Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung
Informatik I
Wintersemester 2006

Kapitel 1: Einführung und formale Grundlagen

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Elke Achtert, Arthur Zimek

http://www.dbs.ifi.lmu.de/Lehre/Info1

Database Systems

Inhalt

1. Was ist Informatik?
2. Formale Grundlagen: Werte, Mengen, Relationen, Funktionen
3. Algorithmen

Überblick

1. Was ist Informatik?
2. Formale Grundlagen: Werte, Mengen, Relationen, Funktionen
3. Algorithmen

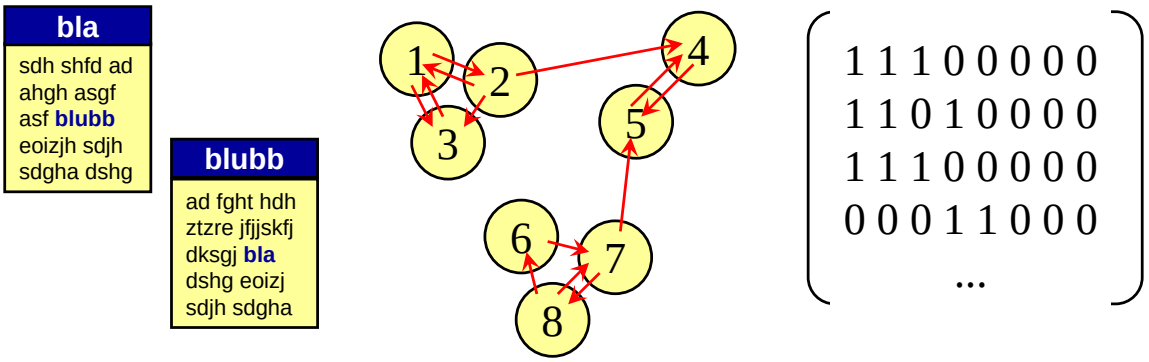
Was ist Informatik?

- Informatik (engl. Computer Science, auch Informatics): Wissenschaft von der systematischen Verarbeitung von Informationen, insbesondere der automatischen Verarbeitung mit Hilfe von Rechenanlagen
- Primäres Lernziel sind also nicht die Komponenten eines Computers oder die Sprachen zu seiner Programmierung, sondern die *Prinzipien und Techniken zur Verarbeitung von Information*
- Wort-Herkunft:
  Abgeleitet aus *Information* und *Mathematik*
  oder *Information* und *Automatik*

Informatik und Mathematik

- Viele Arten von Informationen können mit Mitteln aus der Mathematik dargestellt und verarbeitet werden.
- Beispiel: Das WWW.

Dokument: Link-Graph: Adjazenz-Matrix:



- Mathematische Strukturen sind Grundlage der Informatik

Teilgebiete der Informatik

- Theoretische Informatik
- Praktische Informatik
- Technische Informatik

- Angewandte Informatik
- Informatik und Gesellschaft

Theoretische Informatik

- Automatentheorie:
  Untersucht meist sehr einfache, gedachte Modelle von Rechenmaschinen
- Berechenbarkeits-Theorie:
  Welche Arten von Aufgaben können von welchen Automaten bearbeitet werden?
  Welche Aufgaben können gar nicht allgemein gelöst werden („unentscheidbar“, z.B. die Frage, ob ein vorgegebenes Programm auf einer beliebigen Eingabe endlos laufen wird)

Theoretische Informatik

- Komplexitätstheorie:
  Wie groß ist der Aufwand (benötigte Rechenzeit und benötigter Speicher) für die Berechnung in Abhängigkeit von der Größe der gestellten Aufgabe (z.B. beim Sortieren: Wie viele Elemente sind zu sortieren)
- Logik
- Graphentheorie
- Kryptologie
- usw.

Technische Informatik

- Rechnerarchitektur:
  Zusammenspiel zwischen Prozessor, Arbeitsspeicher und weiteren Bausteinen eines Rechners. Aus welchen Komponenten soll der Prozessor zusammengesetzt sein?
- Rechnerkommunikation:
  Verfahren zum Datenaustausch zwischen Rechnern
- Verteilte Systeme:
  Verfahren, damit mehrere Rechner gemeinsam an einer Aufgabenstellung arbeiten können

Praktische Informatik

- Algorithmen und Datenstrukturen:
  Wie können die Daten organisiert werden, damit Standardaufgaben effizient lösbar sind?
- Programmierung und Software-Technik:
  Systematische Erstellung von Software
  Konzepte für den SW-Entwicklungsprozess von der Idee bis zum fertigen System
  Organisation großer Software-Projekte
- Betriebssysteme
- Datenbanksysteme

