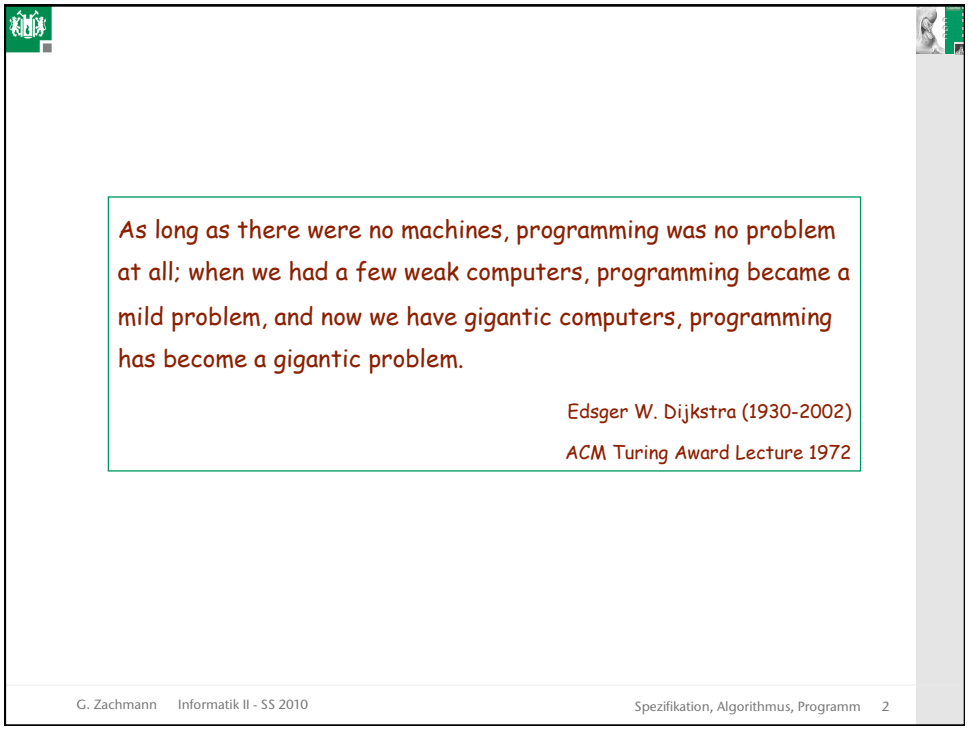
Informatik II
Spezifikation, Algorithmus, Programm

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de



As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become a gigantic problem.

Edsger W. Dijkstra (1930-2002)
ACM Turing Award Lecture 1972

Das "algorithmisches Denken"

- Aufgabe: Muffins backen
- Rezept:

```
Muffins:
Verrühre 200g Butter, 200g Zucker, 4 Eier, 1 Pck.
Backpulver, 1 Pck. Vanillezucker, 250 g Mehl, 3
Eßl. Rum und drei mittelgroße geschälte und
zerteilte Äpfel; fülle den Teig in Muffinförmchen;
backe bei 175-200 Grad für etwa 30 min; bestäube
die Muffins mit etwas Puderzucker.
```

- Unser Rezept ist von einer sehr einfachen Form, denn es besteht lediglich aus einer Aneinanderreihung einfacher elementarer Anweisungen. Komplexere Algorithmen können weitere Strukturen enthalten.

- Aufgabe: Apfelzeit berücksichtigen; ist eine Bedingung, die nicht immer erfüllt ist → *bedingte Anweisungen*:

```
Muffins:
Verrühre 200 g Butter, 200 g Zucker,4 Eier,1 Pck.
Backpulver,1 Pck. Vanillezucker, 250 g Mehl,3 Eßl.
Rum;
falls es Apfelzeit ist:
    füge drei mittelgroße geschälte und
    zerteilte Äpfel hinzu;
sonst:
    füge 100 g Schokoladenstückchen hinzu;
fülle den Teig in Muffinförmchen; backe bei 175-200
Grad für etwa 30 min; bestäube die Muffins mit etwas
Puderzucker.
```

