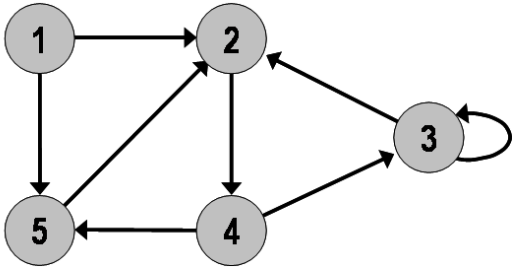
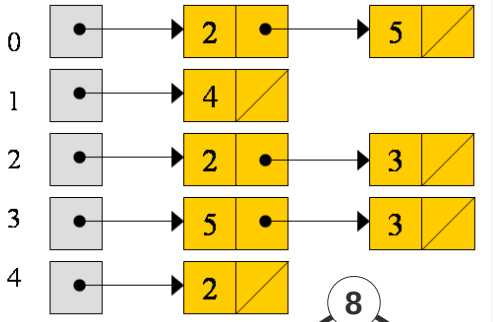
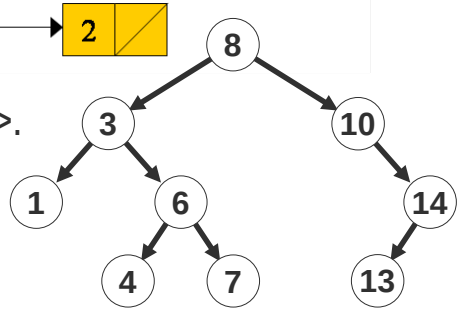
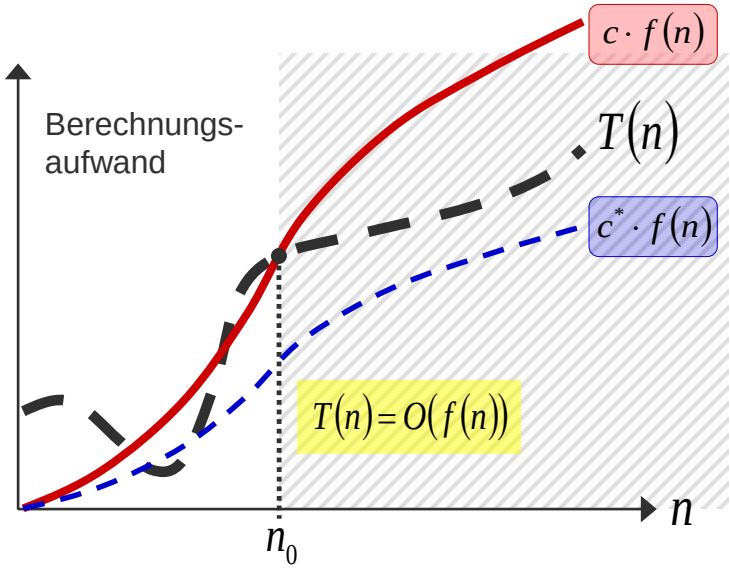
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Fachbereich Informatik




<Zahl> ::= <Ziffer> | <Ziffer> <Zahl>.
<Ziffer> ::= “0“ | “1“.
<Op> ::= “*“ | “/“ | “+“ | “-“.

# Informatik II – SoSe 2018
# Algorithmenkonstruktion

**Martin Butz, martin.butz@uni-tuebingen.de**

# Erinnerung:

- Abgabesystem – InfoMark
  (Anmeldung bitte bis Donnerstag)

  **https://info-cg.informatik.uni-tuebingen.de**
-

- Bitte dann Übungspräferenzen eingeben:
  - Bitte **mehrere Gruppen** wählen!
  - ➢ Wer nur eine Präferenz oder gar keine Präferenz wählt, wird automatisch für alle nicht gewählten Gruppe auf niedrige Präferenz gesetzt!

- Verteilung auf Übungsgruppen wird am Wochenende bekannt gegeben

# II. Aspekte der Algorithmen-konstruktion

1. Der Algorithmenbegriff
2. Konstruktion
3. Eigenschaften und Analyse von Algorithmen

**Lernziele**

Algorithmen stehen im Zentrum der Informatik.
Deren Entwurf, Analyse, Realisierung und Bewertung ist Gegenstand dieser Lehrveranstaltung.

Jetzt:
Entwicklung eines Grundverständnisses für fundamentale Konzepte und Methoden

Nach diesen VL-Folien sollten Sie in der Lage sein …

- Algorithmen und ihre Eigenschaften einzuordnen und deren Charakteristika zu identifizieren;
- Eine Definition für Algorithmen zu geben;
- Die wesentlichen Phasen des Algorithmen-/Programmentwurfs zu skizzieren;
- Darstellungsmethoden für kleine Programme anzuwenden;
- Einen Algorithmus auf seine Korrektheit zu prüfen und dabei die partielle sowie die totale Korrektheit zu analysieren.

# DER ALGORITHMENBEGRIFF

- Eine Begriffsbestimmung
- Beispiele aus dem Alltag
- Definition – Algorithmus
- Historischer Überblick

# Begriffsklärung I

Ein **Algorithmus** ist eine Vorschrift oder eine Anleitung zur **Lösung einer Aufgabenstellung**, die so **präzise formuliert** ist, dass man sie im Prinzip „mechanisch“ ausführen kann.

- **Mit einem Algorithmus** verfolg man ein **Ziel**:
  … man will etwas realisieren;
  … oder zumindest eine Frage klären oder ein Problem lösen.

- Typischerweise hat das Ziel etwas mit **Information** oder **Kontrolle** zu tun.
  - Man will Daten analysieren, um Informationen zu extrahieren.
  - Man will Systeme kontrollieren, um sie zu verbessern:
    - Benutzerfreundlichkeit
    - Energieeffizienz
    - Realitätsnahe Simulation
    - Robotersteuerung
    - …

# Begriffsklärung II

- Ein Algorithmus (Verarbeitungs-Vorschrift) wird als **Nachricht** in einer (Programmier-) Sprache formuliert, die vom Ausführenden (Mensch, Rechner) **interpretiert** werden muss

**Randbedingungen:**

- Verschiedene Formalismen oder Darstellungsformen sind möglich.
- Ein gemeinsames Sprachverständnis ist Voraussetzung.
- „Präzise formuliert“ heißt, die „mechanischen“ Möglichkeiten des Ausführenden zu kennen und in der Formulierung zu nutzen.

# Algorithmen im Alltag

- Gebrauchsanweisung
- Bedienungsanleitung
- Rezepte zur Zubereitung von ...
- Anleitungen zum „Basteln“ eines / einer …
- Vorschriften nach denen man sich richtet (oder auch nicht)
- Prozessabläufe (z.B. Waschmaschine)
- Wegbeschreibung
- ….

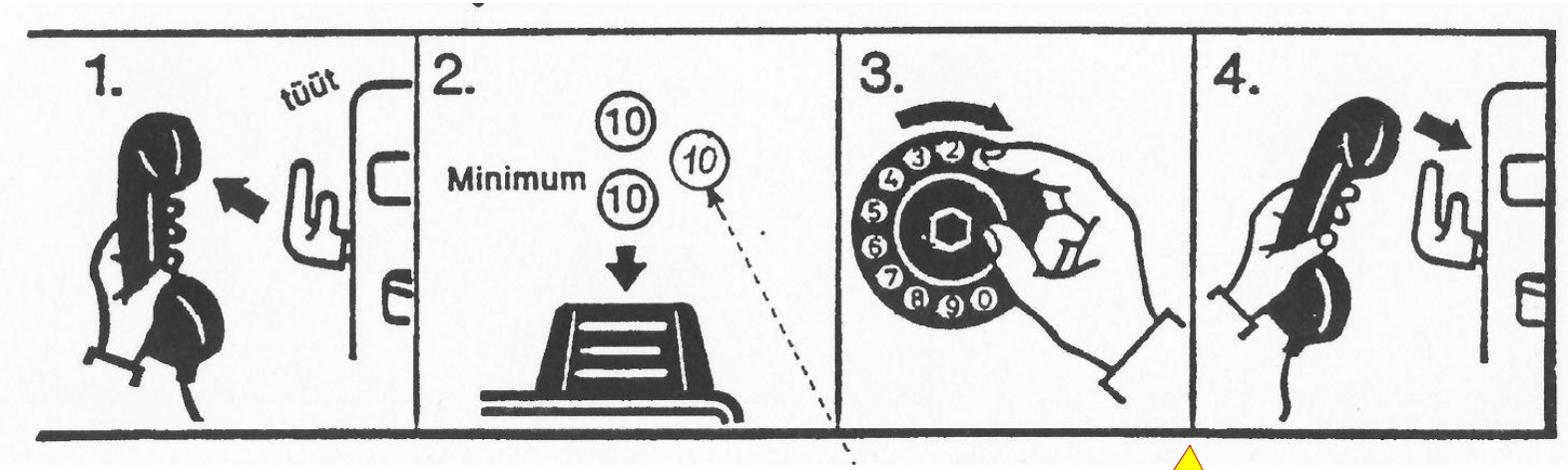
# Charakteristika

- Ein **Algorithmus** (seine Beschreibung) basiert auf seiner **Reproduzierbarkeit**.

