2. Kapitel

ALGORITHMEN

Algorithmen & Datenstrukturen

Prof. Dr. Wolfgang Schramm

Übersicht

Lernziele des Kapitels

- Verstehen was ein Algorithmus ist.
- Möglichkeiten zur Formulierung von Algorithmen kennenlernen.
- Bausteine zur Beschreibung von Algorithmen kennenlernen.
- Verschiedenen Typen von Algorithmen kennenlernen.
- Rekursion kennen- und verstehen lernen.

Algorithmen - Kapitelübersicht

1. Einführung
2. Definition & Eigenschaften
3. Notationen
4. Rekursion

Was ist ein Algorithmus?

- …eine Handlungsanweisung?
- …ein Weg, ein bestimmtes Ziel zu erreichen?
- …?

- …haben Daten etwas damit zu tun?
- …haben Informationen etwas damit zu tun?
- …?

Bekannte Algorithmen 1/3

…aus dem Alltag

- Waschanleitung
  - Wäsche in die Trommel stecken, Waschmittelzugabe abhängig von Hersteller/Verschmutzung/Wasserhärte, Temperatur einstellen (30°/60°/90°), Startknopf drücken
- Rezept
  - 3x täglich 3 Tropfen einnehmen
- Spielanleitung
  - Dame: …
  - Mensch ärgere dich nicht: …

Bekannte Algorithmen 2/3

…aus der Mathematik

- Multiplikation zweier ganzer Zahlen
  - **33 x 24**

    66

    132

    792
- Euklidischer Algorithmus zur Bestimmung des ggT
- Ziehen einer Wurzel
  - …

Bekannte Algorithmen 3/3

…aus der Betriebswirtschaft

- Sortieren einer unsortierten Kartei (Karteikarten) nach der Kundennummer.
- Suchen der Artikelnummer zu einem Artikel.
- Suchen aller Teile, die für die Produktion einer Maschine benötigt werden.

Algorithmenbegriff: 1. Näherung

Intuitive Begriffsbestimmung

- Ein Algorithmus ist eine eindeutige Beschreibung eines in mehreren Schritten durchgeführten (Bearbeitungs-) Vorgangs.
- Ein Algorithmus ist eine formale Vorschrift für ein endliches Verfahren, das den Eingabegrößen eindeutig die Ausgabegrößen zuordnet.

Diskussion

Partnerdiskussion

Definieren Sie:

„Ein Algorithmus ist … “

3 Min.

Algorithmus – Definitionsversuche

- Unter einem Algorithmus versteht man eine Verarbeitungsvorschrift, die so präzise formuliert ist, dass sie von einem mechanisch oder elektronisch arbeitenden Gerät durchgeführt werden kann [Informatik Duden].

- Algoṛithmus [griechisch] *der,* Verarbeitungsvorschrift, die aus einer Folge von Anweisungen besteht und mit der eine Vielzahl gleichartiger Aufgaben gelöst werden kann. Damit ein Algorithmus mit einem Computer gelöst werden kann, muss er mit endlichen Ressourcen (Anzahl der Anweisungen, Speicherplatz) realisierbar sein und in endlicher Zeit ausgeführt werden können. Außerdem muss er für gleiche Eingabewerte unter gleichen Bedingungen stets die gleiche Ausgabe liefern [Meyers Lexikon online].

Diskussion

Partnerdiskussion

Entwickeln Sie einen Algorithmus, der eine Zerlegung von 1 Million in zwei Faktoren, die nicht durch 10 teilbar sind, findet.

10 Min.

Algorithmen – historischer Überblick 1/2

300 v. Chr.: Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier natürlicher Zahlen.

800 n. Chr.: Der persische Mathematiker Abu Abdullah Muhammad bin Musa al-Khwarizmi veröffentlicht eine Aufgabensammlung für Kaufleute und Testamentsvollstrecker – später in lateinischer Übersetzung: Liber Algorithmi.
Algorithmus = Kunstwort aus arithmos (gr.) + Name dieses Mathematikers.

1574: Rechenbuch von Adam Riese verbreitet mathematische Algorithmen in Deutschland.

1614: Die ersten Logarithmentafeln werden algorithmisch berechnet (Dauer: 30 Jahre).

1703: Leibniz führt binäre Zahlensysteme ein.

1815: Geburt von Augusta Ada Lovelace (erste „Computerpionierin“), Konstruktionspläne für verschiedenartige Maschinen, Assistentin von Charles Babbage, entwirft Programme für dessen erste Rechenmaschinen.

Algorithmen – historischer Überblick 2/2

1822: Charles Babbage, entwickelt sog. Difference Engine (1833 in verbesserter Version fertiggestellt). Später (aber nie fertig gestellt) Entwicklung der Analytical Engine, welche die wichtigsten Komponenten eines Computers umfasst.

1931: Kurt Gödel beendet den Traum vieler (damaliger) Mathematiker, dass alle mathematischen Sätze mit algorithmisch konstruierten Beweisen durchgeführt werden können (Fundamentalsatz der Nichtberechenbarkeit).

1936: These von Alonzo Church vereinheitlicht die Welt der Sprachen zur Notation von Algorithmen. Er zeigt, dass viele Notationen die gleiche Ausdrucksfähigkeit haben (mit Hilfe sog. Turingmaschinen).

danach: Mit der Realisierung der ersten Computer $\Rightarrow$ Ausbau der Algorithmentheorie zu einem eigenen Wissensgebiet.

