Teil V

Algorithmik

26. Knobelaufgabe #12: Geht's auch schneller?

Wir haben Listen und Arrays; sprechen die auch miteinander?

Du meinst, ob ich eine Liste in ein Array konvertiert kriege und umgekehrt? Das ist fix programmiert:

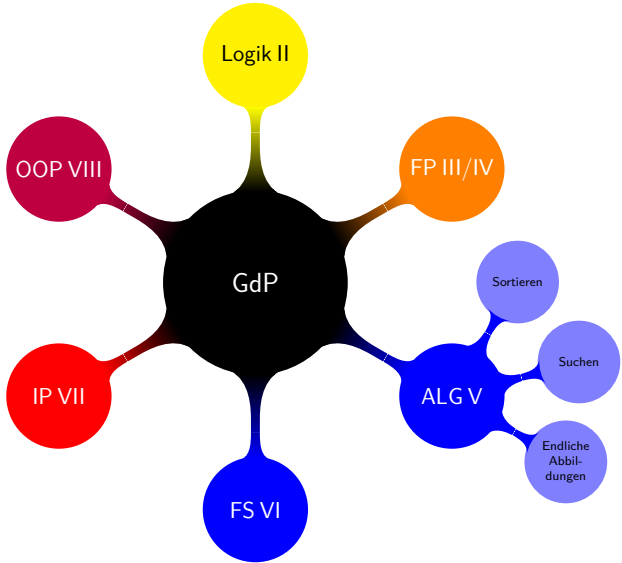
let *to-list* $(xs : 'a\ array) : 'a\ list = [\ $ ***for*** i ***in*** $0\,..\,xs.Length \div 1 \rightarrow xs.[i]\ ]$
let *to-array* $(xs : 'a\ list\ ) : 'a\ array = [|$ ***for*** i ***in*** $0\,..\,xs.Length \div 1 \rightarrow xs.[i]\ |]$

Nicht schlecht — man kann ***for*** auch verwenden, um eine Liste zu konstruieren, ...

Yup. *Mein Tipp:* nicht nur in den Vorlesungsfolien blättern, auch mal in der Online Doku stöbern ;-).

...und $xs.[i]$ um auf das i-te Listenelement zuzugreifen. Das ist aber teuer ...dein *to-array* hat eine quadratische Laufzeit!

26. Gliederung


Logik II
OOP VIII
FP III/IV
GdP
Sortieren
Suchen
IP VII
ALG V
Endliche Abbil-dungen
FS VI

26. Lernziele

Nach Durcharbeitung dieses Kapitels sollten Sie

- verschiedene Sortieralgorithmen kennen,
- elementare Suchstrukturen kennen,
- die Laufzeit einfacher Algorithmen abschätzen können,
- einfache Korrektheitsbeweise führen können,
- einfache Terminierungsbeweise führen können,
- das Konzept des „Abstrakten Datentyps“ verstanden haben,
- mit verschiedenen Programmiertechniken vertraut sein.

27. Algorithmisches Lösen von Problemen

- *Rechnen lassen:* mit der Existenz von Rechenmaschinen wird es interessant, Aufgaben in Rechenaufgaben zu verwandeln, die es von Natur aus nicht sind.
- Wie lassen sich Probleme systematisch lösen? Entwurfsmuster!
  - *Peano:* Problem für n wird auf das Problem für $n \dot{-} 1$ zurückgeführt. *Allgemeiner:* Problem der Größe n wird auf Problem der Größe $n \dot{-} 1$ zurückgeführt.
  - *Leibniz:* Problem für n wird auf das Problem für $n \div 2$ zurückgeführt. *Allgemeiner:* Problem der Größe n wird auf Probleme der Größe $n \div 2$ zurückgeführt.
  - *Struktur:* Problem für eine (Daten-) Struktur wird auf Probleme für Teilstrukturen zurückgeführt.
- Den Entwurfsmustern ist gemeinsam, dass sie Lösungen für Probleme aus Lösungen für „kleinere“ Probleme konstruieren.