- Algorithmen-Definitionen werden durch bestimmte typische Aspekte charakterisiert:
  - **Eindeutigkeit**
  - **Elementare Eigenschaften eines Algorithmus**
  - **Schrittweise Verfeinerung**
  - **Bedingte Schritte in einem Algorithmus**
  - **Wiederholte Schritte in einem Algorithmus**

# Beispiel aus der Vergangenheit

- Anleitung zur Benutzung eines Münzfernsprechers



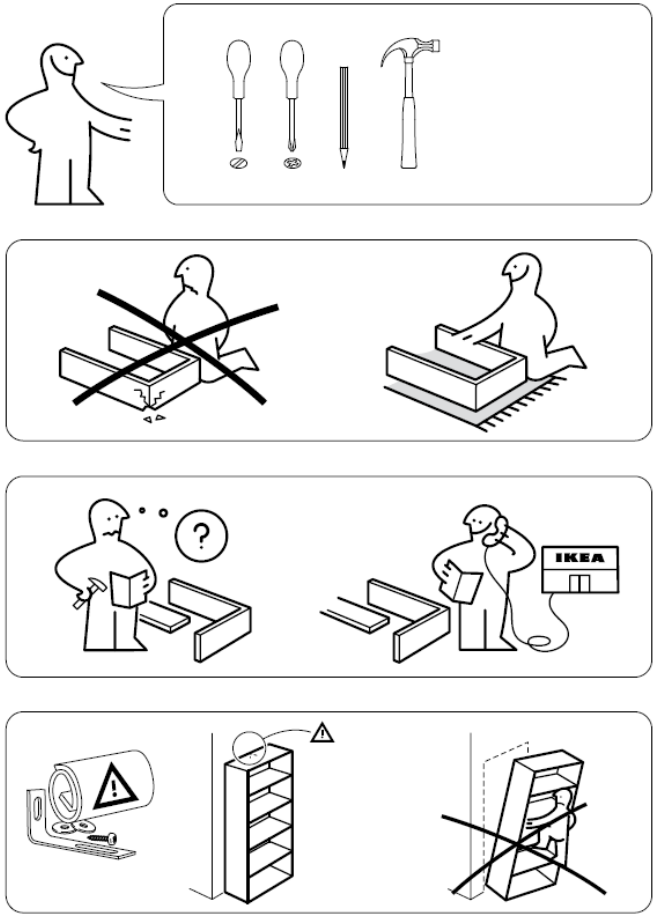

1. Hörer abnehmen; Wählton abwarten
2. Mindestens 30 Cents einwerfen
3. Nummer wählen
4. Hörer einhängen

Das gemeinsame Verständnis ist hier natürlich, dass dazwischen ein Gespräch geführt wird …



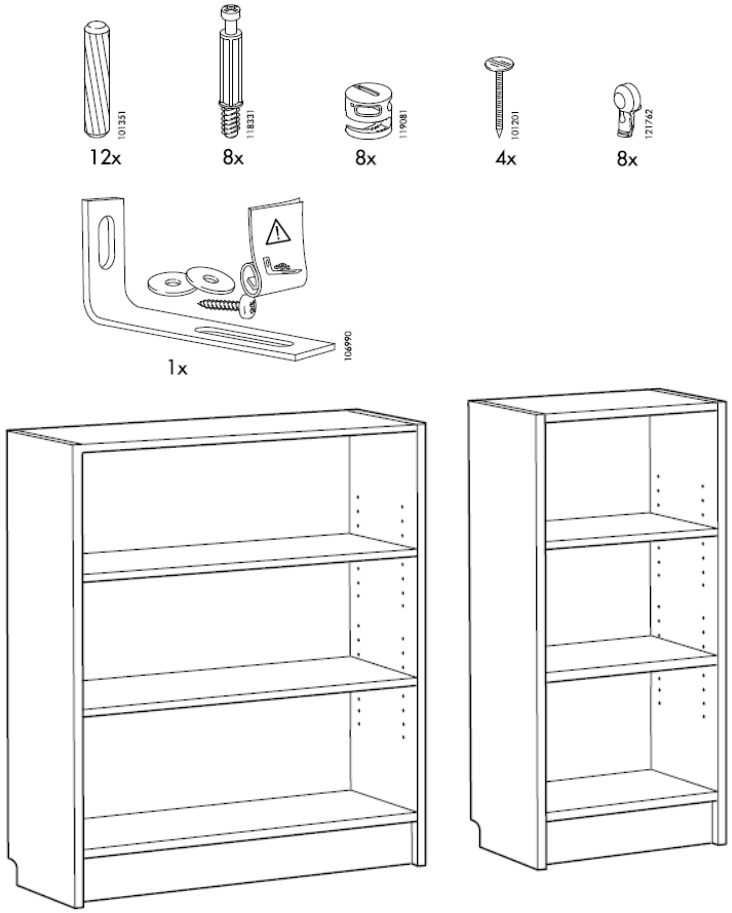
(F.L. Bauer, G. Goos. Informatik – Eine einführende Übersicht, erster Teil, 3. Aufl. Springer, Berlin, 1982)

# Beispiel aus dem Alltag – IKEA Regal Billy (I)




- **Voraussetzungen**
  - Werkzeuge (zur Ausführung von Instruktionen)
  - Unterlage (zur sicheren Bearbeitung)
  - Support Informationen (wer hilft wenn etwas unklar ist)
  - Sicherheitshinweise (was sollte man besser nicht tun)

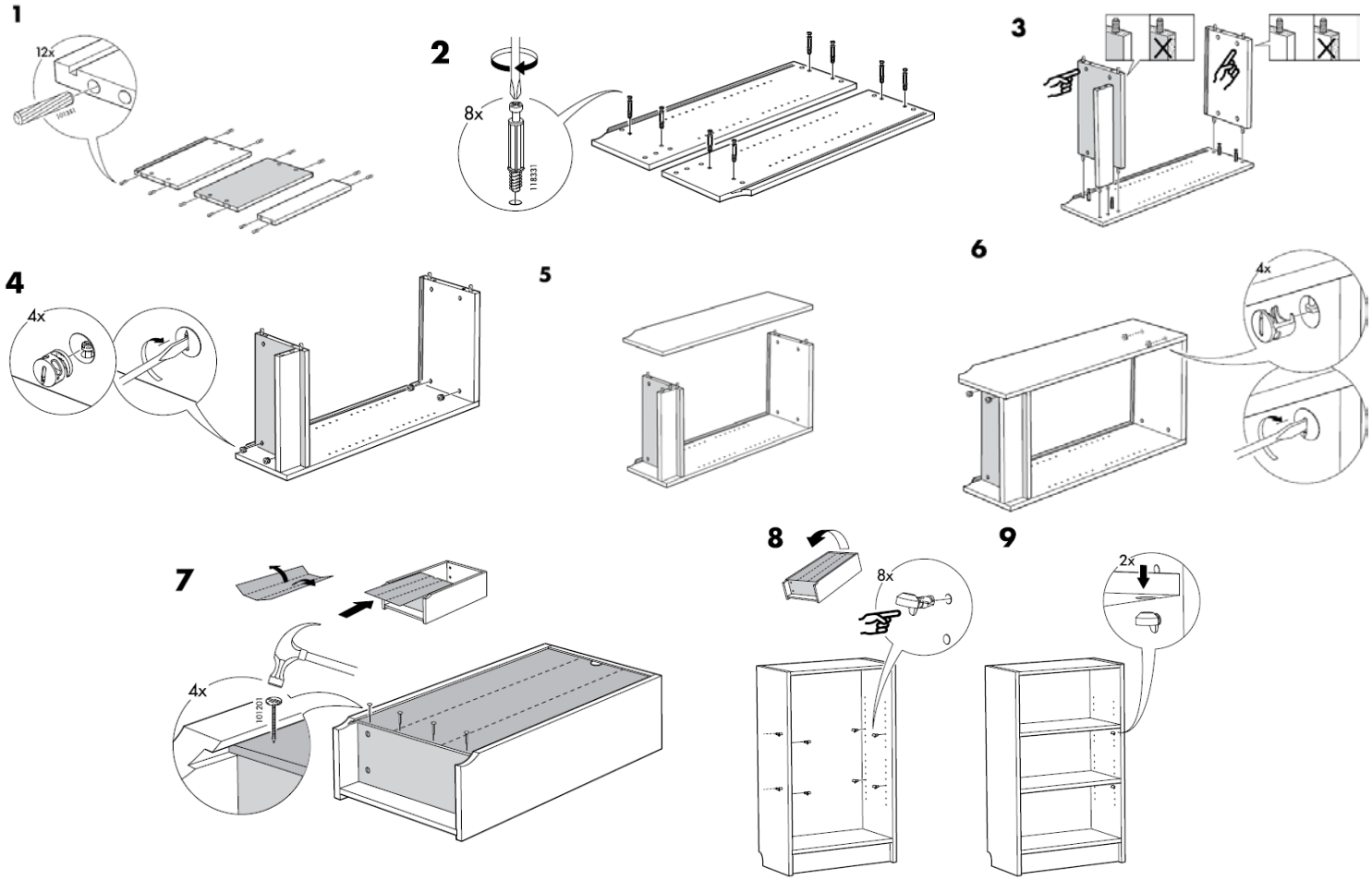
# Beispiel aus dem Alltag – IKEA Regal Billy (II)