Überblick

1. Was ist Informatik?
2. Formale Grundlagen: Werte, Mengen, Relationen, Funktionen
3. Algorithmen

Modellierung

- Eine zentrale Aufgabe der Informatik ist die Modellierung von Aspekten der realen Welt.
- Damit werden konkrete Zusammenhänge oder Aufgaben abstrahiert.
- Ziel einer Abstraktion ist z.B. die Automatisierung eines Vorgangs oder die Verallgemeinerung einer Lösung für ein größeres Spektrum von Anwendungen.
- In dieser Vorlesung betrachten wir Typen und Funktionen als Modellierungsmethoden und untersuchen deren Eigenschaften.

Wertebereiche

- Beispiel: *Ziehe drei Karten aus einem Kartenspiel. Bestimme die höchste der drei Karten.*
- Zur Präzisierung der Aufgabe charakterisieren wir den Wertebereich der Karten: Jede Karte wird durch zwei Eigenschaften beschrieben: Die „Farbe“ (Kreuz, Pik, Herz oder Karo) und den „Wert“ (7, 8, 9, 10, Bube, Dame, König, Ass). Alle Kombinationen dieser beiden Wertebereiche bilden den Wertebereich eines Kartenspiels.

Abstraktion: Mengen

- Eine brauchbare Abstraktion, um mit Wertebereichen umzugehen, finden wir in der Mengentheorie.
- Hier genügt eine informelle Definition:

> Eine Menge *M* ist eine Zusammenfassung von verschiedenen Objekten, den Elementen der Menge.
> Die Notation $a \in M$ bedeutet: *a* ist ein Element der Menge *M*.

Angabe einer Menge

- Eine Menge kann auf zwei Arten angegeben werden:
- *extensional* (durch Aufzählung)
  - Beispiel: {1, 4, 9, 16, 25}
  - nur zur Angabe von endlichen Mengen geeignet
- *intensional* (durch Angabe einer Bedingung – alle Werte, die diese Bedingung erfüllen, und nur diese Werte, sind Elemente der Menge)
  - Beispiel: $\{a \mid a \in \mathbf{N}, a \text{ ist Quadratzahl und } a < 30\}$
- Eine Menge kann beliebig viele Elemente enthalten oder auch gar keine (leere Menge, geschrieben {} oder ∅).

Eigenschaften von Mengen

- Alle Elemente einer Menge sind verschieden.
- Man kann eine Menge {1, 2, 2, 3} angeben, dies ist aber redundant. {1, 2, 3} ist die gleiche Menge.
- Mit einer Menge kann man also nicht mehrfaches Vorkommen eines gleichen Wertes modellieren, dazu benötigen wir andere Konzepte. Man nennt solche Strukturen **Multimengen**. Sie werden wie Mengen geschrieben, haben aber andere Eigenschaften und Rechenregeln.
- Die Elemente in einer Menge sind nicht geordnet.
  - {1, 2, 3} = {1, 3, 2}

Eigenschaften von Mengen (2)

- Mengen können aus einfachen („atomaren“) oder zusammengesetzten Werten gebildet sein:
  - Spielkarten: {(Pik, 10), (Herz, Dame)}
- Mengen können Elemente einer Menge sein:
  - {{1}, {1, 4}, ∅}
- Eine Menge kann verschiedenartige oder unterschiedlich strukturierte Elemente enthalten:
  - {1, (Pik, 10), {1, 3}, 9}

Formale Eigenschaften und Operationen auf Mengen

Bezeichnung	*Notation*	*Bedeutung*
ist Teilmenge	$M \subseteq N$	*aus* $a \in M$ *folgt* $a \in N$
ist echte Teilmenge	$M \subset N$	$M \subseteq N$ *und* $M \neq N$
Vereinigung	$M \cup N$	$\{x \mid x \in M \text{ oder } x \in N\}$
Durchschnitt	$M \cap N$	$\{x \mid x \in M \text{ und } x \in N\}$
Differenz	$M \backslash N$	$\{x \mid x \in M \text{ und } x \notin N\}$
disjunkt	$M \cap N = \emptyset$	*M und N haben keine gemeinsamen Elemente*
Kardinalität	$\lvert M \rvert$	*Anzahl der Elemente von M*