Algorithmus: Definition

Ein Algorithmus ist eine präzise, d.h. in einer festgelegten Sprache abgefasste, endliche Beschreibung eines schrittweisen Problemlösungsverfahrens zur Ermittlung gesuchter Größen aus gegebenen Größen, in dem jeder Schritt aus einer Anzahl ausführbarer eindeutiger Aktionen und einer Angabe über den nächsten Schritt besteht.

Rechenberg/Pomberger: Informatik Handbuch, Hanser, München, 2002

Vom Algorithmus zum Programm

- Ein Prozessor führt einen Prozess (Arbeitsvorgang) auf Basis einer eindeutig interpretierbaren Beschreibung (dem Algorithmus) aus.

- Programmieren im Sinne der Informatik heißt, ein Lösungsverfahren für eine Aufgabe so zu formulieren, dass es von einem Prozessor ausgeführt werden kann.

Rechenberg/Pomberger: Informatik Handbuch, Hanser, München, 2002

Diskussion

Partnerdiskussion

Suchen Sie die wesentlichen Stichworte der Definition heraus!

10 Min.

Eigenschaften von Algorithmen

- Terminiertheit
- Determinismus
- Determiniertheit
- Finitheit
- Berechenbarkeit
- Korrektheit
- Effizienz
  - Geschwindigkeit
  - Platzbedarf
- Ausdrucksfähigkeit verschiedener Notationen
- Datenstrukturen

Eigenschaften von Algorithmen: Terminierung/ Determinismus/Determiniertheit

- **Terminiertheit**
  - Ein Algorithmus heißt **terminierend,** wenn er (bei jeder erlaubten Eingabe von Parameterwerten) nach endlich vielen Schritten abbricht.

- **Determinismus**
  - Ein **deterministischer Ablauf** bedeutet, dass der Algorithmus eine eindeutige Vorgabe der Schrittfolge der auszuführenden Schritte festlegt. Der Algorithmus ist deterministisch.

    M.a.W: Ein Algorithmus ist deterministisch, wenn zu jedem Zeitpunkt der Algorithmusausführung der nächste Handlungsschritt eindeutig definiert ist.
- **Determiniertheit**
  - Ein **determiniertes Ergebnis** wird von Algorithmen dann geliefert, wenn bei vorgegebener Eingabe ein eindeutiges Ergebnis geliefert wird – insbesondere auch bei mehrfacher Durchführung des Algorithmus (mit denselben Eingabeparametern).

Beispiel für einen nichtdeterministischen Ablauf

Bearbeitungsvorschrift für das Sortieren eines Stapels von Karteikarten:

Sortieren:

- Wähle zufällig eine beliebige Karte, bilde 2 Stapel (lexikographisch vor der Karte, lexikographisch nach der Karte),
- sortiere diese beiden (kleineren) Stapel,
- füge die sortierten Stapel mit der ausgewählten Karte in der Mitte wieder zusammen.

Ablauf des Algorithmus: **nichtdeterministisch**

Ergebnis des Algorithmus: determiniert

Algorithmen als Funktion mit bestimmter Semantik

- Terminierende, deterministische und determinierte Algorithmen definieren eine Ein-/Ausgabefunktion:

$$f : \text{Eingabewerte} \rightarrow \text{Ausgabewerte}$$

- Algorithmen geben eine konstruktiv ausführbare Beschreibung dieser Funktion an.
- Die Ein-/Ausgabefunktion bezeichnen wir als Bedeutung (Semantik) des Algorithmus.
- Es kann mehrere verschiedene Algorithmen mit der gleichen Bedeutung geben.

Anmerkung: In der Mathematik gibt es nicht-konstruktive Beweise, z.B. Beweis durch Widerlegen des Gegenteils.

Funktion zu Beispielalgorithmus

Test ob eine gegebene natürliche Zahl eine Primzahl ist:

$f : \mathbf{N} \rightarrow \{\mathbf{ja}, \mathbf{nein}\}$ mit $f(n) =$

$$\begin{cases} \text{ja} & \text{falls } n \text{ Primzahl} \\ \text{nein} & \text{sonst} \end{cases}$$

Beispiele: Nichtdeterminismus und (Nicht-) Determiniertheit

Nichtdeterministische Algorithmen mit determiniertem Ergebnis bezeichnet man als determiniert.

1. Nehmen Sie eine beliebige Zahl x.
2. Zählen Sie die Sekunden der aktuellen Zeit hinzu.
3. Addieren Sie 5 hinzu und multiplizieren Sie mit 3.
4. Schreiben Sie das Ergebnis auf.

Ablauf: nichtdeterministisch

Ergebnis: nicht determiniert

⇒ Algorithmus ist nicht determiniert

1. Nehmen Sie eine Zahl x ungleich 0.
2. Entweder: Addieren Sie das Dreifache von x zu x und teilen das Ergebnis durch x
   $(3x + x) / x$
3. Oder: Subtrahieren Sie 4 von x und subtrahieren das Ergebnis von x
   $x - (x - 4)$
4. Schreiben Sie das Ergebnis auf.

Ablauf: nichtdeterministisch

Ergebnis: determiniert

⇒ Algorithmus ist determiniert

Zusammenhang (Nicht-) Determinismus und (Nicht-) Determiniertheit

- **Deterministische** Algorithmen sind immer auch **determiniert**. Die Umkehrung gilt jedoch nicht!

- Viele Algorithmen sind **deterministisch und determiniert.**
- Von Rechnern ausgeführte Programme sind in der Regel sowohl deterministisch als auch determiniert. Nur mit Mühe kann man nichtdeterminierte Effekte simulieren.