- **Material („Daten“)**
  - Schrauben
  - Nägel
  - Andere Teile ☺
  - Bretter
- Ziel: „Billy“

# Beispiel aus dem Alltag – IKEA Regal Billy (III)

## „Algorithmus“ (Abfolge von „Datenverarbeitungsschritten“)



# Kontrast: Alltagsalgorithmus VERSUS Code

## Alltagsalgorithmus

- Voraussetzungen
  - Material (alle möglichen Dinge, Gadgets, Objekte…)
  - Werkzeuge & Manipulationsfähigkeiten
    - Schraubenzieher, Geld
    - Typische menschliche Fähigkeiten (Schraube befestigen / fest ziehen, Geld einwerfen, Tasten drücken…)
- Instruktionsabfolge von Operationen
  - Intuitive Instruktionen, die komplexe Operationen verlangen.
  - Erwartet einen „intelligenten Menschen“, der diese versteht und ausführen kann.
  - Der „Prozessor“ ist also der Mensch.

## Code-orientierter Algorithmus

- Voraussetzungen
  - Daten / Werte / Eingaben / Sensoren
  - Menge von Instruktionen wie z.B.
    - Mathematische Operatoren
    - Zuweisungen
    - Vergleichsoperatoren
    - Schleifen (Wiederholungen)
- Instruktionsabfolge
  - Eindeutige, wohldefinierte Abfolge von Operationen.
  - Operationen können exakt und eindeutig von einer „dummen“ Maschine (z.B. Computer) ausgeführt werden.
  - Der „Prozessor“ ist also eine Maschine, die selbst nach exakt definierten Regeln arbeitet.

# Elementare Eigenschaften eines Algorithmus (1)

- **Ausführbarkeit** und **Reproduzierbarkeit**
  Die Anleitung ist ausführbar und reproduzierbar
  (das heißt, sie ist „operativ“)

- **Prozessor** und **Prozess**:
  - Die **Ausführung erfolgt schrittweise**.
  - Die Folge der Ausführung von Schritten eines Algorithmus nennt man den von diesem Algorithmus beschriebenen **Prozess**.
  - Der Ausführende des Algorithmus heißt **Prozessor**.

# Elementare Eigenschaften eines Algorithmus (2)

- **Teilalgorithmen** und **elementare Verarbeitungsschritte / Operationen**:
  - Jeder Schritt besteht selbst wieder aus der Ausführung eines oder mehrerer Teilalgorithmen (z.B. „Hörer abnehmen“, Schrauben befestigen).
  - Nicht mehr verfeinerte Teilalgorithmen werden als **elementare Operationen** bezeichnet.

- Der Ausführende (Prozessor) muss
  - die Sprache, in welcher der Algorithmus verfasst ist, kennen;
  - die elementaren Algorithmen bzw. Operationen/Aktionen verstehen; und
  - in der Lage sein, diese auszuführen.

# Elementare Eigenschaften eines Algorithmus (3)

- Ein Algorithmus muss hinreichend **genau** sein, d.h. alle **Schritte** und deren **Ausführungsreihenfolge** müssen eindeutig feststehen.

- Die Formulierung eines Algorithmus setzt deshalb ein wohl gewähltes **Abstraktionsniveau** voraus (d.h. einen bestimmten **Detaillierungsgrad** ≡ Ebene der elementaren Operationen).

- Gegenbeispiel:
  - Führe, falls Linkshänder, die rechte Hand, falls Rechtshänder, die linke Hand zum Hörer.
  - Hebe diesen um 10 cm senkrecht nach oben und führe die obere Muschel des Hörers an das Ohr, und zwar, falls Linkshänder, ans rechte Ohr, falls Rechtshänder …

  - Zu detailliert und für den Menschen nicht „zweckmäßig“.
  - Wieso eigentlich nicht „zweckmäßig“?
    - ➢ Weil der Mensch viele Annahmen in einen Algorithmus hineininterpretiert und fehlende Informationen auf natürliche Weise vervollständigt.
  - Im Gegensatz dazu ein Computer bezüglich eines Programms:
    - Computer macht (fast) keine Annahmen.
    - Alle Operationen müssen vollständig und eindeutig spezifiziert sein.

# Elementare Eigenschaften eines Algorithmus (4)

- **Endlichkeit der Beschreibung**:
  Der vollständige Algorithmus muss in einem **endlichen Text** formuliert und aufgeschrieben sein.

- **Termination** des Algorithmus:
  Der Algorithmus muss nach **endlich vielen Schritten** (bei ggf. wiederholter Ausführung bestimmter Schritte) **zum Ende** kommen!

# Weitere Eigenschaften von Algorithmen

- Ausführung von Teilen eines Algorithmus:
  - Einige Teile sollen **nacheinander** (**sequentiell, seriell**) angewendet werden;
  - Andere sollen **gleichzeitig** (**parallel**) ausgeführt werden;
  - Wiederum andere Teile können in **beliebiger Reihenfolge** (auch gleichzeitig) ausgeführt werden (**kollateral**).

- **Strukturelemente** (Ausführung eines Algorithmus):
  - **Befehl bzw. Operation**
  - **Befehlssequenz**
  - **Wiederholungen von Befehlen oder Befehls-Sequenzen**
  - **Bedingte Auswahl**
  - Mechanismen zur **Zusammensetzung der Strukturelemente**.

# Algorithmen und Daten

- **Eingabe**objekte, **Ausgabe**objekte und **lokale Daten**

- Ein Algorithmus **beschreibt Operationen auf benannten Daten (meist „Objekte“ in Java)**
  - z.B. Hinzufügen (Operation) von Öl (Objekt)
  - Man unterscheidet Eingabeobjekte (werden zur Ausführung benötigt),
  - Ausgabeobjekte (werden durch Ausführung des Algorithmus erzeugt)
  - und lokale Objekte (Objekte existieren nur während des Prozesses).

- Objekte sind
  - **elementar**
    (im Algorithmus als unteilbare Einheiten aufgefasst) oder
  - **zusammengesetzt**
    (einfache Daten oder strukturierte Daten).

# Beschreibung von Algorithmen

- Der Algorithmus (hier nicht vollständig) besteht aus der Beschreibung einiger benannter **Teilalgorithmen**.
- Diese Teilalgorithmen werden zunächst **definiert** und dann in der Definition zusammengesetzter Algorithmen durch Nennung ihrer Namen **„aufgerufen“.**
- Teilalgorithmen können **Parameter** haben, d.h. das Ergebnis ihrer Ausführung hängt von den beim Aufruf erfolgten Angaben ab.
- Teile eines Algorithmus können weiderholt werden (**Wiederholungsanweisung**).
  - Beispiel: Schublade bauen.
  - Wie oft der Teil zu wiederholen ist, wird in der Benutzung festgelegt und hängt häufig von den Daten ab (Anzahl Schubladen).