Entscheidbarkeit in der intensionalen Beschreibung

- Mit der intensionalen Beschreibung kann man Mengen definieren, für die die Frage „gilt $a \in M$?“ prinzipiell nicht entscheidbar ist (Russels Paradoxon):
  - $P := \{x \mid x \notin x\}$ *(Menge aller Mengen, die sich nicht selbst als Element enthalten)*
  - Widerspruch: *„Ist P ein Element von P?“*
  - Problem: P gehört zum Wertebereich, aus dem die Elemente von P stammen.
- Lösung: Angeben, aus welchem größeren, bereits definierten Wertebereich die Elemente stammen.
  - Beispiel: $\{a \mid a \in \mathbf{N}$, a ist Quadratzahl und $a < 30\}$

Entscheidbarkeit und Rekursion

- $M := \{a \mid$ *Bedingung für a ist erfüllt*$\}$
- Damit entschieden werden kann, welche Elemente zu M gehören, muss die Bedingung in der intensionalen Beschreibung entscheidbar sein.
- Möglich ist aber, Mengen *rekursiv* zu beschreiben, also unter Rückgriff auf sich selbst.
  - $Sonnensystem := \{Sonne\} \cup$
    $\{x \mid x \in Himmelskörper,$
    x *umkreist* y *und* $y \in Sonnensystem\}$

Geordnetes Paar, kartesisches Produkt

- Bildung zusammengesetzter Werte: Setze mehrere Elemente aus jeweils einem Wertebereich zu einem Tupel zusammen:

Geordnetes Paar, kartesisches Produkt
Ein geordnetes Paar *(x, y)* besteht aus zwei Werten *x* und *y*, wobei *x* die erste und *y* die zweite Komponente ist.
Das kartesische Produkt $M \times N$ zweier Mengen *M* und *N* ist die Menge aller geordneten Paare mit erster Komponente aus *M* und zweiter Komponente aus *N*.

$M \times N := \{(x, y) \mid x \in M \text{ und } y \in N\}$

n-Tupel, allgemeines kartesisches Produkt

- Verallgemeinerung auf beliebige Anzahl *n* von Mengen:

***n*-Tupel, kartesisches Produkt von *n* Mengen**
Das allgemeine kartesische Produkt über *n* Mengen liefert die Menge aller geordneten *n*-Tupel mit den Komponenten aus diesen Mengen:
$M_1 \times M_2 \times \ldots \times M_n := \{(a_1, a_2, \ldots, a_n) \mid a_i \in M_i \text{ und } i \in I\}$
mit Indexmenge $I := \{1, \ldots, n\}$
Sind alle Mengen identisch, kann man für $M \times M \times \ldots \times M$ auch schreiben M^n.

- KartenSpiel := KartenArten × KartenSymbole
  - KartenArten := {Kreuz, Pik, Herz, Karo}
  - KartenSymbole := {7, 8, 9, 10, Bube, Dame, König, Ass}

Potenzmenge

- Viele Objekte werden durch Mengen beschrieben.
- Der Wertebereich ist dann eine Menge, die Mengen enthält. Spezielle Form: Potenzmenge

Potenzmenge
Die Potenzmenge einer Grundmenge U ist die Menge aller Teilmengen von U, geschrieben $\wp(U)$.
$\wp(U) := \{M \mid M \subseteq U\}$

- Beispiel: $U := \{d, f, s\}$
  - $\wp(U) = \{\varnothing, \{d\}, \{f\}, \{s\}, \{d, f\}, \{d, s\}, \{f, s\}, \{d, f, s\}\}$
- Kardinalität einer Potenzmenge?

$|U| = n \Rightarrow |\wp(U)| = 2^n$

Relationen

- Eine Relation kennen Sie als Prädikat, z. B. $a < b$.
- Eine Relation lässt sich auch auffassen als eine Menge von Tupeln
  - z.B.: $< \subseteq N \times N$
  - $(1, 2) \in <$ $\quad (2, 1) \notin <$
- Eine Relation R ist erfüllt (oder wahr) für **alle** Tupel a mit $a \in R$ und **nur** für diese Tupel.
  - man schreibt auch: $R\ a$
- Für zweistellige Relationen schreibt man auch $x\ R\ y$
  - (z.B. $x < y$)

n-stellige Relationen

Relation
Eine n-stellige Relation R ist eine Menge von n-Tupeln, wobei jedes davon aus einem Wertebereich $M_1 \times M_2 \times \ldots M_n$ mit $n > 1$ stammt, d.h. $R \subseteq M_1 \times M_2 \times \ldots M_n$.

Der Wertebereich, aus dem n-stellige Relationen stammen, ist die Potenzmenge über $M_1 \times M_2 \times \ldots \times M_n$.