Eigenschaften von Algorithmen: Finitheit

- Ein Algorithmus ist eine endlich lange Operation.
- Statische Finitheit:
  - Jeder Algorithmus muss statisch finit sein, d.h. er wird durch einen endlich langen Text (z.B. Programm) beschrieben.
- Dynamische Finitheit:
  - Ein Algorithmus ist dynamisch finit, wenn die von ihm verwendeten Objekte und Strukturen zu jedem Zeitpunkt endlich bleiben.

Eigenschaften von Algorithmen: Berechenbarkeit

- Eine Funktion f:M →N heißt **berechenbar**, wenn es einen Algorithmus gibt, der für jeden Eingabewert m ∈ M, für den f(m) definiert ist, nach endlich vielen Schritten anhält und als Ergebnis f(m) liefert .

- Church‘sche These: die Klasse der berechenbaren Funktionen ist gleich der Klasse der Funktionen ist die durch Turingmaschinen berechnet werden kann.
- Ein Algorithmus bzw. eine Turingmaschine realisiert eine berechenbare Funktion .

Eigenschaften von Algorithmen: Korrektheit, Effizienz

- Ein Algorithmus heißt **korrekt**, wenn er genau die vorgegebene Spezifikation erfüllt, also auf alle Eingabedaten mit den gewünschten Ausgabedaten reagiert .
- Die Verifikation dient als formaler Nachweis von Eigenschaften der Programme die Algorithmen implementieren.

- Ein Algorithmus heißt **effizient**, wenn er ein vorgegebenes Problem in möglichst kurzer Zeit und/oder mit möglichst geringem Aufwand an Betriebsmitteln löst.

Bausteine für Algorithmen 1/3

- **Elementare Operationen** (primitives): Basiselemente, die ausgeführt werden, ohne näher aufgeschlüsselt zu werden.

  Schneide Fleisch in kleine Würfel.

- **Sequenz** (sequence): Hintereinanderausführen von Schritten.

  Bringe das Wasser zum Kochen, dann gib das Paket Nudeln hinein, schneide das Fleisch, dann das Gemüse.

- **Parallelität** (concurrency): Gleichzeitiges Ausführen von Schritten.

  Ich schneide das Fleisch, du das Gemüse.

- **Bedingung** (condition): Ein Schritt wird nur ausgeführt, wenn eine bestimmte Bedingung erfüllt ist.

  Wenn die Soße zu dünn ist, füge Mehl hinzu.

Bausteine für Algorithmen 2/3

- **Schleife** (iteration, loop): Wiederholung einer Tätigkeit, bis eine vorgegebene Endbedingung erfüllt wird.

  Koche die Nudeln solange, bis sie al dente sind.

- **Unterprogramm** (subroutine): Beschreibt durch seinen Namen eine Bearbeitungsvorschrift, die „aufgerufen“ wird, um ausgeführt zu werden. Nach Durchführung dieser Bearbeitungsvorschrift, fährt man mit dem eigentlichen Algorithmus an der Stelle fort, an der man zum Unterprogramm gewechselt war.

  Bereite Soße nach Rezept auf Seite 42.

- **Rekursion** (recursion): Anwendung desselben Prinzips auf in gewisser Weise „kleinere“ oder „einfachere“ Teile, solange, bis sie direkt gelöst werden können.

  Viertele das Fleischstück in 4 gleichgroße Teile. Falls die Stücke länger sind als 2 cm, verfahre mit den einzelnen Stücken genauso bis die gewünschte Größe erreicht ist.

Bausteine für Algorithmen 3/3

Minimale Menge von Konstrukten (nach Dijkstra 1930-2002):

- Elementare Operationen +
- Sequenz +
- Bedingung +
- Schleife.

Im Zusammenhang mit der Einführung des Begriffs der strukturierten Programmierung in der Informatik (Anfang 1970er Jahre).

Notationen

- Natürliche Sprache
- Pseudocode
- Struktogramme
- Aktivitätsdiagramme

Pseudocode-Notation

- Pseudocode-Algorithmen werden i.d.R. unter Verwendung spezieller englisch-sprachiger Begriffe formuliert.
- Diese Begriffe sind der Alltagssprache entnommen und haben eine festgelegte Bedeutung für den Ablauf eines Verfahrens. Man bezeichnet sie als Kontroll- oder **Schlüsselwörter**.
- Um sie besser kenntlich zu machen, werden Schlüsselworte entweder **fett** geschrieben oder unterstrichen.
- Basisoperationen haben die Form einfacher Befehlssätze (wie in Programmiersprachen).
- Die folgenden Beispiele sind aus dem täglichen Leben.

Pseudocode-Notation: Sequenz

Durchnummerierung der Schritte

(1) Koche Wasser

(2) Gib Kaffeepulver in die Tasse

(3) Fülle Wasser in die Tasse

Vorteil: Elegante Form der Verfeinerung von Schritten möglich ⇒ Entwurfsprinzip der schrittweisen Verfeinerung

(2) Gib Kaffeepulver in die Tasse

(2.1) Öffne Kaffeeglas

(2.2) Entnehme Löffel voll Kaffee

(2.3) Kippe Löffel in Tasse

(2.4) Schließe Kaffeeglas

Expliziter Sequenzoperator: **;**

Sequenz-operator

Koche Wasser;

Gib Kaffeepulver in die Tasse;

Fülle Wasser in die Tasse;

Vorteil: Erspart das oft umständliche Durchnummerieren.