27. Algorithmisches Lösen von Problemen

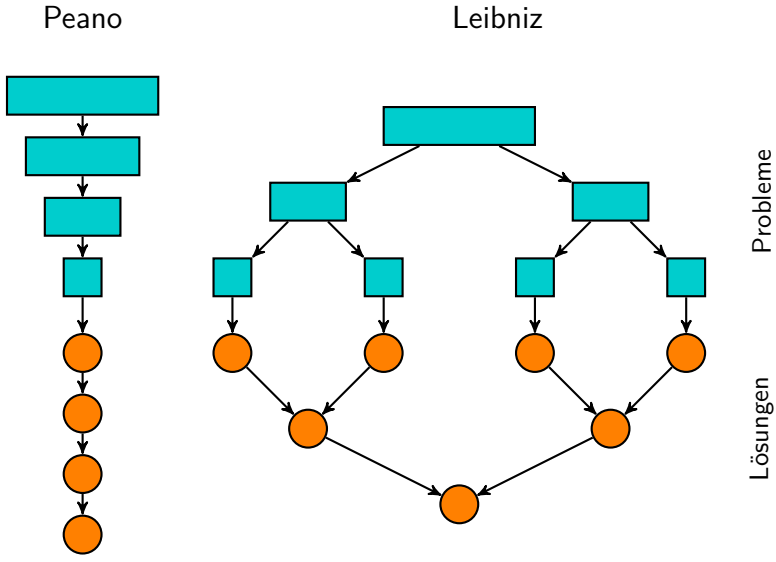
Allgemeiner Ansatz:

Um ein Problem zu lösen, genügt es zu zeigen, dass sich eine Lösung für jede Probleminstanz aus Lösungen kleinerer Probleminstanzen konstruieren lässt.

☞ Es ist nicht nötig, jede Probleminstanz von Grund auf zu lösen.

☞ Stattdessen versuche, eine Probleminstanz auf kleinere Probleminstanzen zu *reduzieren*.

27. Algorithmisches Lösen von Problemen



27. Sortieren

Computer manufacturers of the 1960s estimated that more than 25% of the running time on their computers was spent on sorting, when all their customers were taking into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either

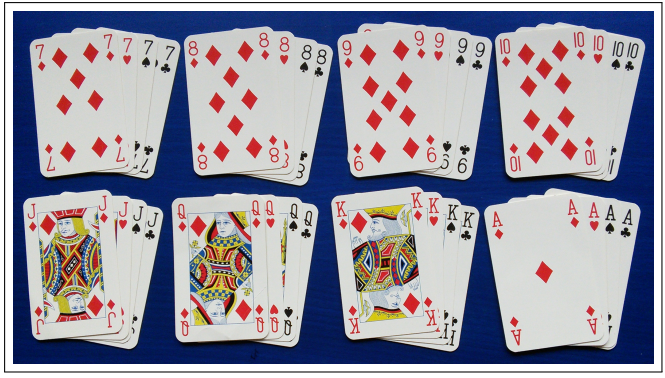
(i) *there are many important applications of sorting, or*
(ii) *many people sort when they shouldn't, or*
(iii) *inefficient sorting algorithms have been in common use.*

The real truth probably involves all three of these possibilities, but in any event we can see that sorting is worthy of serious study, as a practical matter.

— *Donald E. Knuth, TAOCP 3*

27. Beispiel: Spielkarten

27. Beispiel: Spielkarten



27. Beispiel: Musikstücke

- Gegeben:

Künstler	Titel	Länge	Genre	Popularität
Yes	Siberian Khatru	8:57	rock	119
Beatles	Helter Skelter	4:27	rock	147
Lorde	Royals	3:10	pop	2677
Genesis	Supper's Ready	22:53	rock	298
Kraftwerk	Autobahn	22:30	pop	346

- nach dem Künstler sortiert:

Künstler	Titel	Länge	Genre	Popularität
Beatles	Helter Skelter	4:27	rock	147
Genesis	Supper's Ready	22:53	rock	298
Kraftwerk	Autobahn	22:30	pop	346
Lorde	Royals	3:10	pop	2677
Yes	Siberian Khatru	8:57	rock	119

27. Beispiel: Musikstücke

- nach dem Genre sortiert (es gibt 12 mögliche Antworten, warum?):

Künstler	Titel	Länge	Genre	Popularität
Lorde	Royals	3:10	pop	2677
Kraftwerk	Autobahn	22:30	pop	346
Yes	Siberian Khatru	8:57	rock	119
Beatles	Helter Skelter	4:27	rock	147
Genesis	Supper's Ready	22:53	rock	298