Eigenschaften 2-stelliger Relationen

Eigenschaften 2-stelliger Relationen
Für 2-stellige Relationen $R \in \wp(M \times M)$ sind u. a. folgende Eigenschaften definiert:
reflexiv, wenn für alle $x \in M$ gilt: $x\ R\ x$;
symmetrisch, wenn für alle $x, y \in M$ gilt: aus $x\ R\ y$ folgt $y\ R\ x$;
antisymmetrisch, wenn für alle $x, y \in M$ gilt:
aus $x\ R\ y$ und $y\ R\ x$ folgt $x = y$;
transitiv, wenn für alle $x, y, z \in M$ gilt:
aus $x\ R\ y$ und $y\ R\ z$ folgt $x\ R\ z$;
alternativ, wenn für alle $x, y \in M$ gilt: $x\ R\ y$ oder $y\ R\ x$.

Ordnungsrelationen

Ordnungsrelationen
Eine 2-stellige Relation $R \in \wp(M \times M)$ ist eine
partielle Ordnung, wenn R reflexiv,
antisymmetrisch und transitiv ist;
totale Ordnung, wenn R eine alternative partielle Ordnung ist.

- Sie kennen die Ordnungen $<$ und $\leq$ auf den ganzen Zahlen – sind diese Ordnungen total?
- KartenSpiel := KartenArten × KartenSymbole
  - Ordnung z. B.: A ≤ B ⇔ A.Symbol < B.Symbol oder
    A.Symbol = B.Symbol und
    A.Art ≤ B.Art

Funktionen

- Eine Funktion bildet Werte aus ihrem Definitionsbereich auf Werte ihres Bildbereiches ab.
- Funktionen sind Relationen mit speziellen Eigenschaften (setzen Werte des Definitions- und des Bildbereiches *rechtseindeutig* zueinander in Beziehung).

Funktion
Eine Funktion f ist eine 2-stellige Relation $f \subseteq D \times B$, für die gilt: Aus $(x, y) \in f$ und $(x, z) \in f$ folgt $y = z$, d.h. einem Wert aus D ist höchstens ein Wert aus B zugeordnet.
Die Menge D heißt Definitionsbereich und die Menge B heißt Bildbereich der Funktion.

Schreibweisen: $(x, y) \in f \Leftrightarrow y = f(x) \Leftrightarrow f(x) = y$

Wertebereich und Signatur von Funktionen

- Als spezielle Relationen stammen Funktionen aus Wertebereichen, die Teilmengen entsprechend strukturierter Relationen sind.

Wertebereich von Funktionen
Der Wertebereich $D \to B$ ist die Menge aller Funktionen, die von D nach B abbilden: Es gilt: $D \to B \subseteq \wp(D \times B)$.
$D \to B$ enthält als Elemente alle Mengen von Paaren über $D \times B$, die Funktionen sind. Für eine Funktion $f \in D \to B$ gilt: $f \subseteq D \times B$.
Für $f \in D \to B$ sagt man:
f hat die **Signatur** $D \to B$
oder: $f : D \to B$

Signatur ist ein zentrales Konzept in der Spezifikation von Programmen

Eigenschaften von Funktionen

Eigenschaften von Funktionen
Eine Funktion $f : D \to B$ ist
total, wenn es für jedes $x \in D$ ein Paar $(x, y) \in f$ gibt;
surjektiv, wenn es zu jedem $y \in B$ ein Paar $(x, y) \in f$ gibt;
injektiv, wenn es zu jedem $y \in B$ höchstens ein Paar $(x, y) \in f$ gibt;
bijektiv, wenn f zugleich surjektiv und injektiv ist.

- Eigenschaften von (linearen) Funktionen werden genauer in der Vorlesung „Lineare Algebra“ untersucht.
- Man sagt auch, eine Funktion ist **partiell**, wenn es gleichgültig ist, ob sie total ist oder nicht. Wenn ein Mathematiker nicht angibt, ob eine Funktion total oder partiell ist, meint er in der Regel eine totale Funktion. Für einen Informatiker ist die partielle Funktion der Normalfall.