- Backzeit implementieren → *Wiederholungen (Schleifen)*:

```
Muffins:
Verrühre 200 g Butter, 200 g Zucker, 4 Eier,1 Pck.
Backpulver,1 Pck. Vanillezucker, 250 g Mehl, 3 Eßl.
Rum;
falls es Apfelzeit ist
    füge drei mittelgroße geschälte und
    zerteilte Äpfel hinzu;
sonst
    füge 100 g Schokoladenstückchen hinzu;
fülle den Teig in Muffinförmchen;
solange die Muffins noch nicht gar sind:
    backe bei 175-200 Grad;
bestäube die Muffins mit etwas Puderzucker.
```


- Anzahl Personen berücksichtigen → Algorithmus muß man parametrisieren durch Variablen, die von außen gesetzt werden, z.B. per Kommandozeilenparameter, oder aus einem File, oder über ein GUI (Input)

```
Muffins:
Frage nach, wie viele Personen da sind;
Speichere die Antwort in der Variablen x;
y=x/4;
Verrühre y*200 g Butter, y*200 g Zucker,y*4 Eier,y Pck.
Backpulver,y Pck. Vanillezucker,y*250 g Mehl, y*3 Eßl.
Rum;
falls es Apfelzeit ist
    füge y*drei mittelgroße geschälte und
    zerteilte Äpfel hinzu;
sonst
    füge y*100 g Schokoladenstückchen hinzu;
fülle den Teig in Muffinförmchen;
solange die Muffins noch nicht gar sind:
    backe bei 175-200 Grad;
bestäube die Muffins mit etwas Puderzucker.
```

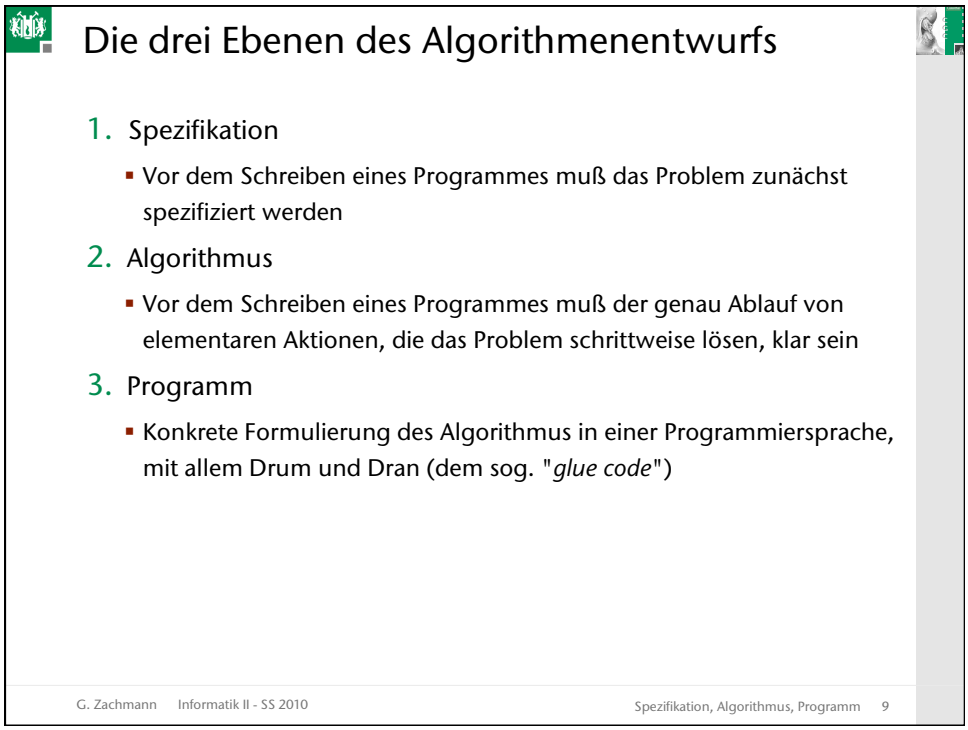
- Strukturierung für Übersichtlichkeit und Wiederverwendbarkeit von Teilen des Rezeptes → Algorithmus aufteilen auf mehrere *Teilprogramme*:

```
Muffins:
Frage nach, wie viele Personen da sind;
Speichere die Antwort in der Variablen x;
y=x/4;
Verrühre y*200 g Butter, y*200 g Zucker,y*4 Eier, y Pck.
Backpulver,y Pck Vanillezucker,y*250 g Mehl, y*3
Eßl.Rum;
falls es Apfelzeit ist:
    füge y*drei mittelgroße geschälte und
    zerteilte Äpfel hinzu;
sonst
    füge y*100 g Schokoladenstückchen hinzu;
fülle den Teig in Muffinförmchen;
solange Stäbchentest(Muffin)= "nicht fertig":
    backe bei 175-200 Grad;
bestäube die Muffins mit etwas Puderzucker
```