Pseudocode-Notation: Bedingung


```
if Bedingung
    then Schritt
fi
```

```
if Ampel rot und grüner Pfeil
    then biege vorsichtig rechts ab;
fi
```

```
if Bedingung
    then Schritt a;
    else Schritt b;
fi
```

```
if Ampel rot oder gelb
    then stoppe;
    else fahre weiter;
fi
```

eingerückt

umgedrehtes if

Schachtelung ist möglich:

```
if Ampel ausgefallen
    then
        fahre vorsichtig weiter;
    else
    if Ampel rot oder gelb
       then stoppe;
       else fahre weiter;
    fi
fi
```

- Werden mehrere Schritte in einem Zweig ausgeführt, dann werden sie eingerückt.
- Eine Schrittfolge wird beendet entweder durch eine erneute Bedingung (Schachtelung) oder durch das Wort **fi**.

Pseudocode-Notation: Schleife / Iteration 1/3

do Schritte
while Bedingung

Schleife wird durchlaufen.

Nach dem Durchlaufen wird überprüft, ob die Schleife nochmals durchlaufen werden muss.

```
Initialisiere Zahl mit 1;

/* gib nächste Primzahl aus */
do
    addiere 1 auf Zahl;
    teste Zahl auf Primzahleigenschaft;
while Zahl keine Primzahl ist;

gebe Zahl aus;
```

Schleifenrumpf

Pseudocode-Notation: Schleife / Iteration 2/3


```
while Bedingung
do
  Schritte
od
```

Vor dem Durchlaufen wird überprüft, ob die Schleife nochmals durchlaufen werden muss.

```
/* Bestimmung der größten Zahl einer Liste */
Setze erste Zahl als bislang größte Zahl;
while Liste nicht erschöpft
do
    lies nächste Zahl der Liste;
    if diese Zahl > bislang größte Zahl
       then setze diese Zahl als bislang größte Zahl;
    fi;
od;

gebe bislang größte Zahl aus;
```

Schleifenrumpf

Pseudocode-Notation: Schleife / Iteration 3/3

foreach Bereichsangabe
do
  Schritte
od

Schleife wird sooft durchlaufen, wie angegeben.

```
/* Bestimmung des Durchschnittsalters der Hörer der Vorlesung */

Setze Durchschnittsalter auf 0;
foreach Hörer der Vorlesung ADS
do
   ermittle sein bzw. ihr Alter;
   addiere das Alter auf das Durchschnittsalter;
od;
gebe Durchschnittsalter /  Anzahl der Hörer aus;
```

} Schleifenrumpf

Pseudocode-Notation: Unterprogramm 1/4

Definition einer Funktion:

funct *Name* (*Parameter*) **returns** *Typ*
    Schritte
tcnuf

Eingabewerte

Ergebnis (typ)/
Ausgabe

```
/* Berechne arithmetischen Mittelwert  */
funct mittel (wert1, wert2) returns integer
    addiere wert1 und wert2, ergibt summe;
    teile summe durch 2, ergibt dwert;
    return dwert
tcnuf
```

Pseudocode-Notation: Unterprogramm 2/4

Aufruf einer Funktion:

Name (*Parameter*);

```
/* errechne Mittelwert */
eingabe zahl1;
eingabe zahl2;
mittelwert = mittel (zahl1, zahl2);
drucke (mittelwert);
```

gleicher Name ➔ gleiche Anzahl

Das Ergebnis wird einer Variablen von selben Typ zugewiesen

Auch geschachtelte Funktionsaufrufe sind möglich:

```
drucke (mittel (zahl1, zahl2));
```

Pseudocode-Notation: Unterprogramm 3/4

Unterprogramm ohne Rückgabe (Ergebnis) = Prozedur

proc *Name (Parameter)*
    Schritte

corp

drucke **(…)**

Kein Rückgabewert!

Pseudocode-Notation: Unterprogramm 4/4

Formale vs. aktuelle Parameter

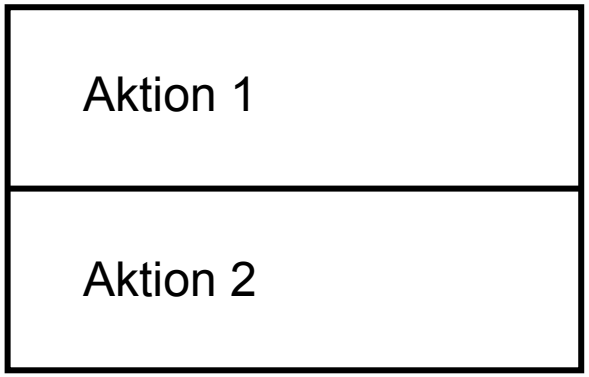
```
/* Berechne arithmetisches Mittel*/
   funct mittel (integer a, integer b) returns integer
      addiere a und b, ergibt summe;
      teile summe durch 2, ergibt mittelwert;
      return mittelwert
   tcnuf

   ...

   mittelwert = mittel (zahl1, zahl2);
```