Stelligkeit von Funktionen

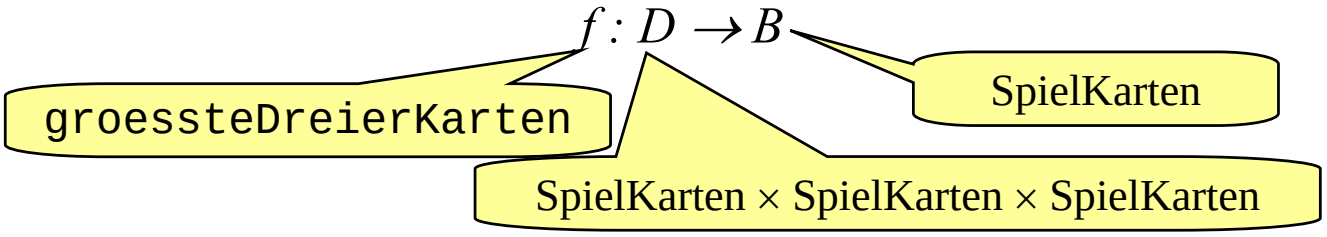
- Funktionen aus dem Wertebereich $D \rightarrow B$ sind n-stellig, wenn der Definitionsbereich D eine Menge von n-Tupeln ist.
- Ist D nicht als kartesisches Produkt strukturiert, so sind die Funktionen aus $D \rightarrow B$ 1-stellig, wenn D nicht leer ist, und 0-stellig, wenn D leer ist.
- 0-stellige Funktionen sind **konstante Funktionen**, kurz **Konstanten**, für jeweils einen Wert aus B.

Beispiele

- Wir können nun genauer sagen, was ein Prädikat ist: eine Funktion aus $M \rightarrow$ {wahr, falsch}.
  - Die Menge {wahr, falsch} oder {true, false} ist die Menge der boole'schen Werte, oder kurz **bool**.
  - z. B. $\leq \in Z \times Z \rightarrow$ bool
- Um die höchste von 3 Karten zu finden, benötigen wir eine Maximumsfunktion, basierend auf einer Ordnung, die ein Paar von Karten auf die höhere der beiden Karten abbildet:
  - `max` : SpielKarten $\times$ SpielKarten $\rightarrow$ SpielKarten

Beispiel

- Die Funktion zur Bestimmung der höchsten von drei Karten lässt sich durch wiederholtes Anwenden der `max`-Funktion definieren:
  - `groessteDreierKarten((k1, k2, k3)) = max(k1, max(k2, k3));`
- Diese Funktion lässt sich nur auf 3-Tupel (Tripel) von Karten anwenden. Signatur:



Überblick

1. Was ist Informatik?
2. Formale Grundlagen: Werte, Mengen, Relationen, Funktionen
3. Algorithmen

Begriff „Algorithmus“

- Das Wort „Algorithmus“ leitet sich von dem arabischen Namen eines bedeutenden persischen Mathematikers des 9. Jahrhunderts, *Muhammad ibn Musa al-Chwarizmi* ab.
- Allgemein meint man mit Algorithmus eine möglichst genaue Handlungsvorschrift, um ausgehend von bestimmten Vorbedingungen ein bestimmtes Ziel zu erreichen.
  - Kochrezept
  - Gebrauchsanleitung
  - Waschmaschinenprogramm

Beginn einer lateinischen Ausgabe des Lehrbuchs über das Rechnen mit indischen Ziffern: *Dixit algorizmi*

Definition Algorithmus

Algorithmus
Ein Algorithmus ist ein Verfahren mit einer *präzisen, endlichen Beschreibung* unter Verwendung *effektiver, elementarer Verarbeitungsschritte.*

- **präzise, endliche Beschreibung:** in einem endlichen Text in einer eindeutigen Sprache genau festgelegte Abfolge von Schritten
- **effektiver Verarbeitungsschritt:** Jeder einzelne Schritt ist tatsächlich ausführbar, benötigt z. B. zu jedem Zeitpunkt der Ausführung nur endlich viel Speicherplatz.
- **elementarer Verarbeitungsschritt:** Jeder Schritt ist eine Basis-Operation oder selbst durch einen Algorithmus spezifiziert.

Eigenschaften von Algorithmen

Ein Algorithmus heißt

- **terminierend**, wenn er für alle zulässigen Schrittfolgen stets nach endlich vielen Schritten endet;
- **deterministisch**, wenn in der Auswahl der Schritte keine Freiheit besteht;
- **determiniert**, wenn das Resultat eindeutig bestimmt ist;
- **sequenziell**, wenn die Schritte stets hintereinander ausgeführt werden;
- **parallel** (nebenläufig), wenn gewisse Verarbeitungsschritte nebeneinander (im Prinzip gleichzeitig) ausgeführt werden.

Formale Spezifikation eines Algorithmus

- Der Rückgriff auf definierte Funktionen erlaubt eine exakte Beschreibung des Vorgehens.
- Welche einzelnen Schritte genau definiert werden müssen, hängt vom Kontext ab.
  - Ein Kochbuch für Anfänger könnte die Funktion „Soutieren“ genau erklären, während ein Spezial-Kochbuch für Meisterköche die Kenntnis dieser Funktion voraussetzen wird.
- Zur Lösung des Problems „größte von drei Karten“ wurde die `max`-Funktion vorausgesetzt. Sie könnte auch genauer definiert werden.