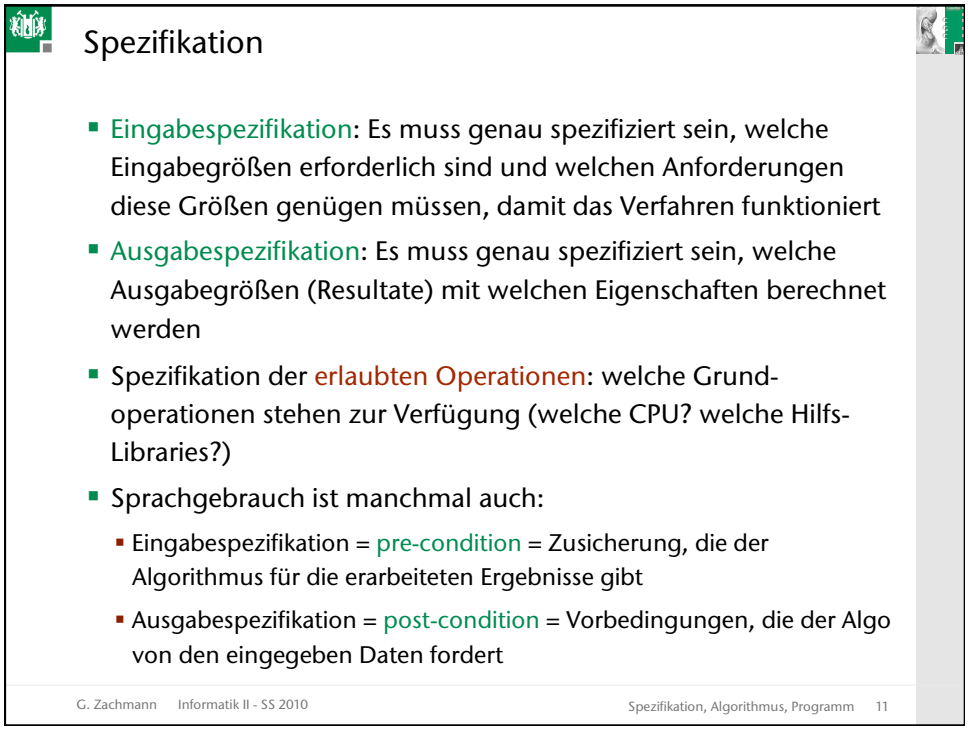
```
Unterpgroamm Stäbchentest( muffin ):
stecke ein Holzstäbchen in das muffin;
falls Teig kleben bleibt:
    antworte "noch nicht fertig";
sonst:
    antworte "fertig"
```

Der Algorithmus ist jetzt schon ganz schön kompliziert geworden. Er ist in gewisser Hinsicht noch einfach, denn es handelt sich um eine *sequentielle* Folge von Anweisungen. Das heißt, ein einziger Koch ist hier am Werk, der die Anweisungen nacheinander ausführt.
Alternativen sind *parallele* Programme, in denen mehrere Anweisungen gleichzeitig durch verschiedene Prozessoren ausgeführt werden können. Beim Backen könnte etwa der Küchenjunge schon mal die Äpfel schälen, während der Chefkoch den Teig zubereitet. Die Aktionen müssen in parallelen Programmen geeignet *synchronisiert*, d.h., abgestimmt werden. Etwa der Küchenjunge muss mit den Äpfeln fertig sein, bevor der Chefkoch sie zum Teig fügen kann.