formale Parameter

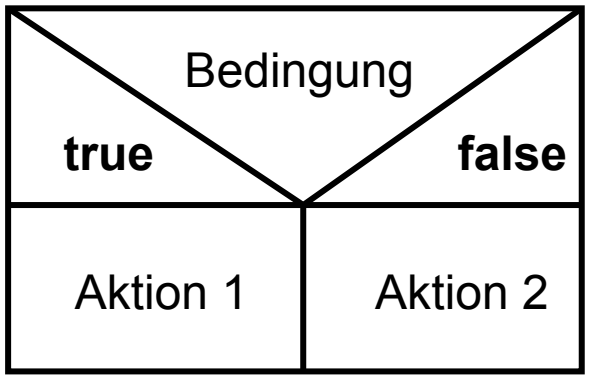
aktuelle Parameter

Struktogramme: Bausteine 1/2

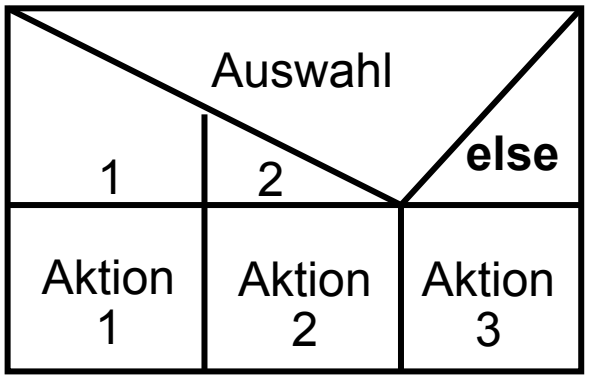
Prozess



Sequenz



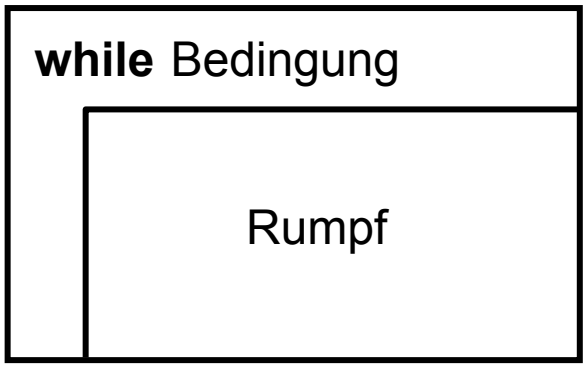
Bedingte Anweisung

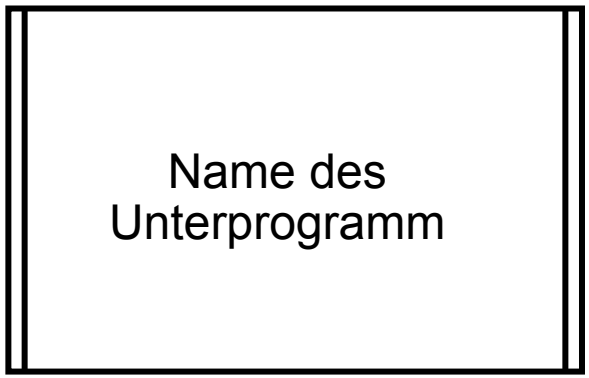


Mehrfach-Auswahl

Struktogramme: Bausteine 2/2

Schleifen



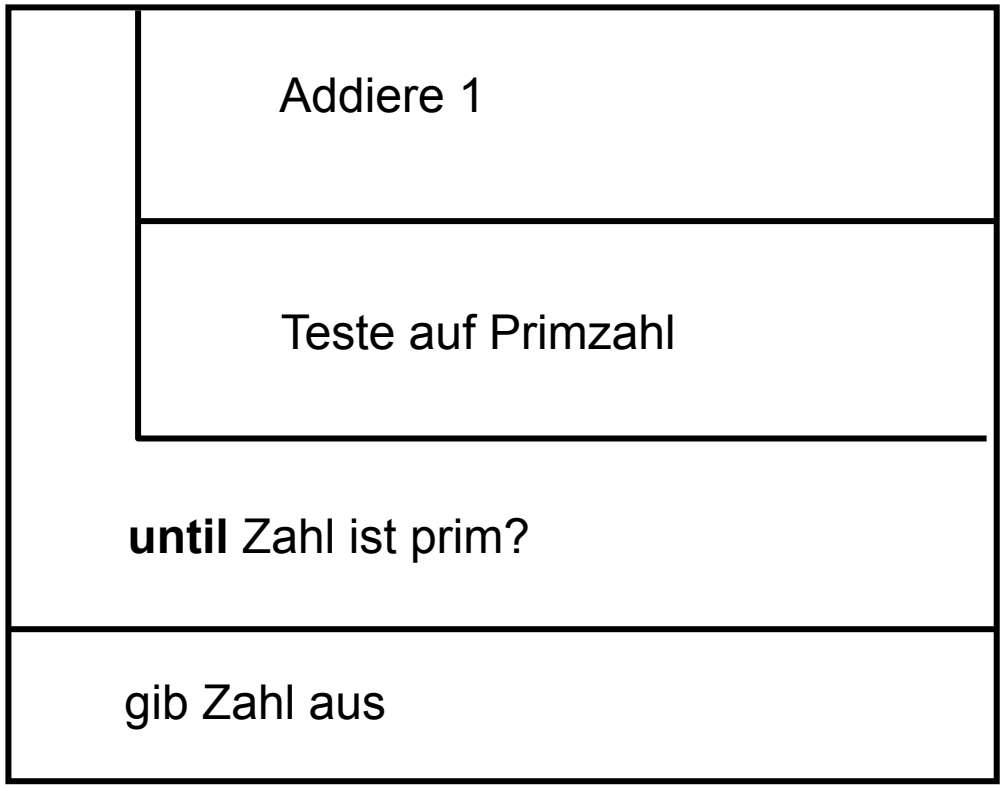


Unterprogrammaufruf

Name:

Unterprogramm-
vereinbarung

Struktogramme - Beispiel



Aktivitätsdiagramme - Eigenschaften

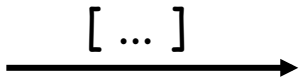
- Aus der Unified Modeling Language (UML) – activity diagrams

- Aktivitätsdiagramme beschreiben:
  - Wie Aktivitäten koordiniert werden.
  - Was an unterschiedlichen Dingen bewerkstelligt werden muss.
  - Wie sich die einzelnen Arbeitsabläufe entwickeln und von anderen abhängen.
  - Wie ein gesamter Arbeitsfluss aussieht.
  - Welche Vorgänge parallel ablaufen können.
  - Arbeitsabläufe in Unternehmen.