Algorithmus max

$\max(k1, k2) =$

wenn $k1 \geq k2 : (k1, k2) \rightarrow k1;$

sonst: $(k1, k2) \rightarrow k2.$

- Hier setzen wir wiederum die Ordnung „ ≥“ voraus.
- Variante: Die Ordnungsrelation ist Teil des Definitionsbereichs einer 3-stelligen Maximums-Funktion:

$$\max : K \times K \times O \rightarrow K$$

$$\text{mit } O : K \times K \rightarrow \text{bool}$$

$\max(k1, k2, O) =$

wenn $O(k1, k2) : (k1, k2, O) \rightarrow k1;$

sonst: $(k1, k2, O) \rightarrow k2.$

Größte Karte: allgemeinere Lösung

- In der Informatik sucht man oft nach möglichst allgemeinen Lösungen – man formuliert Algorithmen, die ein Problem möglichst generell lösen.
  - die größte Karte einer beliebig großen Menge von Karten
  - noch allgemeiner: das größte Element einer Menge, gegeben eine Ordnungsrelation, die über der Menge total ist
- Idee: wir wollen die 2- (bzw. 3-)stellige max-Funktion (mit einer gegebenen Ordnung) weiterhin verwenden.
- Signatur unserer Lösung:

$\wp(\text{SpielKarten})\backslash\varnothing \rightarrow \text{SpielKarten}$

beliebige, nicht-leere Teilmenge der Menge aller Spielkarten

Datenstruktur

- Als Datenstruktur für eine beliebige Teilmenge einer Menge M beschreiben wir eine Liste (feste Reihenfolge der Elemente).
  - Liste(M) $\in \wp$(M)
  - Beispiel: {Pik As, Karo 7, Herz Dame} : Liste(SpielKarten)
- Annahme von Hilfsfunktionen für Listen:
  - `erstes` : Liste(M) $\rightarrow$ M
    (bildet eine Liste auf ihr erstes Element ab)
  - `rest` : Liste(M) $\rightarrow$ Liste(M)
    (bildet eine Liste auf die restliche Liste ohne das erste Element ab)
  - `laenge` : Liste(M) $\rightarrow$ N
    (bildet eine Liste auf die Anzahl ihrer Elemente ab)

„ist vom Typ“ (= Signatur) hier: 0-stellige Funktion

Beschreibung einer Lösungsidee

- Beschreibung des allgemeinen Lösungsprinzips (Algorithmus):
- wir unterscheiden mögliche Fälle:
  - wenn die Liste keine Element enthält:
    es gibt keine Lösung
  - wenn die Liste ein Element enthält:
    dieses Element ist die Lösung
  - wenn die Liste zwei Elemente enthält:
    das größte der beiden Elemente ist die Lösung (`max`)
  - wenn die Liste mehr als zwei Elemente enthält?
    Rückführung auf den Fall „zwei Elemente“...

Rekursion: Zurückführen auf einen einfacheren Fall

- Rückführung drei Elemente auf zwei Elemente:
  - `max(k1, max(k2, k3))`
- allgemein:

groesstesElement(L) =

wenn `laenge`(L) = 1: L → `erstes`(L);

wenn `laenge`(L) = 2:

L → `max`(`erstes`(L), `erstes`(`rest`(L)));

wenn `laenge`(L) > 2:

L → `max`(`erstes`(L), **groesstesElement**(`rest`(L)));

Rekursion: die Funktion verwendet sich selbst

mit einer kleineren Liste!

Prinzip der Rekursion

- Algorithmen, die (wie groesstesElement) auf sich selbst oder auf Teilalgorithmen von sich Bezug nehmen, nennt man **rekursiv**. Die dabei verwendete Technik nennt man **Rekursion** – eine zentrale Technik der Informatik, die uns in dieser Vorlesung in verschiedenen Variationen beschäftigen wird.
- Für das Gelingen einer rekursiven Definition ist es wesentlich, dass die Wiederholung der Funktion (der *rekursive Aufruf*) ein in irgendeiner Weise reduziertes Argument bekommt (eine kleinere Menge, eine kürzere Liste, eine kleinere Zahl).
- Schließlich wird das Argument so einfach, dass kein weiterer *rekursiver Aufruf* nötig ist, sondern eine andere Operation durchgeführt werden kann.
- Deshalb findet man in einer rekursiven Definition in der Regel (mindestens) einen **Rekursionsfall** und (mindestens) einen **Basisfall**.