Die drei Ebenen des Algorithmenentwurfs

1. Spezifikation
   - Vor dem Schreiben eines Programmes muß das Problem zunächst spezifiziert werden
2. Algorithmus
   - Vor dem Schreiben eines Programmes muß der genau Ablauf von elementaren Aktionen, die das Problem schrittweise lösen, klar sein
3. Programm
   - Konkrete Formulierung des Algorithmus in einer Programmiersprache, mit allem Drum und Dran (dem sog. "*glue code*")



Spezifikation

- Eingabespezifikation: Es muss genau spezifiziert sein, welche Eingabegrößen erforderlich sind und welchen Anforderungen diese Größen genügen müssen, damit das Verfahren funktioniert
- Ausgabespezifikation: Es muss genau spezifiziert sein, welche Ausgabegrößen (Resultate) mit welchen Eigenschaften berechnet werden
- Spezifikation der erlaubten Operationen: welche Grund-operationen stehen zur Verfügung (welche CPU? welche Hilfs-Libraries?)
- Sprachgebrauch ist manchmal auch:
  - Eingabespezifikation = pre-condition = Zusicherung, die der Algorithmus für die erarbeiteten Ergebnisse gibt
  - Ausgabespezifikation = post-condition = Vorbedingungen, die der Algo von den eingegeben Daten fordert

Beispiel

- Aufgabe: ggT(m, n)
  - "Für beliebige Zahlen m und n berechne den größten gemeinsamen Teiler ggT(m,n), also die größte Zahl, die sowohl m als auch n teilt"
- Spezifikationen:
  - Pre-conditions:
    - Welche Zahlen m, n sind zugelassen?
    - Positive Zahlen oder auch negative oder rationale? Ist 0 erlaubt?
  - Post-conditions:
    - Was wird ausgegeben, falls m, $n < 0$?
    - Was wird ausgegeben, falls Eingabe nicht den pre-conditions genügt?
  - Grundoperationen:
    - '+', '-' oder auch div und mod?

Beschreibung von Spezifikationen

- Formale Spezifikation = Paar P und Q von logischen Aussagen (Aussagen-/Präd.logik!)
- P : Vorbedingung / Eingabespezifikation
  - alle relevanten Eigenschaften, die vor Ausführung des Algorithmus gelten
  - Z.B.: $0 < m, n < 2^{31}$
- Q : Nachbedingung / Zusicherung
  - alle relevanten Eigenschaften, die nach Ausführung des Algorithmus gelten
  - Für das Resultat $r = \text{ggT}(m, n)$ des Algorithmus gilt

$$r = \max\{x \ : \ x \in \mathbb{N} \ \wedge \ x|n \ \wedge \ x|m\}$$

- In der Praxis werden häufig weniger formale Beschreibungen in natürlicher Sprache verwendet (Pflichtenhefte)
  - Häufig umfangreich, mehrdeutig, inkonsistent
  - Ist die Ausgabespezifikation das, was der Kunde wollte?
  - Bekommt der Kunde von seinen "Zulieferern" das, was die Eingabespezifikation sagt?
  - Tut der Algorithmus das, was in der Spezifikation steht?
  - Tut der Algo das, was der Kunde wollte?
  - Siehe Software-Engineering
- Mein Tip: versuchen Sie, für Algorithmen immer eine "möglichst formale" Spezifikation zu schreiben

Herkunft des Wortes "Algorithmus"

Muhammad ibn Musa abu Djafar al-Choresmi, ca. 780-850 n. Chr.

- arabischer Mathematiker, geboren in Choresmien (heute Usbekistan)
- arbeitete in Bagdad im "Haus der Weisheit"
- war beteiligt an der Übersetzung der Werke griechischer Mathematiker ins Arabische
- schrieb ein "Kurzgefasstes Lehrbuch für die Berechnung durch Vergleich und Reduktion" (Lehrbuch zum Rechnen mit Dezimalzahlen)
- lateinische Übersetzung dieses Buches ("liber algorismi") kam durch Kreuzfahrer nach Europa
- verfasste auch ein Buch mit dem Titel "Al-Mukhtasar fi Hisab al-Jabr wa l-Muqabala"

Zum Begriff des Algorithmus

- Definition Algorithmus:

> Geordnete Menge eindeutiger, diskreter, effektiv durchführbarer Schritte, die die gegebene Vorbedingung in endlich vielen Schritten in die gegebene Nachbedingung überführen.