# Meta-Information – Korrektheit

- Häufig wird zwischendurch geprüft, ob bisher alles (hinreichend) korrekt ausgeführt wurde.
- Solche Teile des Algorithmus sind typischerweise
  - Nicht operativ,
  - Enthalten keine Handlungsanweisungen,
  - Sondern **dienen lediglich zur Überprüfung** des korrekten Ablaufs (d.h. der **Korrektheit**) des Prozesses.
- Manchmal führen sie aber auch zu Korrekturen im Ablauf
  - Detektion eines Fehlers wie zum Beispiel
    - z.B.: Schubladenseite falsch rum eingesetzt? -> Vorgang korrigieren.
  - Oder auch Kontrolle der Qualität
    - z.B.: Abschmecken beim Kochen… zu wenig Salz? -> nachsalzen.

# Definition – Algorithmus

- Aus der Betrachtung der vorherigen Fallbeispiele und ihrer Eigenschaften ergibt sich folgende **Definition** eines Algorithmus und seiner grundsätzlichen *Charakteristika*:

*Ein **Algorithmus** ist eine präzise (d.h. in einer festgelegten Sprache abgefasste) endliche Beschreibung eines allgemeinen (vom Ausführenden interpretierbaren) Verfahrens zur Lösung einer Aufgabe / eines Problems unter Verwendung ausführbarer, elementarer Verarbeitungsschritte mit einer endlichen Gesamtausführungsdauer.*

# Historischer Überblick

**300 v. Chr.**

**Euklids Algorithmus** zur Bestimmung des größten gemeinsamen Teilers zweier ganzer Zahlen (ggT) (beschrieben im *7.Buch der Elemente*). Beispiel: ggT(300, 200) = 100

Mittels dieses Algorithmus lässt sich der größte gemeinsame Teiler zweier ganzer Zahlen sehr effizient berechnen; ggT ist die erste Beschreibung eines Verfahrens, das modernen Kriterien für Algorithmen gerecht wird!

**800 n. Chr.**

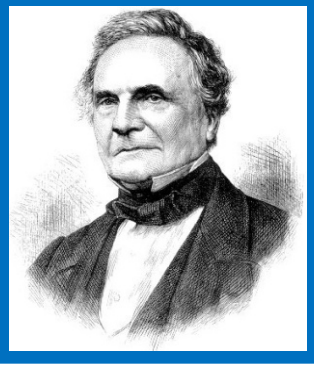
Der persisch-arabische Mathematiker ***Muhammed ibn Musa abu Djafar alChoresmi*** (auch: *al Chworesmi* oder *al Charismi*) veröffentlicht eine Aufgabensammlung für Kaufleute und Testamentvollstrecker, die später ins Lateinische als *Liber Algorithmi* übersetzt wird.

# Rechnerentwicklung und Programmierung II

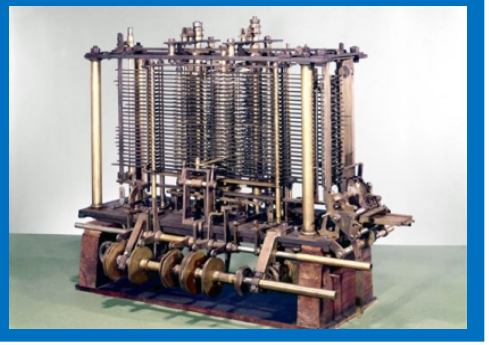
| | |
|---|---|
| **1574**   | **Adam Rieses** Rechenbuch verbreitet mathematische Algorithmen in Deutschland. |
| **1614** | Die ersten Logarithmentafeln werden algorithmisch berechnet – ohne Computer dauerte dies mal gerade 30 Jahre. |
| **1703** | Binäre Zahlensysteme werden von **Leibniz** eingeführt; diese Zahlensysteme bilden die Grundlage der internen Verarbeitung in modernen Computern. |
| **1815**  | **Augusta Ada Lovelace**, die erste „Computer-Pionierin“, wird geboren; sie entwickelte schon früh Konstruktionspläne für verschiedene Maschinen, wird Assistentin von **Babbage** (siehe nächste Folie) und entwirft Programme für erste Rechenmaschinen. |

# Rechnerentwicklung und Programmierung III

**1822**



**Charles Babbage** entwickelt die sog. *Difference Engine*, die in einer verbesserten Version 1833 fertig gestellt wird; später entwickelt er auch die *Analytical Engine*, die bereits die wichtigsten Komponenten eines Computers umfasst, aber niemals vollendet wird.



**1886**

**H. Hollerith** konstruiert elektromagnetische Lochkarten-maschine, die mit Erfolg zur 11. amerik. Volkszählung (Zensus) eingesetzt wird („Programmierung“ durch Stecktafeln).

**1931**

Gödels Unvollständigkeitssatz beendet den Traum vieler damaliger Mathematiker, die gesamte Beweisführung aller Sätze in der Mathematik mit algorithmisch konstruierten Beweisen durchzuführen.

# Rechnerentwicklung und Programmierung IV

| | |
|---|---|
| **1936** | Die Church'sche Theorie vereinheitlicht die Welt der Sprachen zur Notation von Algorithmen, indem für viele der damaligen Notationen die gleiche Ausdrucksmächtigkeit postuliert wird. |
| **1941**  | **Konrad Zuse** plant und baut die erste funktionsfähige, programmierbare Rechenmaschine der Welt (ZUSE Z-3)<br>• Addition / Subtraktion über 6 dekadische Zählräder<br>• Multiplikation / Division über drehbare Zylinder<br>(basiert auf Napiers Rechenstab-Prinzip; neu: Übertragungszähler zw. jeder Dekade). |
| **1943** | COLOSSUS: Spezial Röhren-Rechner für die erfolgreiche De-Codierung der ENIGMA-Chiffren der deutschen Wehrmacht (z.B. im U-Boot Krieg; Kryptologie). |
| **1944** | **H. Aiken** (Harvard-Univ., zus. mit IBM) baut Amerikas ersten programmgesteuerten Rechenautomaten. |
| **1945** | **ZUSE Z-4** wird funktionstüchtig. |

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

# Rechnerentwicklung und Programmierung V

| | |
|---|---|
| **1945/6** | **ENIAC**: erster Elektronenröhren-Rechner (*1. Computer-Generation*) – Berechnung ballistischer Bahnen (Geschosse, Raketen, etc.) |
| **1950** | Beginn der Förderung von Rechnerentwicklungen in Deutschland durch die DFG |
| **1953** | Erster Magnetkernspeicher |
| **1955** | Halbleiter-Transistoren lösen Röhren ab (*2. Computer-Generation*) |
| **1957** | Entwurf der ersten „höheren“ Programmiersprachen |
| **1962** | Miniaturisierung der Transistoren führt zu höheren Rechengeschwindigkeiten und geringerem Platzbedarf: Beginn der *3. Computer-Generation* |
| **1968** | Integrierte Schaltkreise in Miniaturausführung |
| **1978** | Hochintegrierte Schaltkreise |

… und es natürlich noch weiter….

# KONSTRUKTION, EIGENSCHAFTEN UND ANALYSE VON ALGORITHMEN

- Vom Problem zum Algorithmus und zum Programm
- Eigenschaften und Aussagen über Algorithmen
- Darstellung von (einfachen) Algorithmen
- Partielle und totale Korrektheit

Vom Problem (Ziel / Idee / Aufgabe) zum Algorithmus und zum Programm

# ALGORITHMENDESIGN

# Vom Problem zum Algorithmus und zum Programm

## Von Alltagsformulierungen zum maschinellen Programm

Beim Übergang von einem (evtl. umgangssprachlich) formulierten Problem / Ziel / Aufgabe,