Rekursion und Terminierung

- Wenn ein Algorithmus rekursiv definiert ist, ist es nicht immer offensichtlich, ob er terminiert.
- Man muss hierzu zeigen, dass der wiederholte Rekursionsfall immer einfacher wird und schließlich in einem Basisfall mündet.

groesstesElement(L) =
  wenn `laenge`(L) = 1: L → `erstes`(L);
  wenn `laenge`(L) = 2:
    L → `max`(`erstes`(L), `erstes`(`rest`(L)));
  wenn `laenge`(L) > 2:
    L → `max`(`erstes`(L), **groesstesElement**(`rest`(L)));

Basisfall

Rekursionsfall

Mit jedem Aufruf des Rekursionsfalles wird die übergebene Liste um ein Element kürzer – bis sie schließlich nur noch 2 Elemente enthalten wird.

Beweisprinzip der vollständigen Induktion

Formell beweist man eine wichtige Eigenschaft wie die Terminierung mit einem Induktionsbeweis.

Beweisprinzip der vollständigen Induktion
Sei f: $N \rightarrow M$ eine (totale) Funktion von den natürlichen Zahlen in eine Menge M. Sei B die Menge $\{f(n) \mid n \in N\}$.
Um zu zeigen, dass jedes Element $f(x) = b$ von B mit $x \geq n_0$ eine Eigenschaft E besitzt, genügt es zu zeigen:
1. Induktionsbasis: $f(n_0)$ besitzt Eigenschaft E.
2. Induktionsschritt: Sei n eine beliebige natürliche Zahl $\geq n_0$.
Wenn $f(n)$ die Eigenschaft E besitzt, dann auch $f(n+1)$.

Anwendung: Terminierung von groesstesElement

groesstesElement(L) =
  wenn `laenge`(L) = 1: L → `erstes`(L);
  wenn `laenge`(L) = 2:
    L → `max`(`erstes`(L), `erstes`(`rest`(L)));
  wenn `laenge`(L) > 2:
    L → `max`(`erstes`(L), **groesstesElement**(`rest`(L)));

- Induktionsbasis:

  groesstesElement terminiert für Listen der Länge 1 und 2 (n_0=1)

  (klar)
- Induktionsschritt:

  (Zu zeigen:) Wenn **groesstesElement** für eine Liste der Länge *n* terminiert, terminiert er auch für eine Liste der Länge *n*+1.

Anwendung: Terminierung von groesstesElement (2)

- Der Beweis des Induktionsschrittes setzt eine **Induktionsannahme** voraus:

  Induktionsannahme: groesstesElement terminiert für eine Liste der Länge *n*.
- Beweis des Induktionsschrittes:

  groesstesElement mit einer Liste der Länge *n*+1 vergleicht das erste Element mit dem größten Element der restlichen Liste. Hierzu wird **groesstesElement** mit einer Liste der Länge *n* aufgerufen. Dieser Aufruf terminiert *nach Induktionsannahme*. Daher terminiert auch der Aufruf mit einer Liste der Länge *n*+1.

  q.e.d.
- **Bemerkung:** Man könnte auf einen der Basis-Fälle verzichten – wie würde man den Algorithmus dann formulieren?

Eigenschaften eines Algorithmus: Korrektheit

- Damit ein Algorithmus **korrekt** ist, darf er nie ein falsches Ergebnis liefern.
- Dies ist kein hinreichendes Kriterium. In diesem Sinne wäre dieser Algorithmus für die Funktion

$$\text{plus} : N \times N \rightarrow N$$

bereits korrekt:

$$\text{plus}(x, y) = (\text{plus}(x, y))$$

Dieser Algorithmus terminiert nie, liefert also auch nie ein falsches Ergebnis.

- Man unterscheidet daher noch zwischen **partieller** und **totaler Korrektheit**.

Partielle und totale Korrektheit

- Partielle Korrektheit:

  Liefert der betrachtete Algorithmus ein Ergebnis, so ist es das erwartete (richtige) Ergebnis.

  (Beispiel: Liefert plus(a, b) einen Wert c, so gilt: $c = a + b$.)

- Totale Korrektheit:

  Der Algorithmus terminiert für alle erlaubten Eingaben und liefert jeweils das erwartete (richtige) Ergebnis.

  (Beispiel: für alle natürlichen Zahlen a und b terminiert plus(a, b) und liefert die Summe $a + b$.)