- Also: eine präzise Vorschrift, wie in endlich vielen Schritten ein Problem gelöst werden kann
- Beispiel: Konvertierung zwischen Dezimal ↔ Binär

Eine sehr alte Beschreibung eines Algorithmus'

Verdoppeln einer ganzen Zahl nach Adam Ries(e) (16. Jahrhundert)

Duplieren

Lehret wie du ein zahl zweyfaltigen solt. Thu ihm also:
Schreib die zahl vor dich /mach ein Linien darunter/ heb an zu forderst / Duplir die erste Figur. Kompt ein zahl die du mit einer Figur schreiben magst / so setz die unden. Wo mit zweyen / schreib die erste / Die ander bahalt im sinn. Darnach duplir die ander / und gib darzu / das du behalten hast / und schreib abermals die erste Figur / wo zwo vorhanden / und duplir fort biß zur letzten / die schreibe gantz auß / als folgende Exempel aufweisen.

41232	98765	68704
82464	197530	137408

aus: A. Risen, Rechenbuch, 1574

Algorithmus und Spezifikation

- Eine Spezifikation $\{P\}$ $\{Q\}$ beschreibt Problem
- Ein Algorithmus A sei Lösung des Problems
- Dann ist die Schreibweise

$$\{P\}\ A\ \{Q\}$$

äquivalent zu der Aussage

> Wird der Algorithmus A in einer Situation gestartet, in der P gilt, dann gilt nach Beendigung des Algorithmus Q

- Algorithmenentwurf ist wie Gleichung mit einer Unbekannten X:

  {P} X {Q}

  - Gesucht ist ein Algorithmus X, der P in Q überführt
- Achtung: nicht jede Spezifikation hat eine Lösung
  - Beispiel: {m<0} X {$x \in$ R und x = log(m)}
- Falls eine Lösung (Algo) X existiert, gibt es unendlich viele
  - Beispiel: {m>0} X {x=2*m}
    1. Lsg.: addiere m und m
    2. Lsg.: multipliziere m mit 2
    3. Lsg.: berechne $(m+2)^2 - m^2 - 4$
- Ziel: den "kürzesten" (in gewissem Sinn) Algorithmus zu finden

- Wir gehen (zumindest hier) immer von der Terminierung eines Algorithmus A aus:
  - terminiert A nicht, ist {P}A{Q} trivialerweise erfüllt
    - ("... gilt nach Beendigung des Algorithmus Q")
  - ein nicht-terminierender Algorithmus löst aber kein Problem

Begriffe in der Definition für "Algorithmus"

- Ausführung des Algorithmus erfolgt in diskreten Schritten:
  - einzelne, einfache Elementaraktionen
  - Welche dies sind hängt vom sog. Algorithmischen Modell ab
    - Maschineninstruktionen?
    - Hochsprachliche Konstruktionen / mathem. Funktionen? (log(x) ?)
    - Bitkomplexität? (Addition beliebig langer Zahlen)
- Endliche Beschreibung: das Verfahren muss in einem endlichen Text vollständig beschrieben sein

- Geordnete Menge:
  - Reihenfolge der Schritte genügt gewissen Regeln
  - Schritte müssen nicht unbedingt nacheinander ausgeführt werden
    - es ist erlaubt, dass mehrere Schritte parallel von verschiedenen Einheiten ausgeführt werden (parallele oder verteilte Algorithmen)
    - bei einer einzigen aktiven Einheit allerdings gibt es einen ersten, zweiten, etc. Schritt (sequentielle Algorithmen)
- Eindeutigkeit: Der Verfahrensablauf ist zu jedem Zeitpunkt determiniert, d.h.,
  - zum Zeitpunkt der Ausführung eines Schrittes muß eindeutig und vollständig festgelegt sein, was zu tun ist; und,
  - gleiche Eingaben sollen zum gleichen Ablauf und Ergebnis führen.
  - Ausnahme: randomisierte Algorithmen, deren Ablauf von (mathematisch bestimmten) Zufallsgrößen abhängt
  - Weitere Ausnahme: Parallelisierung