über einen abstrakten Algorithmus,
der das Problem lösen soll,

bis zu einem Programm,
welches wiederum auf einer Maschine ablaufen soll,

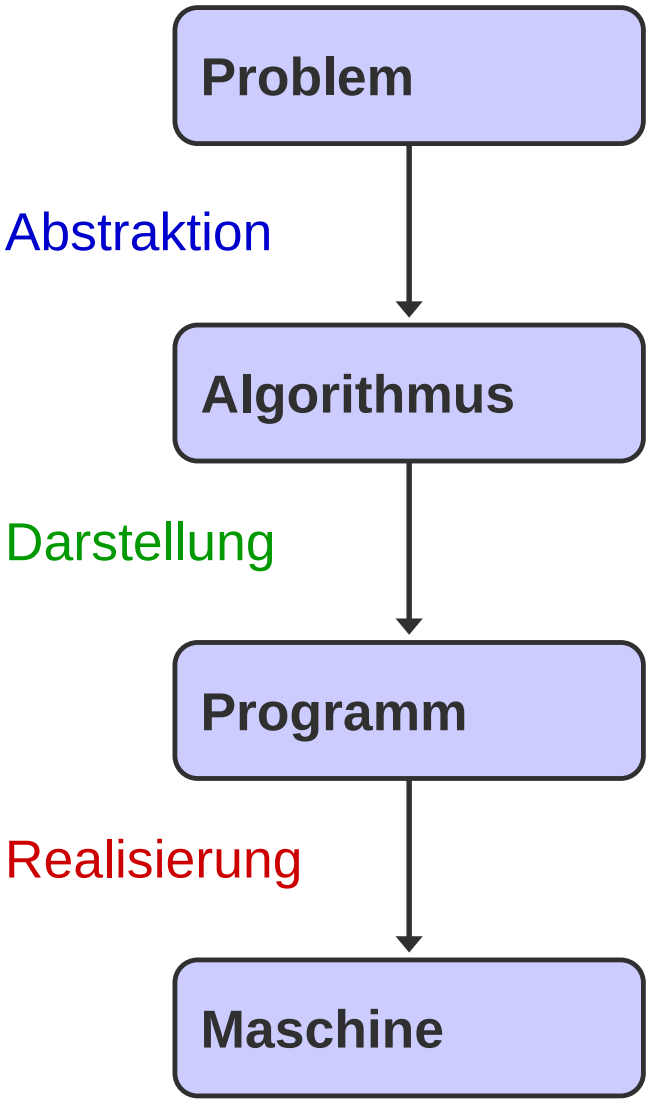
sind typischerweise mehrere Phasen zu durchlaufen.

# Phasen

Reale Welt

Abstrakte Objekte

Informations-verarbeitendes System




- *Problemanalyse*
- *Funktionale Spezifikation*
- *Algorithmenentwurf*

- *Darstellung in formalisierter Sprache (Programmier-Sprache) zur Ausführung durch Rechner*

- *Umsetzung in Elementaroperationen einer Maschine (Maschinensprache / Assembler)*

# Abstraktion – Problemformulierung und Analyse

1. Problemformulierung (Aufgabe):

   *„Finde die jüngste Person im Raum!“*

2. Problemanalyse:

   *Wie kann die Lösung aussehen? Gibt es überhaupt eine? Genau eine?*

   Problemspezifikation:

| | |
|---|---|
| *Was ist „Alter“?* | Positive ganze Zahl |
| *Was ist „jüngste“?* | Ordnungsrelation bei Zahlen |
| *Was heißt „finde“?* | Altersvorgaben müssen gegeben sein |
| *Was sind „Personen“?* | Altersangaben $a_1$ … $a_n$, die mit Personen 1, …, n indiziert sind |

# Darstellung des Alters

- Im letzten Jahrtausend geboren
- 22 Jahre
- 24 Jahre, 7 Monate, 4 Tage, 12 Stunden, …..
- 130825805252432 Millisekunden seit der Geburt

# Darstellung des Alters

- Im letzten Jahrtausend geboren
- 22 Jahre
- 24 Jahre, 7 Monate, 4 Tage, 12 Stunden, …..
- 130825805253432 Millisekunden seit der Geburt

# Darstellung des Alters

- Im letzten Jahrtausend geboren
- 22 Jahre
- 24 Jahre, 7 Monate, 4 Tage, 12 Stunden, …..
- 130825805254521 Millisekunden seit der Geburt

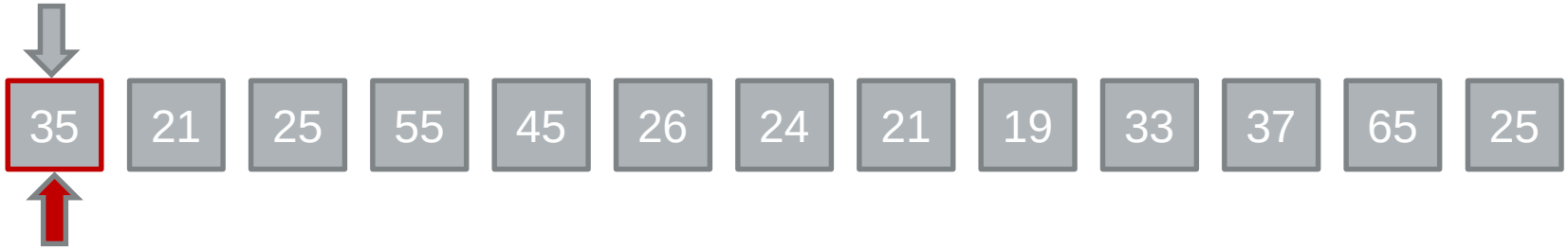
# Darstellung des Alters

- Im letzten Jahrtausend geboren
- 22 Jahre
- 24 Jahre, 7 Monate, 4 Tage, 12 Stunden, …..
- 130825805256413 Millisekunden seit der Geburt

# Linearer Algorithmus

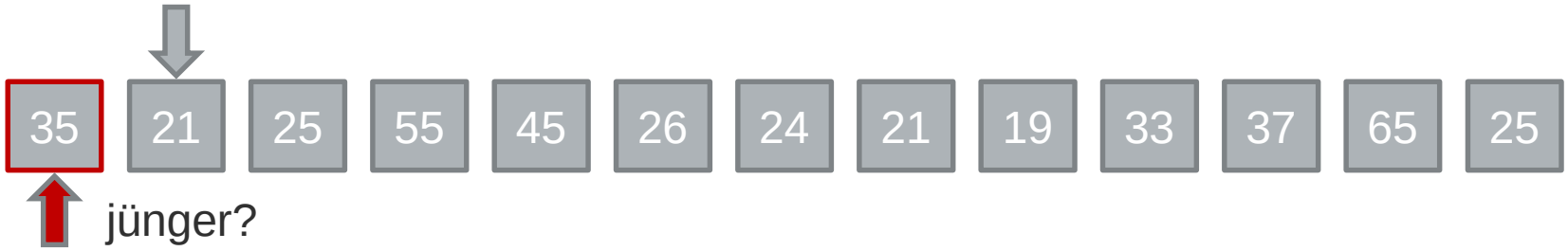
```
youngestAge = age[0];              Fange mit erster Person an
youngestPerson = 0;
for (int i = 0; i < N; i++) {      Frage jeden
                                          ob er/sie jünger ist
    if (age[i] < youngestAge) {           als die bisher
                                          jüngste Person

        youngestAge = age[i];             merke jüngste Person
        youngestPerson = i;               und Alter

    }
}
cout << i;                         Bekanngabe
```