Aktivitätsdiagramme – Elemente 1/4

- **Aktivität** (activity)

  Vorgang der zu erledigen ist. Wenn der Vorgang erledigt ist, wird die Aktivität verlassen. Eine Aktivität kann viele Schritte einschließen (→schrittweise Verfeinerung).

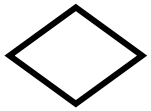


- **Übergang** (transition)

  Erfolgt zwischen 2 Aktivitäten. Der Abschluss einer Aktivität löst die folgende Aktivität (Quellzustand → Zielzustand) aus. Es kann mehrere nach außen gehende Übergänge geben. Übergänge können Wächter (guard conditions) beinhalten, mit denen die Verzweigung des Kontrollflusses beschrieben wird.

Aktivitätsdiagramme – Elemente 2/4

- **Entscheidungskonten** (decision nodes)



Darstellung von Entscheidungen, die als Alternative zu Bedingungen für unterschiedliche Übergänge dienen, die denselben Zustand verlassen. Die Entscheidungskriterien werden an den Übergängen aus den Entscheidungsknoten mit Hilfe von guard conditions - [...] - notiert.

Die Raute wird auch für die Zusammenführung von Kontrollflüssen verwendet.

- Anfangs- und Ende**markierungen** (start and final markers)

Kennzeichnen den Beginn bzw. das Ende einer komplexen Aktivität. Die Markierungen sind optional. Es gibt maximal eine Anfangsmarkierung. Endemarkierungen kann es mehrere geben.

Es gibt auch eine spezielle Markierung für das Ende eines Kontrollflusses.

Aktivitätsdiagramme – Elemente 3/4

- Synchronisationsbalken (synchronisation nodes)

  Sobald alle Aktivitäten abgeschlossen sind, die Übergänge auf den Synchronisationsbalken haben, kann er passiert werden. Zu diesem Zeitpunkt werden alle aus dem Synchronisationsbalken herausführenden Übergänge ausgelöst. D.h. die Aktivitäten, auf welche die die Übergänge verweisen werden parallel ausgeführt.

  Man unterscheidet 2 Arten von Übergängen:

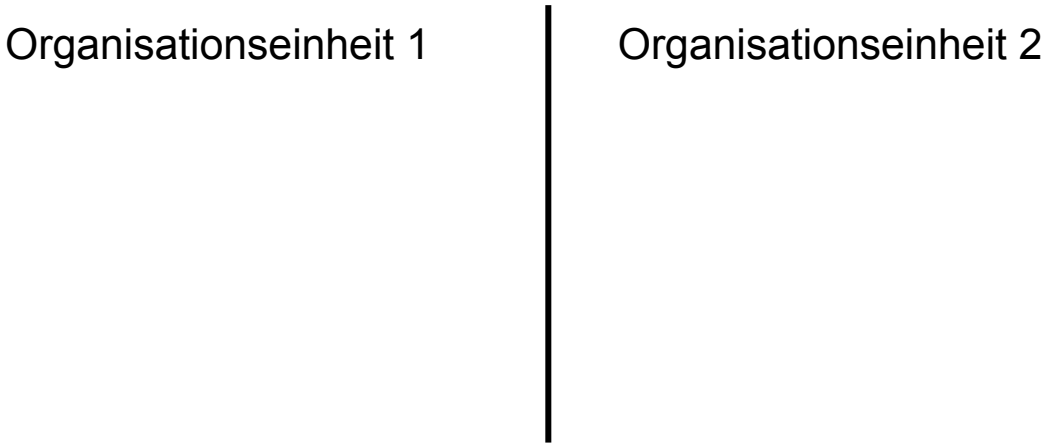
  a) Verknüpfung (synchrone Zusammenführung - join): warten, bis alle untergeordneten Aufgaben erledigt sind, bevor man weiter geht.

  b) Gabelung (parallele Ausführung - fork): gleichzeitiges Starten untergeordneter Aufgaben.