- nach dem Genre und dann nach der Popularität sortiert:

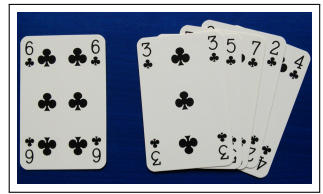
Künstler	Titel	Länge	Genre	Popularität
Lorde	Royals	3:10	pop	2677
Kraftwerk	Autobahn	22:30	pop	346
Genesis	Supper's Ready	22:53	rock	298
Beatles	Helter Skelter	4:27	rock	147
Yes	Siberian Khatru	8:57	rock	119

27. Sortieralgorithmen

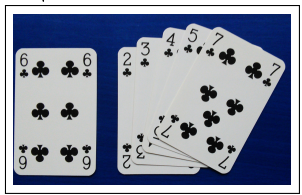
- *Peano Entwurfsmuster:* Problem der Größe n wird auf Problem der Größe $n-1$ zurückgeführt.
- *Sortieren:* die Problemgröße entspricht der Anzahl der zu sortierenden Elemente.
- *Sortieren durch Einfügen* (“bridge player method“):
  - Lege die erste Karte zur Seite,
  - sortiere den restlichen Stapel,
  - *füge* die erste Karte in den sortierten Stapel *ein*.
- Fokussiert auf die Eingabe.
- *Sortieren durch Auswählen*:
  - *Wähle* die kleinste Karte *aus*,
  - sortiere den restlichen Stapel,
  - lege die kleinste Karte auf den sortieren Stapel.
- Fokussiert auf die Ausgabe.

27. Sortieren durch Einfügen

lege erste
Karte zur Seite



sortiere Reststapel



füge erste
Karte ein

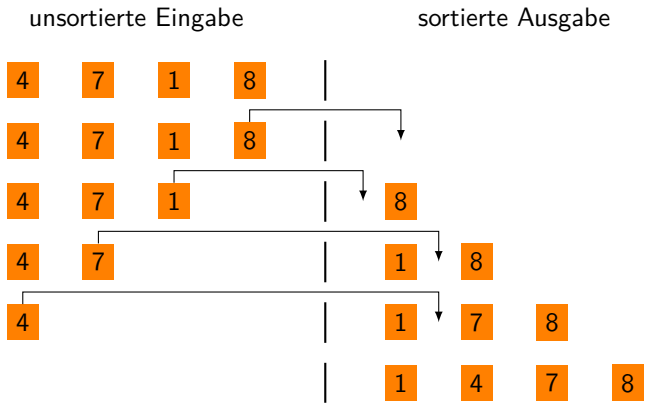
27. Sortieren durch Einfügen

Sortieren durch Einfügen in F#:

```
let insertion-sort (⩽) =
  let rec insert k = function
      | []      → [k]
      | x :: xs → if k ⩽ x then k :: x :: xs
                           else x :: insert k xs
  let rec sort = function
      | []      → []
      | x :: xs → insert x (sort xs)
  in sort
```

☞ Wir abstrahieren von der Ordnungsrelation $\leqslant$.

27. Sortieren durch Einfügen: Korrektheit



Zu zeigen:

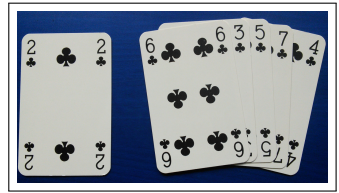
Wenn *xs* sortiert ist, dann ist auch *insert k xs* sortiert.

27. Sortieren durch Einfügen: Laufzeit

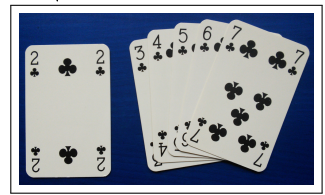
- Sortieren durch Einfügen ist *adaptiv:* „vorsortierte“ Eingaben werden schneller verarbeitet.
- *Der beste Fall:* die Eingabe ist bereits aufsteigend sortiert.
  - Einfügen: lediglich ein Vergleich wird benötigt.
  - Sortieren: benötigt insgesamt $n - 1$ Vergleiche (optimal).
- *Der schlechteste Fall:* die Eingabe ist absteigend sortiert.
  - Einfügen: durchläuft die gesamte Liste und benötigt $i - 1$ Vergleiche.
  - Sortieren: benötigt insgesamt $\sum_{i=0}^{n-1} i = n \cdot (n-1)/2$ Vergleiche.
- Kurz: die Laufzeit ist quadratisch.