# Initialisierung



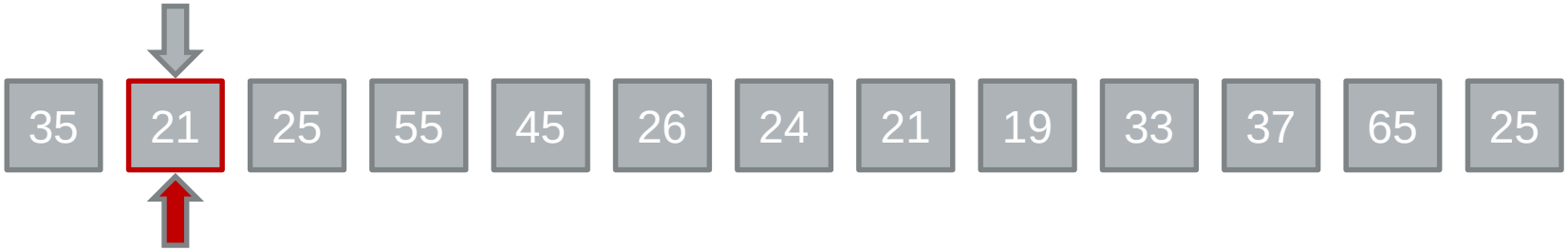

jüngste Person

# Linearer Algorithmus



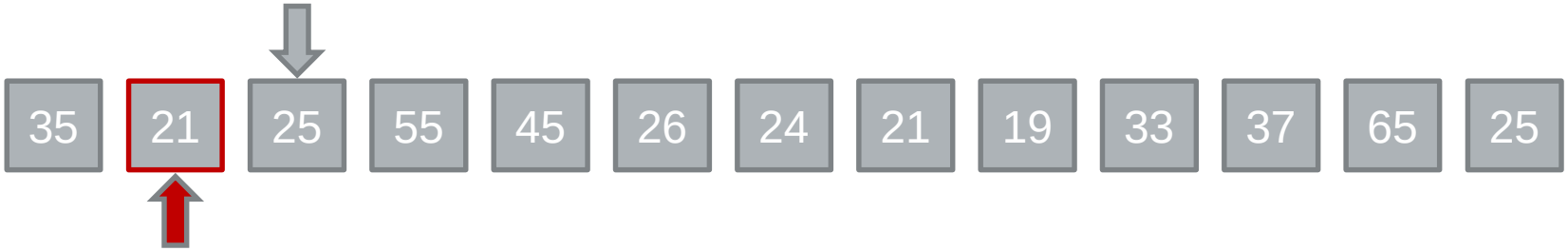

jüngste Person

# Linearer Algorithmus



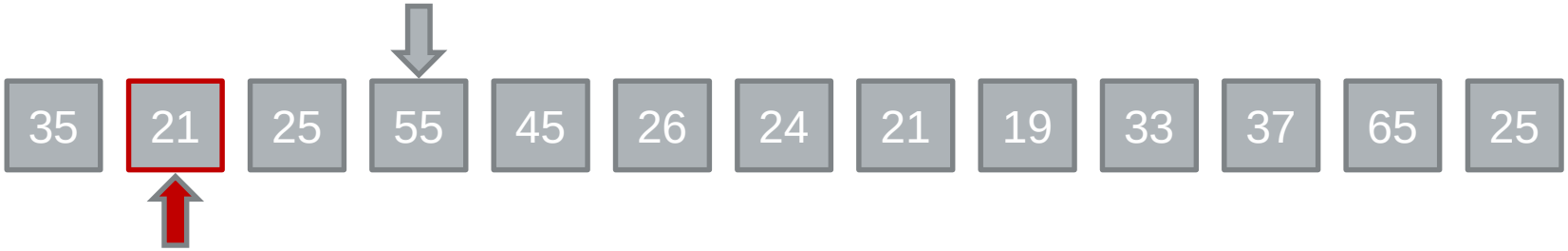

jüngste Person

# Linearer Algorithmus

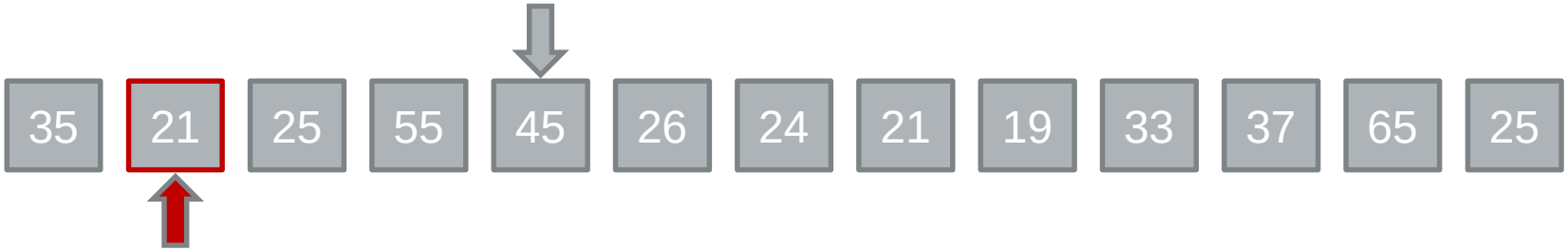



jüngste Person

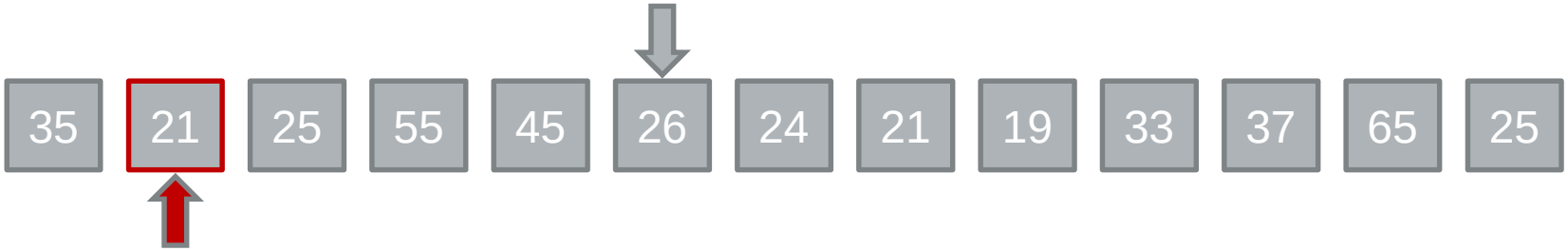
# Linearer Algorithmus



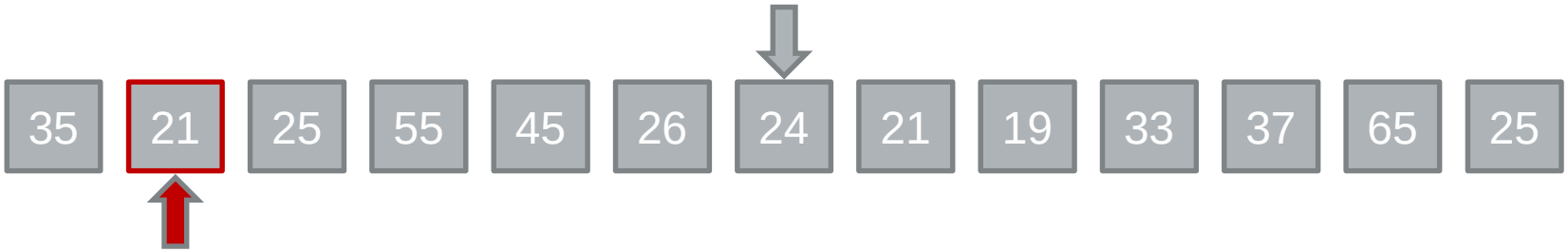
# Linearer Algorithmus



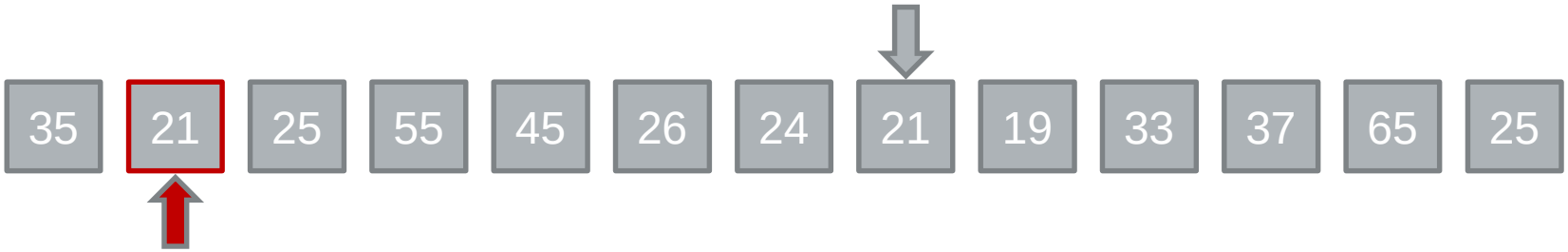
# Linearer Algorithmus



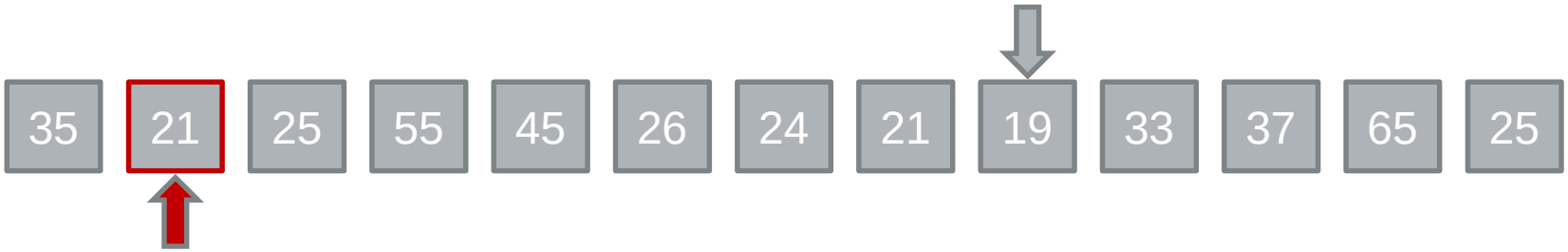
# Linearer Algorithmus



# Linearer Algorithmus



# Linearer Algorithmus



# Linearer Algorithmus

# Linearer Algorithmus

# Linearer Algorithmus

# Linearer Algorithmus

# Linearer Algorithmus

# Linearer Algorithmus


Beendet nach 13 Schritten !