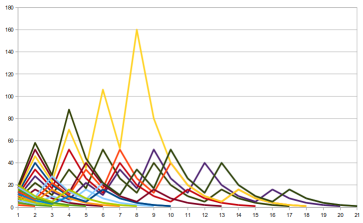
- Effektivität: Jeder Schritt des Verfahrens muß effektiv mechanisch ausführbar sein
  - Bemerkung: "Effektivität" (= erzielt das gewünschte Resultat) ist nicht zu verwechseln mit "Effizienz" (= Wirtschaftlichkeit)
- Folgendes ist z.B. kein Algorithmus zur Gesichtserkennung:
  1. Hole ein Bild von der Kamera (z.B. per Library-Funktion)
  2. Lege um jedes Gesicht in dem Bild eine Ellipse
  3. Vergleiche jede Ellipsenregion des Bildes mit allen Ellipsenregionen anderer Bilder, die schon als Gesicht klassifiziert wurden und einen Namen haben

  ⇒Ein Mensch kann ziemlich schnell und sicher Schritt 2 ausführen, ein Computer aber nicht

  ⇒Fazit: dies ist höchstens der Ansatz zu einer Idee zu einem Gesichtserkennungsalgorithmus

- Terminierung: Der Algorithmus hält nach endlich vielen Schritten mit einem Ergebnis an — sofern die Eingaben der Eingabespezifikation genügt haben!
- Nicht erlaubt sind z.B. Schrittfolgen, die nie ein Ergebnis liefern:
  - Starte mit Zahl 0
  - Addiere 2
  - Falls Ergebnis ungerade, halte an, sonst weiter bei Schritt 2.
- Sonderfälle:
  - Betriebssystem, Datenbanksystem, ... (haben eine *infinite main loop*)
  - Bei numerischen Algorithmen (z.B. Nullstellenberechnung), möchte man häufig offen lassen, wann genau die Berechnung in der Praxis abbrechen soll

- Achtung: es gibt Algorithmen(?), von denen man nicht *weiß*, ob sie in jeder Situation terminieren!
- Beispiel: die sog. Ulam-Funktion bzw. das sog. Collatz-Problem
  1. Starte mit beliebiger positiver ganzer Zahl *n*
  2. Falls *n* ungerade, ersetze *n* durch 3*n*+1, sonst ersetze *n* durch *n*/2.
  3. Fahre fort, bis *n* = 1 erreicht ist.
     - Nicht bekannt, ob dieser Algo für beliebige Startwerte terminiert!
- Collatz-Graph für die Startwerte 1...20



Korrektheit

- Partielle Korrektheit: Jedes berechnete Ergebnis genügt der Ausgabespezifikation, sofern die Eingaben der Eingabespezifikation genügt haben
  - D.h., Ein- und Ausgabeparameter genügen im Falle der Terminierung immer den Spezifikationen
  - Achtung: partielle Korrektheit sagt nichts über die Terminierung!
- Totale Korrektheit: Ein Rechenverfahren ist ein total korrekter Algorithmus, wenn es partiell korrekt ist und für jede Eingabe, die der Eingabespezifikation genügt, terminiert.
  - Unterschiedliche Beweisverfahren für beide Fälle notwendig
  - Manchmal ist es sinnvoll, auf die Anforderung der Terminierung zu verzichten

Ein erweiterter Algorithmenbegriff

- Ausnahmen ... bestätigen die Regel: nach unserer Definition gäbe es nämlich keine
  - nicht-terminierende Algos (oft wird aber ein Algo als solcher akzeptiert, wenn er manchmal, oder für manche Eingaben, kein Ergebnis liefert, wenn das beweisbar "selten genug" vorkommt)
  - nicht-deterministische, randomisierte Algos (Ablauf und Ergebnis kann, bei gleicher Eingabe, jedes Mal anders sein)
  - ...
- Solche Algorithmen machen aber durchaus Sinn!
- Bemerkung: man bezeichnet ein Betriebssystem i.A. nicht als Algorithmus

The Model of Computation (computational model)