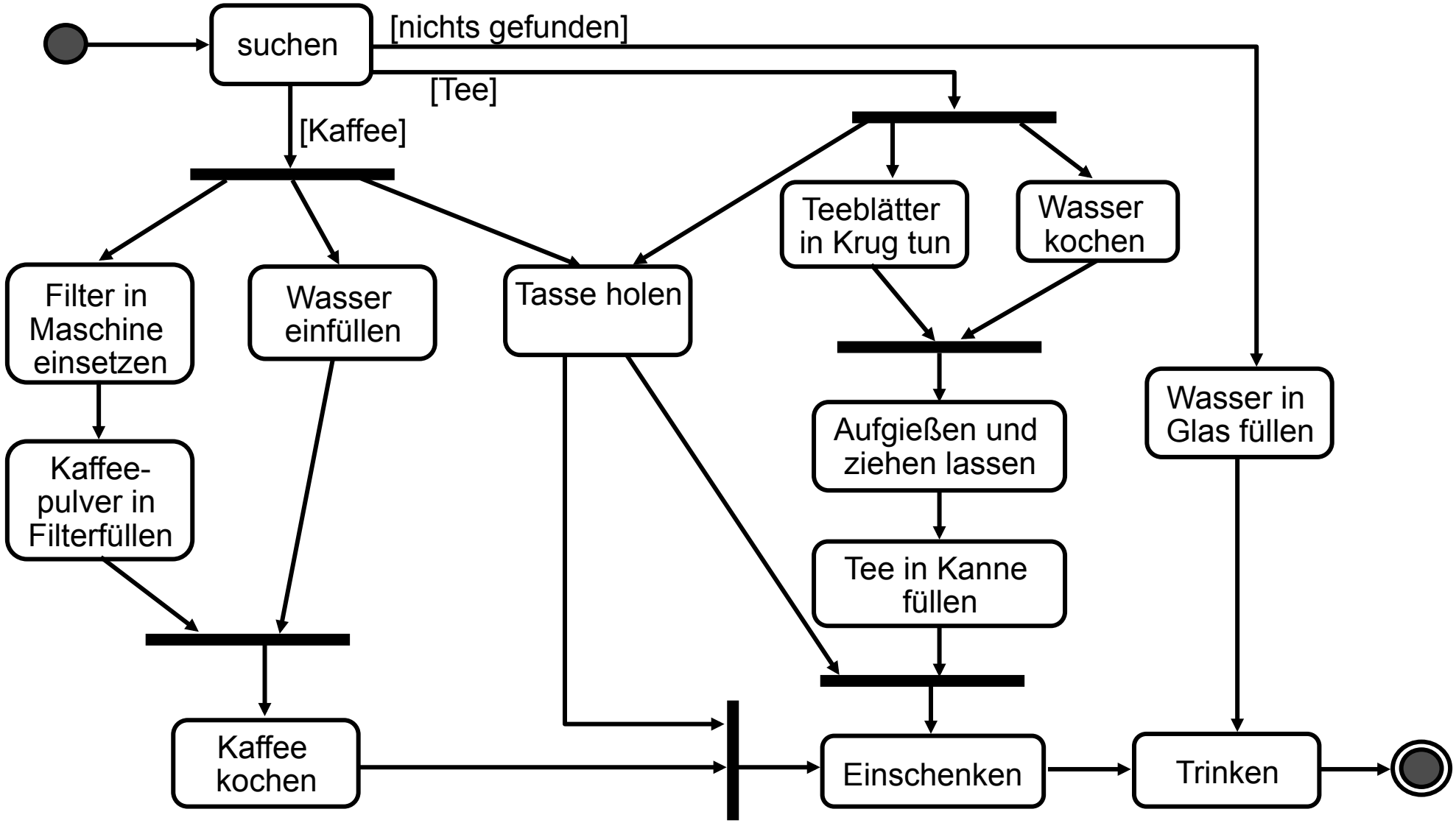
Aktivitätsdiagramme – Elemente 4/4

- Vertikale Aufteilung in sog. Schwimmbahnen (swimlanes)

Durch die Aufteilung in vertikale Abschnitte (vertikale Linien) ist es möglich Aktionen nach **Verantwortlichkeiten** zu organisieren, z.B. nach den zuständigen Organisationseinheiten einer Firma, die am Kopf der Abschnitte angegeben werden.



Aktivitätsdiagramme – Beispiel



Zurückführen eines Problems auf das gleiche Problem mit geringerer Komplexität.

Rekursion: Beispiel 1/3

- „Löse das Problem,
  indem du es auf das gleiche Problem,
  aber in kleinerem Maße zurückführst.“
- Beispiel 1:
  - Fakultät
  - Problem
    - Wie berechne ich 4! ?

$4! = \underbrace{1 * 2 * 3}_{3!} * 4$

- gleiches Problem, jedoch kleiner
  → Wie berechne ich 3! ?
- Wenn ich weiß, was 3! ist, so weiß ich auch, was 4! ist
  → $4! = 1 * 2 * 3 * 4 = (1 * 2 * 3) * 4 = 3! * 4$

Rekursion: Beispiel 2/3

- allgemein
  `n! = (n-1)! * n`
- Klappt das?

Aufruf: `n!` — Problem „Größe“ n

Aufruf: `(n-1)!` — Problem „Größe“ n-1

Nein:

```
fac (4) → 4 * fac (3)
fac (3) → 3 * fac (2)
fac (2) → 2 * fac (1)
fac (1) → 1 * fac (0)
fac (0) → 0 * fac (-1)
```
etc.

Irgendwann muss ein Ende sein!

Rekursion 3/3


```
n! = | 1            falls n=1
     | (n-1)! * n    falls n>1
```

Terminierender Fall
(n minimal)

Rekursiver Fall
(n wird kleiner!)

- Klappt das?