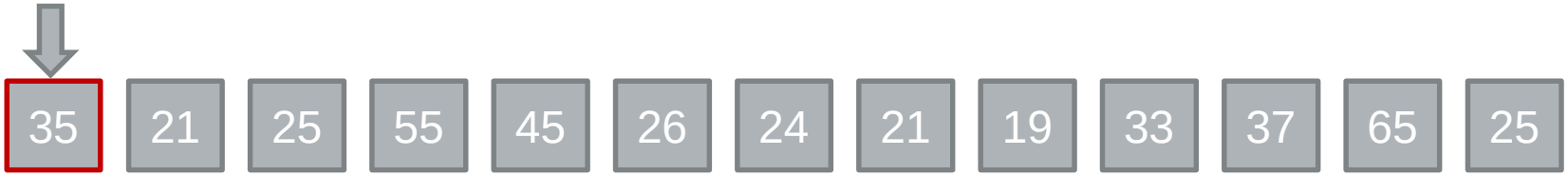
# Variante 2 – Teile und Herrsche

[computerworld.ch]

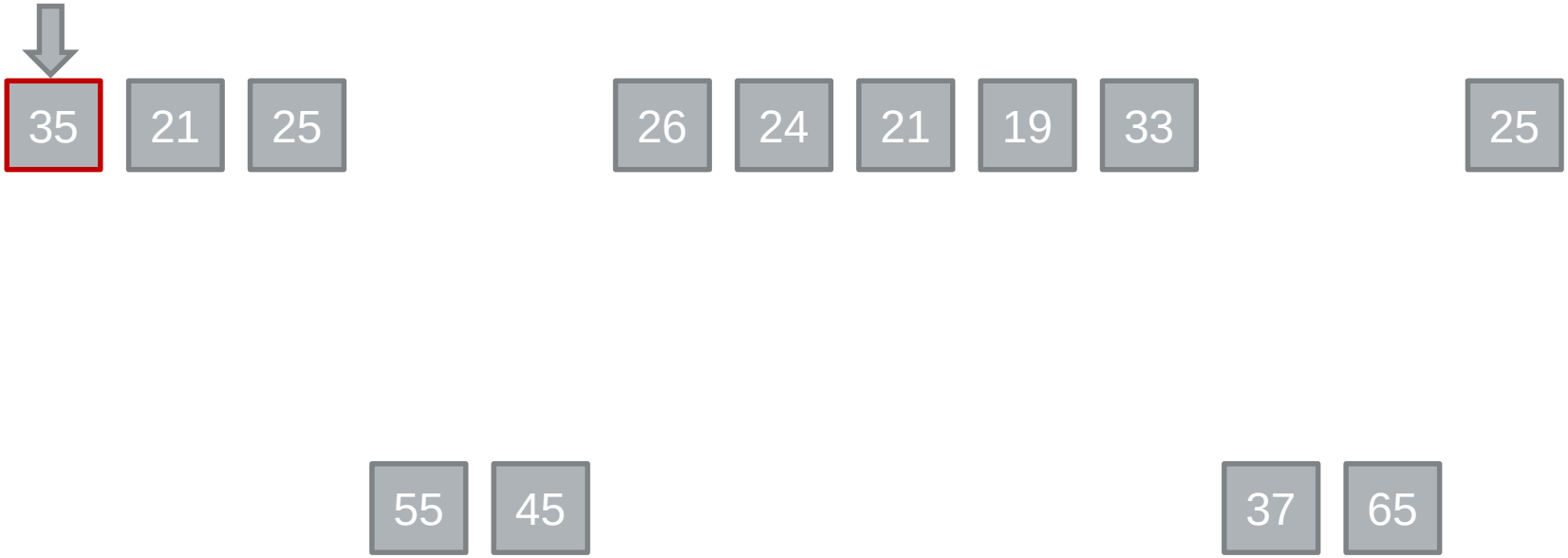
# Teile-und-Herrsche-Algorithmus

```
Fange mit erster Person an

Solange noch mehr als 2 Personen stehen

        Frage eine stehende Person nach dem Alter

        Alle, die älter sind, setzen sich


Bekanngabe
```