- Das *Model of Computation* definiert, welches genau die Elementaren Aktionen sind und welche Kosten diese haben
- Für diesen Kurs gehen wir von einer "real RAM" aus := RAM (random access machine), wobei jede Speicheradresse genau einen Float mit einheitlicher und endlicher Präzision speichert (z.B. 4 Bytes)
- Unsere elementare Aktionen (primitive Operationen) sind
  - Aus Speicher lesen
    - Benutzung eines Variablennamens im Programm "auf der rechten Seite"
    - auch Array-Elemente
  - In Speicher schreiben (Zuweisung eines Wertes zu einer Variablen)
  - Die "üblichen" Operatoren und Funktionen:
    - Grundrechenarten (+, -, ×, ÷), Vergleiche (<, ≤, ≠, ...), k-te Wurzel, trig. Funktionen, e^x, log, etc.

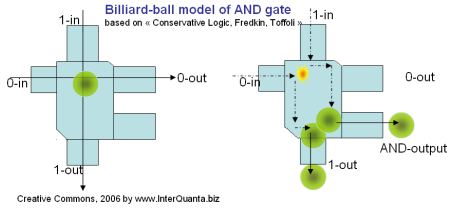
- Funktionsaufrufe
- Einen Float oder Integer ausgeben

- Alle elementaren Operationen haben Zeit-Kosten = 1

- Vorteil dieser Definitionen: Entwicklung von Algorithmen kann weitgehend ohne eine konkrete Maschine und ohne konkrete Programmiersprache erfolgen!

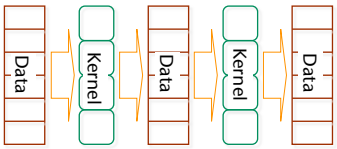
Exkurs: andere Computational Models

- Es gibt 100e andere Computational Models!
- Turing-Maschine
- Quanten-Computer:
  - Elementare "Speicher-Größe" = Qubit = der Spin eines Elektrons
  - Operationen u.a.: Superposition und Verschränkung
- Billiard-ball computer:
  - Basiert auf idealen, reibungsfreien Kugeln unter Newton'scher Mechanik



Zum Selber-Experimentieren:
http://jameslin.name/bball/

- Stream Programming Model:
  - *"Streams of data passing through computation kernels."*
  - Stream = Array von einheitlichem Element-Typ
  - Kernel = konventionelles Programm, das gleichzeitig auf alle Elemente angewandt wird, und gleichzeitig jeweils ein neues Element produziert




```
stream A, B, C;
kernelfunc1( input: A,
             output: B );
kernelfunc2( input: B,
             output: C);
```

  - Eine aktuelle Inkarnation: Ihre Graphikkarte

Darstellung / Beschreibung von Algorithmen

- zahlreiche Methoden, Algorithmen darzustellen, z.B.
  - informelle Beschreibung = "Roman" (s. Adam Ries)
  - Flussdiagramme
  - Struktogramme
  - Pseudocode
  - UML-Diagramme (s. Software-Engineering)
  - Programmcode
  - ...

Pseudocode

- Häufigstes Hilfsmittel zur Formulierung eines Algorithmus
- Syntax ist nicht genau festgelegt
  - hier: ähnlich zu Python
  - sollte intuitiv zugänglich sein
- Auch verbale Beschreibung von Blöcken sind möglich
  - z.B. "Daten einlesen" oder "alphabetisch sortieren"
- Ist leicht in imperative Programmiersprache übertragbar

Beispiel: ggT

- Eigenschaften von t = ggT(m, n)
  1. Fall $m = n$: ggT(m, m) = m
  2. Fall $m > n$: ggT(m, n) = ggT(m-n, n)
     - Denn:

$$t|m \ \wedge \ t|n \ \Rightarrow \ m = ta \ \wedge \ n = tb \ \Rightarrow \ m - n = (a - b)t$$

  3. Fall $m < n$: ggT(m, n) = ggT(n, m)

- Dies ist im Prinzip schon eine Beschreibung des Algorithmus' in Pseudocode!