27. Sortieren durch Auswählen

lege kleinste
Karte zur Seite



sortiere Reststapel



füge kleinste
Karte hinzu

27. Sortieren durch Auswählen

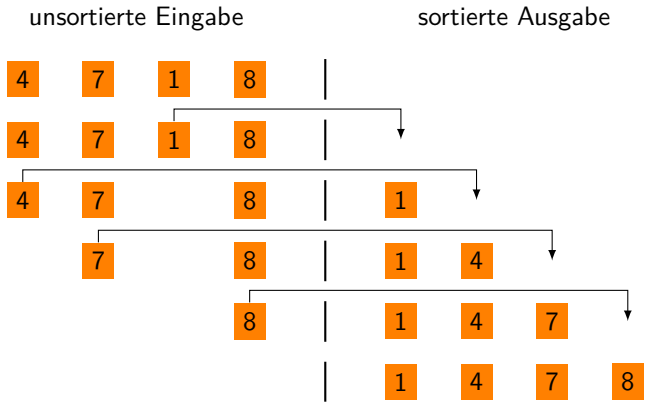
Sortieren durch Auswählen in F#:

```
let selection-sort (⩽) =
  let rec split-min = function
      | []      → None
      | x :: xs → Some (match split-min xs with
                          | None        → (x, xs)
                          | Some (m, ys) → if x ⩽ m then (x, xs)
                                           else (m, x :: ys))
  let rec sort xs =
    match split-min xs with
      | None        → []
      | Some (m, ys) → m :: sort ys
  in sort
```

Der Aufruf *split-min xs* gibt das kleinste Element von *xs* und die restliche Liste zurück (sofern ein kleinstes Element existiert).

☞ Die Definition von *sort* folgt *nicht* dem Struktur Entwurfsmuster: *ys* ist im allgemeinen nicht die Restliste von *xs*.

27. Sortieren durch Auswählen: Korrektheit



Zu zeigen:

Wenn *xs* nicht leer ist, dann ist *split-min xs* = *Some* (a, ys), wobei *a* das kleinste Element von *xs* ist und *ys* die restliche Liste. Anderenfalls ist *split-min xs* = *None*.

27. Sortieren durch Auswählen: Laufzeit

- Sortieren durch Auswählen ist „vergesslich“ (engl. oblivious): die Laufzeit hängt nicht von den Eingabeelementen ab.
- *Bester und schlechtester Fall:*
  - Auswahl: durchläuft die gesamte Liste und benötigt $i - 1$ Vergleiche (optimal, warum?).
  - Sortieren: benötigt insgesamt $\sum_{i=0}^{n-1} i = n \cdot (n-1)/2$ Vergleiche.
- Kurz: die Laufzeit ist quadratisch.

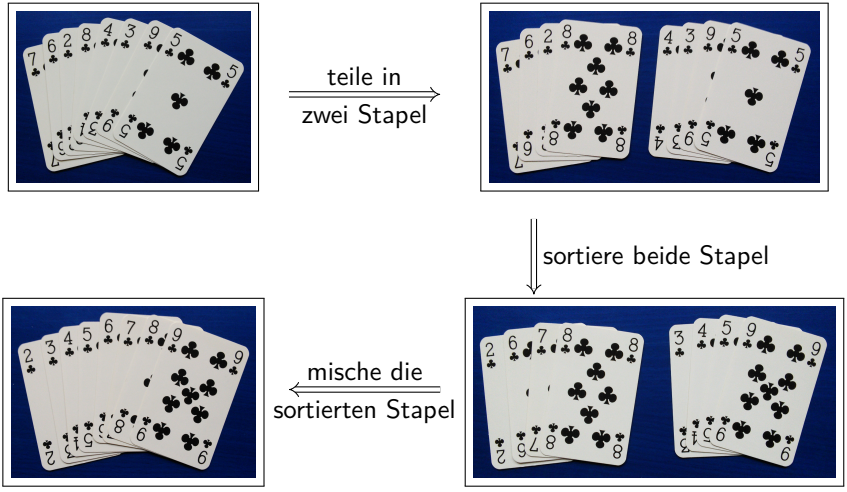
Writing programs needs genius to save the last order or the last millisecond. It is great fun, but it is a young man's game. You start it with great enthusiasm when you first start programming, but after ten years you get a bit bored with it, and then you turn to automatic-programming languages and use them because they enable you to get to the heart of the problem that you want to do, instead of having to concentrate on the mechanics of getting the program going as fast as you possibly can, which is really nothing more than doing a sort of crossword puzzle.

