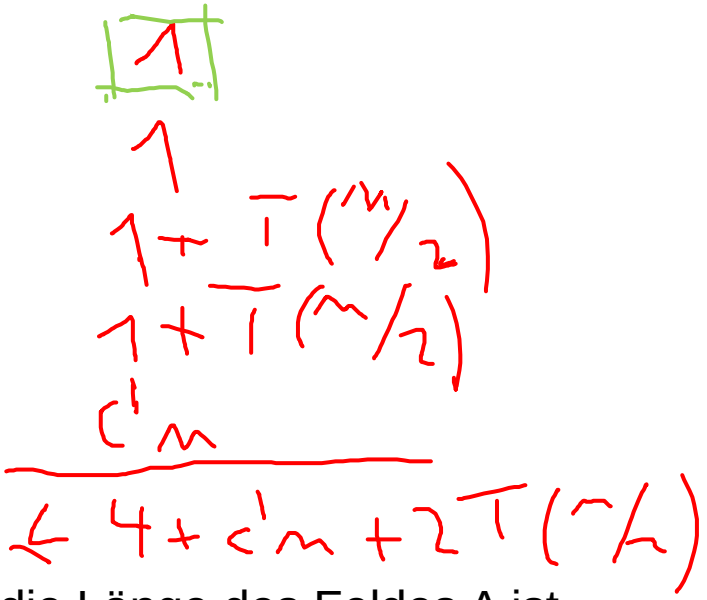
Beweis (2. Teil)

- Es gilt q≥p und q<r
- Dann wird MergeSort rekursiv in den Grenzen p,q bzw. q+1,r aufgerufen
- Nach Induktionsannahme sortiert MergeSort in diesem Fall korrekt
- Nun folgt die Korrektheit aus der Tatsache, dass Merge die beiden Bereiche korrekt zu einem sortierten Feld zusammenfügt

Laufzeitanalyse - MergeSort

MergeSort(A,p,r) * Sortiert A[p..r]

1. **if** p<r **then**
2. q=⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)



Aufruf des Algorithmus

- MergeSort(A,1,r), wobei r die Länge des Feldes A ist
- Sei T(n) die Worst-Case Laufzeit von MergeSort, um ein Teilarray der Größe n=r-p+1 zu sortieren

Laufzeitanalyse - MergeSort

Laufzeit als Rekursion

- $T(n) \leq \begin{cases} 1 & \text{, falls } n=1 \\ 2\,T(n/2) + cn & \text{, falls } n>1 \end{cases}$

$4 + c \cdot n + 2T(n/2)$

- Wobei c eine genügend große Konstante ist.

Zusammenfassung

Zusammenfassung

- Korrektheitsbeweise
  - Kommentare
  - Rekursionen
  - Beweis per Induktion

- Teile & Herrsche Verfahren
  - Sortieralgorithmus MergeSort
  - Eine erste Laufzeitrekursion

Referenzen

- T. Cormen, C. Leisserson, R. Rivest, C. Stein. Introduction to Algorithms. The MIT press. Second edition, 2001.