Korrektheit von groesstesElement

groesstesElement(L) =
wenn `laenge`(L) = 1: L → `erstes`(L);
wenn `laenge`(L) > 1:
L → `max`(`erstes`(L), **groesstesElement**(`rest`(L)));

- Ist **groesstesElement** total oder nur partiell korrekt?
- Ergebnis für eine Liste der Länge 0?
- Aber: Signatur war

$$\wp(\text{SpielKarten})\backslash\varnothing \rightarrow \text{SpielKarten}$$

d. h., für eine Liste der Länge 0 muss der Algorithmus kein korrektes Ergebnis liefern, er muss für eine solche Eingabe nicht einmal terminieren.

Eigenschaften eines Algorithmus: Komplexität

- Die „Komplexität“ eines Algorithmus beschreibt seinen Verbrauch an Speicherplatz und Laufzeit.
- Ein Algorithmus, der so wenig Platz und Zeit verbraucht wie möglich, heißt **effizient**.
- Man kann auch genauer unterscheiden nach **Speicherplatz-Effizienz** oder **Laufzeit-Effizienz**.
- Betrachten wir im Beispiel einen Aufruf von `max` als kleinste Zeiteinheit. Was passiert?
  `max`(k_1, `max`(k_2, `max`(k_3, ... `max`(k_{n-1}, k_n)...)))
  n-1 Aufrufe von `max` für eine Liste der Länge n.
- Man spricht von **linearem** Laufzeitverhalten, weil die Laufzeit durch eine *lineare* Funktion der Eingabelänge beschrieben wird.

Spezifikation und Programm

- Die Spezifikation eines Algorithmus erfolgt in **Pseudocode**.
- Es ist nicht genau festgelegt, wie Pseudocode aussehen muss (seine **Syntax** ist frei). Pseudocode muss den Ablauf aber eindeutig beschreiben.
- Die Spezifikation eines Algorithmus in Pseudocode ist kein Programm, ermöglicht aber, den Algorithmus in ein tatsächlich ausführbares Programm zu übersetzen.
- Die konkrete Implementierung kann je nach verwendeter Programmiersprache sehr unterschiedlich aussehen – die abstrakte Spezifikation bleibt jedoch die gleiche.

Informatik und Programmierung

- In der Informatik geht es nur zu einem kleinen Teil um Programmierung. Diese Vorlesung führt *auch* in Programmierung ein, da Informatik Programmierung voraussetzt. Wichtiger sind aber andere Aspekte, die an Programmierbeispielen veranschaulicht werden.
- Verschiedene Programmiersprachen folgen unterschiedlichen **Paradigmen** und sind zur Umsetzung unterschiedlicher Aspekte jeweils besser oder schlechter geeignet.

Programmierparadigmen

- Wichtige Paradigmen sind:
  - das imperative Paradigma
  - das logische Paradigma
  - das objekt-orientierte Paradigma
  - das funktionale Paradigma
- Viele Programmiersprachen kombinieren zwei oder mehr Paradigmen.
- Die Vorlesung Informatik 2 ist dem objekt-orientierten und imperativen Paradigma gewidmet und verwendet Java, eine Sprache, die in Industrie und Forschung stark verwendet wird.
- Die Beispiel-Sprache für Informatik 1 ist SML, eine eher akademische Sprache, die das funktionale Paradigma umsetzt.

SML – funktionales Paradigma

- Nach dem funktionalen Paradigma ist ein SML Programm als eine Sammlung von Gleichungen zu verstehen, die Funktionen modellieren.
- Außer in speziellen Anwendungen ist SML als Sprache nicht von praktischer Bedeutung – es ist aber eine sehr „reine“ Sprache, die Ihnen das Verständnis theoretischer Zusammenhänge erleichtern soll.
- Weitere Programmiersprachen für die praktische Anwendung sollten Sie im Lauf des Studiums selbständig lernen! Die Konzepte sind immer wieder die gleichen.

`max` und `groesstesElement` in SML (Ausblick)

- als Beispiel die Funktionen `max` (3-stellig, d. h. mit Ordnung) und `groesstesElement` in SML:

```
val max = fn (x, y, ord) =>
  if ord(x, y) then x
  else y;
val rec groesstesElement = fn (liste, ord) =>
  if length(liste) = 1 then hd(liste)
  else max(hd(liste),
           groesstesElement(tl(liste), ord),
           ord);
```

- Eine grundlegende Einführung in SML sehen wir im nächsten Kapitel, Datenstrukturen wie eine Liste erst später – aber vielleicht verstehen Sie ja nun schon das eine oder andere Element...