— *Christopher Strachey*

27. Teile und Herrsche

- *Leibniz:* Problem der Größe n wird auf Probleme der Größe $n \div 2$ zurückgeführt.
- *Sortieren durch Mischen:* (engl. sorting by merging)
  - Teile den Eingabestapel in zwei ungefähr gleich große Stapel,
  - sortiere jeden der Teilstapel,
  - „*mische*“ die zwei sortierten Teilstapel.
- (Ein alternativer Ansatz ist *Sortieren durch Austauschen:*
  - Wähle eine Pivotkarte aus und teile den Eingabestapel in kleinere und größere Karten,
  - sortiere jeden der Teilstapel,
  - hänge die beiden sortierten Teilstapel aneinander.
- Auch bekannt unter dem Namen *Quicksort.*)

27. Sortieren durch Mischen



27. Sortieren durch Mischen

Teilaufgabe: eine Liste halbieren.

Das Struktur Entwurfsmuster führt unmittelbar zum Ziel:

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{unzip} = \textbf{\textit{function}} \\ \quad \mid [] \quad\ \ \to ([], []) \\ \quad \mid x :: xs \to \textbf{\textit{let}}\ (xs_1, xs_2) = \mathit{unzip}\ xs \\ \qquad\qquad\quad\ \ (x :: xs_2, xs_1)\end{array}$$

Der Aufruf *unzip xs* teilt die Liste *xs* in die Liste der Elemente an geraden und die Liste der Elemente an ungeraden Positionen.

27. Sortieren durch Mischen

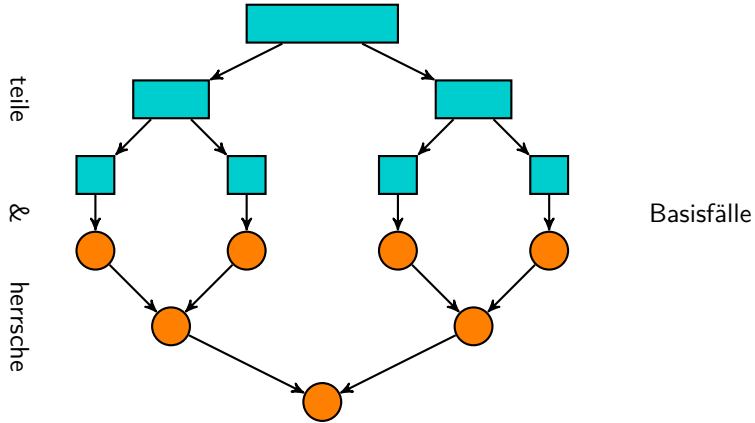
Sortieren durch Mischen in F#:

```
let merge-sort (⩽) =
  let rec merge = function
    | ([], xs) | (xs, []) → xs
    | (x :: xs, y :: ys)  →
      if x ⩽ y then x :: merge (xs, y :: ys)
               else y :: merge (x :: xs, ys)
  let rec sort = function
    | []   → []
    | [x]  → [x]
    | xs   → let (xs1, xs2) = unzip xs
             merge (sort xs1, sort xs2)
  in sort
```

☞ *merge* verallgemeinert *insert* ($A \cup B$ versus $\{a\} \cup B$).

27. Teile und Herrsche: Laufzeit

- Leibniz Entwurfsmuster (teile und herrsche, divide et impera):
- *Fall* $n < 2$*:* ad hoc.
- *Fall* $n \geq 2$*:* teile in Probleme der Größen $\lfloor n/2 \rfloor$ und $\lceil n/2 \rceil$ auf.



- Wie tief ist der „Rekursionsbaum“? Mit anderen Worten, wie oft lässt sich n halbieren, bis wir 1 erhalten?