Ja:

```
fac (4) → 4 * fac (3) → 4 *          6 = 24
fac (3) → 3 * fac (2) → 3 *      2 = 6
fac (2) → 2 * fac (1) → 2 * 1 = 2
fac (1) → 1
```

Rekursion: Beispiel zusammengefasst

Beispiel Fakultät

$$n! = \begin{cases} 1 & \text{falls } n=1 \\ (n-1)! * n & \text{falls } n>1 \end{cases}$$

- Schreibweise in Pseudocode
  fac (n)
- Lösung
  - Problem:
  - Kleineres Problem:
  - Wenn kleineres Problem gelöst:
  - Trivialer Fall:
  - Unterscheidung kl. Probl./triv. Fall:

```
fac (n)
fac (n-1)
n * fac (n-1)
fac (1) = 1
n = 1
➔ Triv. Fall
```

Rekursion: Beispiel in Pseudocode

Pseudocode für Fakultät:

```
funct fac (n) returns integer
   if n = 1
   then
      return 1
   else
      return n * fac (n-1)
   fi
tcnuf
```

Rekursion: Regeln zur Erstellung einer rekursiven Funktion

0. Gegeben: Aufgabe der Größenordnung n.
1. Finde eine kleinere Aufgabe.
2. Drücke die ursprüngliche mit der kleineren Aufgabe aus → rekursiver Fall der Programms/Rekursionsschritt.
3. Finde den trivialen Fall
   → terminierender Fall/Rekursionsabbruch.
4. Bestimme Fallunterscheidung
   → if-Bedingung.
5. Schreibe Funktion.

```
funct f (p) returns integer
   if   <<if-Bedingung>>
   then <<terminierender
          Fall>>
   else <<rekursiver Fall>>
   fi
tcnuf
```

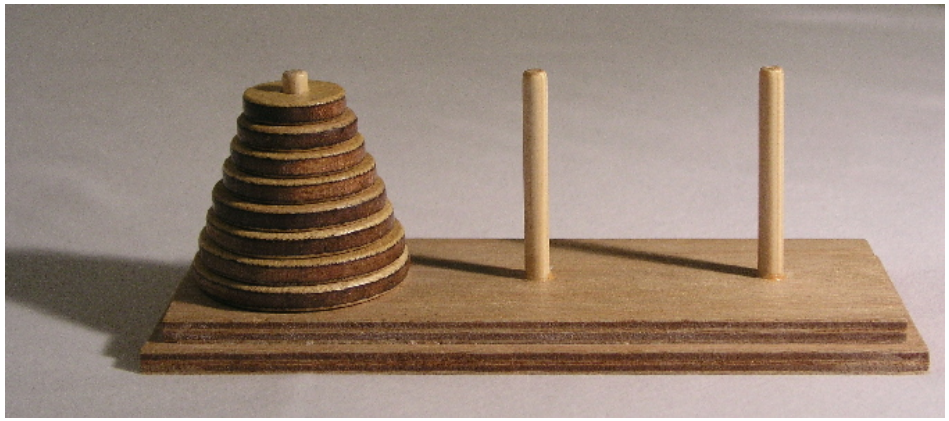
Rekursion: Beispiel – Türme von Hanoi 1/3

Türme von Hanoi

- Zu jedem Zeitpunkt können Türme von Scheiben unterschiedlichen Umfangs auf 3 Plätzen stehen. Der ursprüngliche Standort wird als Quelle bezeichnet, das Ziel als Senke. Der dritte Platz dient als Arbeitsbereich (bezeichnet als AB), um Scheiben zwischenzulagern.

- Nur die jeweils oberste Scheibe eines Turms darf einzeln bewegt werden.

- Dabei darf niemals eine größere auf einer kleineren Scheibe zu liegen kommen.



Rekursion: Beispiel – Türme von Hanoi 2/3

Wie löst man diese Aufgabe – z.B. für n = 4 Scheiben?

Es ist gar nicht so einsichtig, in welcher Reihenfolge man Scheiben von wo nach wo bewegen muss, um tatsächlich dieses Ziel zu erreichen.

Durch Nachdenken kommt man allerdings zu der Erkenntnis, dass man, sofern man weiß, wie man einen um eins kleineren Turm (n = 3) bewegen muss, auch den größeren Turm (n = 4) bewegen kann:

```
subroutine Turmbewegung (n, Quelle, Senke, AB)
/* Bewegt einen Turm der Höhe n von Quelle nach Senke unter
   Zuhilfenahme des Arbeitsbereichs AB                    */
if n = 1                                   <-- Rekursionsabbruch
then bewege oberste Scheibe von Quelle zur Senke;
else Turmbewegung (n-1, Quelle, AB, Senke);
     bewege oberste Scheibe von Quelle zur Senke;
     Turmbewegung (n-1, AB, Senke, Quelle);   <-- Rekursionsschritt
```

Rekursion: Beispiel – Türme von Hanoi 3/3

Das Prinzip sagt folgendes:

Möchte ich einen Turm der Höhe 4 von Platz A nach Platz B ziehen (unter Zuhilfenahme von Platz C), kann ich das dadurch erreichen, indem einen Turm der Höhe 3 erst von A nach C bewege (jetzt unter Zuhilfenahme von Platz B), dann die unterste (= größte) Scheibe (das ist die 4. Scheibe) direkt nach Platz B lege und den Turm der Höhe 3 von Platz C nach Platz B ziehe.

Das **Verfahren** heißt **rekursiv**, da sich die Turmbewegung (der Größe n) unter anderem wiederum durch 2 Turmbewegungen (nun der Höhe n-1) beschreiben lässt.

Rekursion vs. Iteration

Iteration und Rekursion

Iteration kann man gut verstehen	Rekursion kann man nicht verstehen
Erst versteht man nichts, dann versteht man etwas, dann noch etwas, und noch etwas, usw. usw.	Rekursion versteht man nur, wenn man Rekursion versteht

[Quelle: Prof. Schnitzspan]