- Der Pseudo-Code:

```
Algorithm ggT:
Input:  m,n ∈ N, m, n > 0   (precondition)
Output: t ∈ N, t > 0        (postcondition)
while m ≠ n:
  if m < n:
    swap m,n
  m ← m - n
output m
```

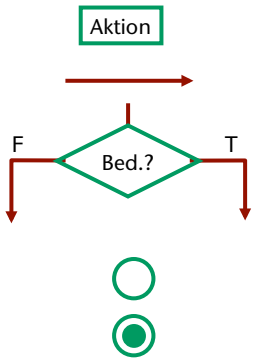

Noch ein Beispiel

- Finden des größten Elements in einem Array:

```
Algorithm maxOfArray( a, n ):
Input: a = array of integers
       n = Anzahl Elemente in a
Output: max{ a[0], …, a[n-1] }
        Index m mit a[m] = max
currentMax = a[0]
indexOfMax = 0
for i = 1 .. n-1:
  if a[i] > currentMax:
    currentMax = a[i]
    indexOfMax = i
output currentMax, indexOfMax
```

Flussdiagramm (*flow chart*)

- Graphische Notation für Algorithmen
- Intuitiv verständlich
- Kann bei umfangreichen Algorithmen unübersichtlich werden
- Die graphischen Elemente:



Auszuführende Elementaraktion

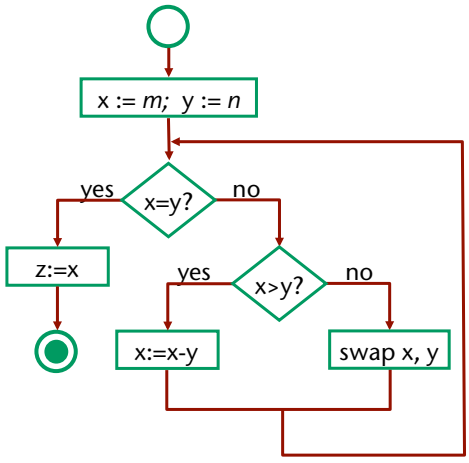
Reihenfolge der Elementaroperationen

Teste Bedingung in der Raute;
ist sie erfüllt, weiter bei T,
sonst weiter bei F

Beginn der Ausführung des Algorithmus

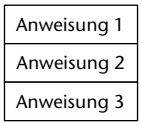
Ende der Ausführung des Algorithmus

Der ggT-Algorithmus als Flussdiagramm

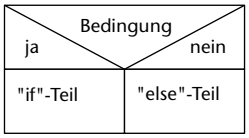


Struktogramme

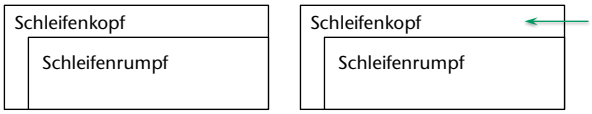
- Auch bekannt als Nassi-Shneiderman-Diagramme:
- Sequenz von Anweisungen:



- Verzweigung / Bedingung (**`if`**):



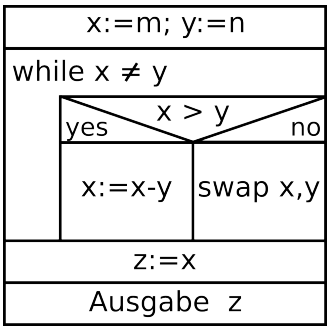
- Schleifen (**`while`**, **`repeat .. until`**, **`for ..`**):



Z. B.:
- zähle i von 1 .. 100
- while Auto fährt noch

Beispiel

- Der ggt-Algorithmus als Struktogramm:



Anwendung von Moore's Law auf Algorithmen

- Moore's "Law": Die Anzahl Transistoren pro Chip-Fläche verdoppelt sich alle 2 Jahre
- Konsequenzen:
  - Die Leistungsfähigkeit (Geschwindigkeit) verdoppelt sich alle 2 Jahre
  - Speicherkapazität verdoppelt sich alle 2 Jahre
  - Problemgrößen verdoppeln sich alle 2 Jahre
- Konsequenz:
  Auf der neuen Maschine braucht ein Algorithmus länger!

  - "Beweis":
    - Auf alter Maschine: Programm benötigt Zeit (in Sek.) $n \log n$
    - Auf neuer Maschine: Programm bekommt doppelt großen Input und benötigt

$$\frac{1}{2} 2n \log(2n) = n \log n + n$$

- Konsequenz: Entwicklung schneller Algorithmen lohnt sich!