27. Teile und Herrsche: Laufzeit

- *Frage:* wie oft lässt sich n halbieren, bis wir 1 erhalten?
- *Antwort:* der binäre Logarithmus von n.

$$\lg n = \log_2 n$$

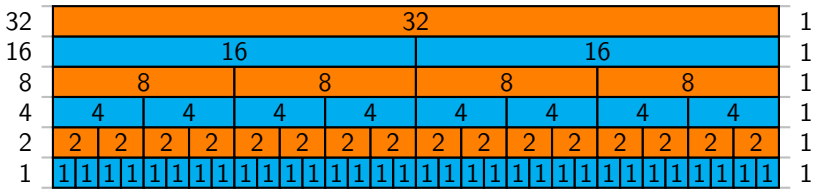
- Wachstum von $\lg n$:

n	$\lg n$
100	$\approx 6,6$
1.000	$\approx 10,0$
10.000	$\approx 13,3$
100.000	$\approx 16,6$
1.000.000	$\approx 20,0$

- Zur Erinnerung: $\lg(ab) = \lg a + \lg b$, zum Beispiel ist $\lg 1.000.000 = 2 \cdot \lg 1.000 \approx 20,0$.

27. Sortieren durch Mischen: Laufzeit

- Die Laufzeit von *unzip* ist proportional zur Eingabegröße.
- Die Laufzeit von *merge* ist proportional zur Ausgabegröße.
- Sortieren durch Mischen: die Laufzeit ergibt sich als Produkt der Rekursionstiefe ($\lg n$) und des Aufwands pro Rekursionsebene (n).



- Die Laufzeit von *merge-sort* beträgt entsprechend $n \lg n$.

27. Sortieren durch Mischen: Laufzeit

- Wachstum von $n \lg n$:

n	$\lg n$	$n \lg n$	n^2
100	$\approx 6,6$	≈ 660	10.000
1.000	$\approx 10,0$	≈ 10.000	1.000.000
10.000	$\approx 13,3$	≈ 133.000	100.000.000
100.000	$\approx 16,6$	$\approx 1.660.000$	10.000.000.000
1.000.000	$\approx 20,0$	$\approx 20.000.000$	1.000.000.000.000

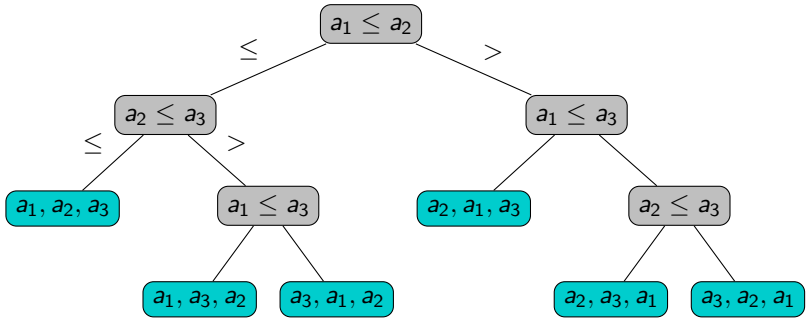
- Für hinreichend große n ist Sortieren durch Mischen wesentlich schneller als Sortieren durch Einfügen oder Auswählen.

27. Sortieren: Problemkomplexität

- Sortieren durch Mischen hat eine Laufzeit von $n \lg n$.
- Geht's noch schneller? Können wir in linearer Zeit sortieren?
- Leider nein: jedes Sortierverfahren, *das auf dem Vergleichen von Elementen basiert*, benötigt im schlechtesten Fall $n \log n$ Vergleiche.
- Damit ist *merge-sort asymptotisch optimal*.
- Die *Komplexität* des Sortierproblems ist $n \log n$.

27. Sortieren: Problemkomplexität

- Ein Entscheidungsbaum, um drei Elemente zu sortieren:



- Eine Ausführung des korrespondieren Sortierprogramms entspricht einem Pfad von der Wurzel zu einem Blatt, der sortierten Permutation der Eingabe.
- Ein Entscheidungsbaum, der n Elemente sortiert, muss notwendigerweise $n!$ Blätter besitzen.
- Man kann zeigen, dass die Höhe des Baums ungefähr $n \log n$ ist.

27. Untere und obere Schranken