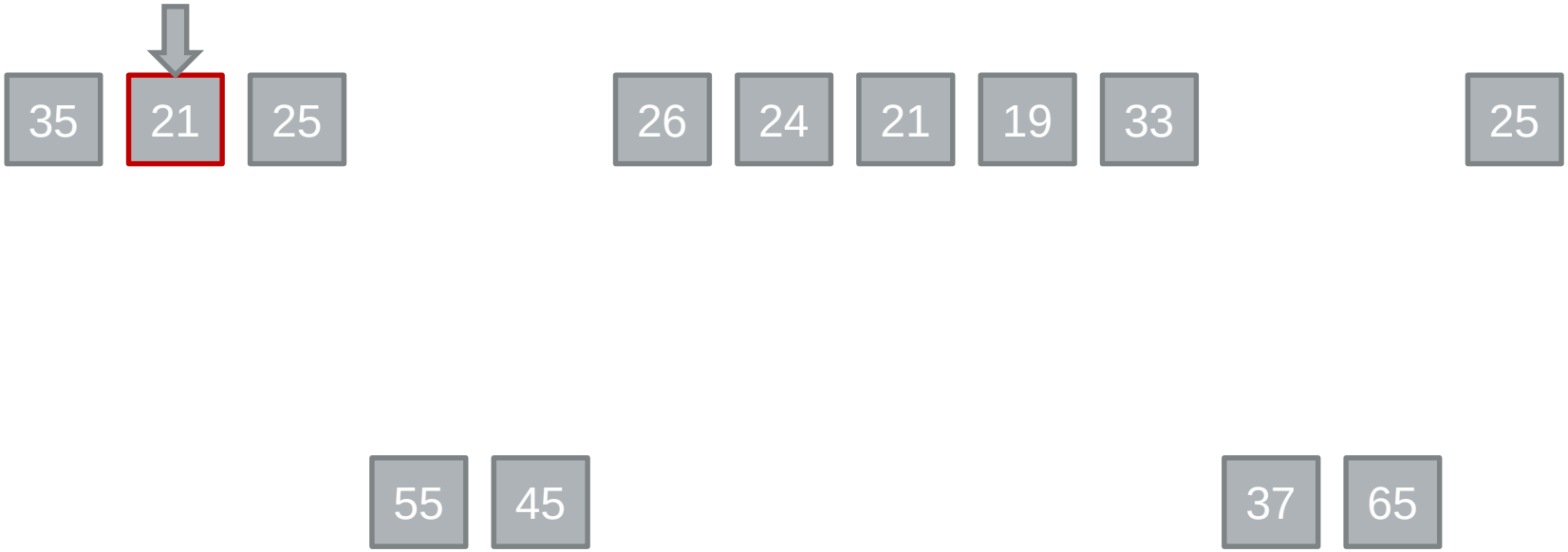
# Teile und Herrsche



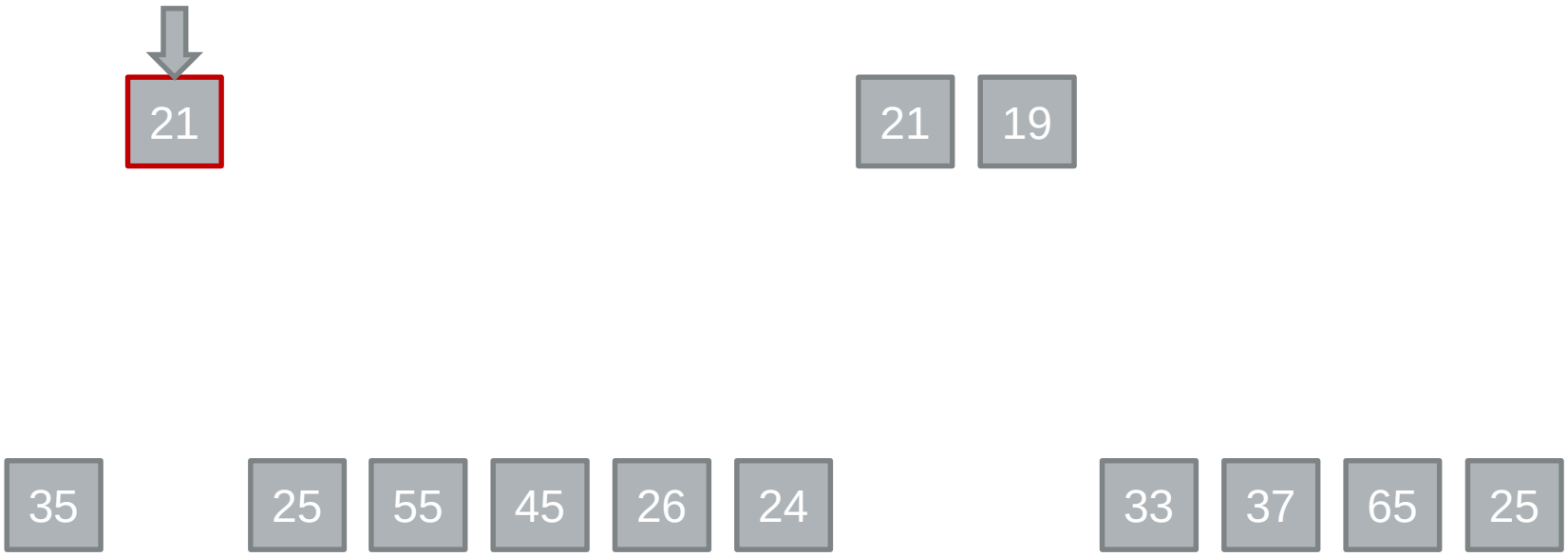
# Teile und Herrsche



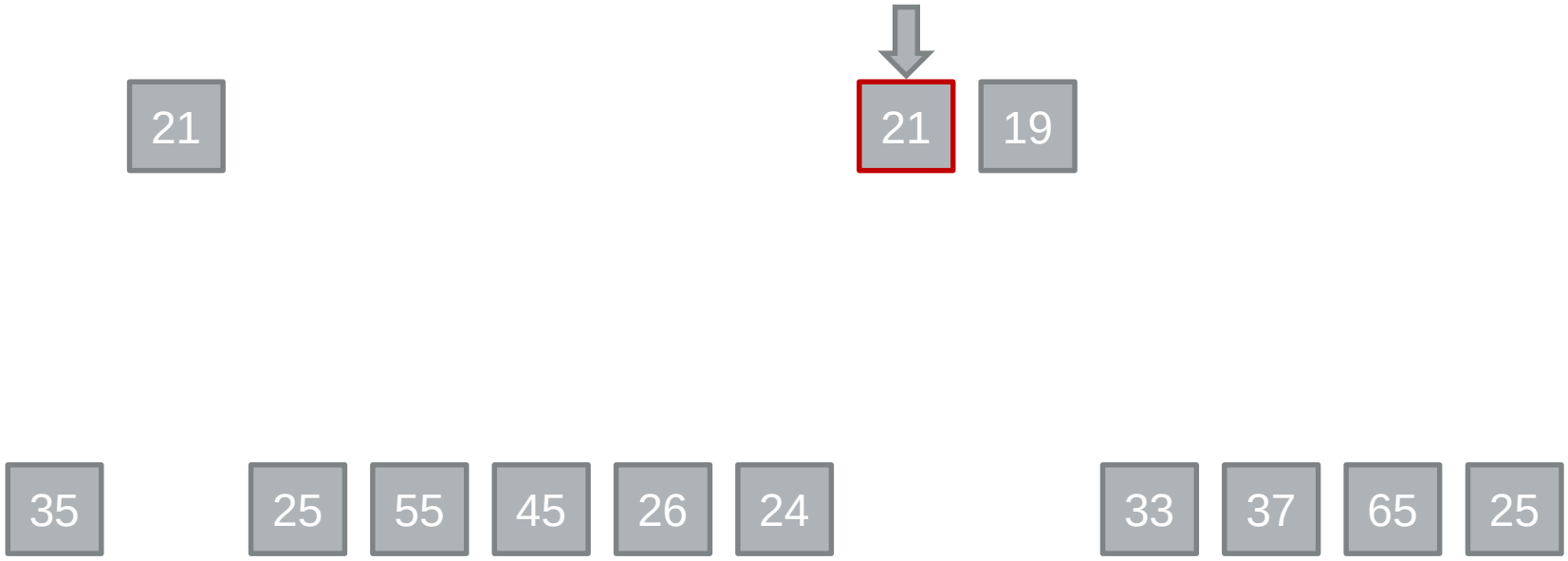
# Teile und Herrsche



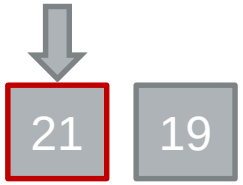
# Teile und Herrsche



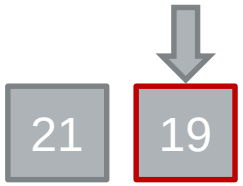
# Teile und Herrsche



# Teile und Herrsche




# Teile und Herrsche




# Teile und Herrsche


# Teile und Herrsche


jüngste Person


Beendet nach 4 Schritten !

# Vorgehen bei der Algorithmenentwicklung (idealisiert)

1. Problem formulieren
2. Problemanalyse, Problemspezifikation, Problemabstraktion
3. Algorithmenentwurf
4. Korrektheitsnachweis, Verifikation
5. Aufwandsanalyse
6. Programmkonstruktion

**[ Hinweis**: Wir betrachten diese Prinzipien im Folgenden anhand eines Beispiels. Hierbei wird bewusst in Kauf genommen, dass einiges an Stoff vorweg genommen wird (Darstellungsformen, Notationen, Methoden, …), um die Facetten der Informatik, von der Theorie bis zur Praxis, besser aufzeigen zu können! **]**

# Abstraktion – Problemformulierung und Analyse

1. Problemformulierung (Aufgabe):

   *„Finde die jüngste Person im Raum!“*

2. Problemanalyse:

   *Wie kann die Lösung aussehen? Gibt es überhaupt eine? Genau eine?*

   Problemspezifikation:

   | | |
   |---|---|
   | *Was ist „Alter“?* | Positive ganze Zahl |
   | *Was ist „jüngste“?* | Ordnungsrelation bei Zahlen |
   | *Was heißt „finde“?* | Altersvorgaben müssen gegeben sein |
   | *Was sind „Personen“?* | Altersangaben $a_1 \ldots a_n$, die mit Personen 1, …, n indiziert sind |

   Problemabstraktion:

   | | |
   |---|---|
   | Gegeben: | Eine Folge $a_1 \ldots a_n$ von positiven, ganzen Zahlen |
   | Gesucht: | Der kleinste Positionsindex j mit $a_j$ = minimum($a_1 \ldots a_n$)<br>(wenn mehrere kleinste Werte, dann kleinster Index) |

   (funktionale Spezifikation)

# Abstraktion – Algorithmenentwurf

3. Algorithmenentwurf:

   Ableitung eines Verfahrens aus der funktionalen Spezifikation.
   Oft gibt es verschiedene Lösungsmöglichkeiten.
   Erwünschte Lösung: elegant, kurz, verständlich, transparent, schnell, leicht veränderbar (Eigenschaften sind nicht immer miteinander vereinbar.)

   **Eine Lösung: Algorithmus MinAlter**:

1. [Wähle 1. Kandidat; n = Anzahl der Personen]
   Setze j := 1 und x := $a_j$  // Dies entspricht bereits der Lösung, falls n = 1
2. [Suchlauf]
   Setze i := 2
   Solange i $\leq$ n gilt, wiederhole:
   Falls $a_i$ < x, setze j := i und x := $a_j$;
   [jetzt gilt: $a_j$ = min($a_1$ … $a_i$)]
   Erhöhe i um 1
3. [Ausgabe]
   Person j mit Alter x ist eine jüngste Person

# Zahlenbeispiel

Gegeben: Folge von 8 Altersangaben: 20, 21, 20, 18, 19, 17, 18, 20

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | i | j | x |
|---|---|---|---|---|---|---|---|---|---|---|
| 20 | | | | | | | | 1 | 1 | 20 |
| 20 | 21 | | | | | | | 2 | 1 | 20 |
| 20 | 21 | 20 | | | | | | 3 | 1 | 20 |
| 20 | 21 | 20 | 18 | | | | | 4 | 4 | 18 |
| 20 | 21 | 20 | 18 | 19 | | | | 5 | 4 | 18 |
| 20 | 21 | 20 | 18 | 19 | 17 | | | 6 | 6 | 17 |
| 20 | 21 | 20 | 18 | 19 | 17 | 18 | | 7 | 6 | 17 |
| 20 | 21 | 20 | 18 | 19 | 17 | 18 | 20 | 8 | 6 | 17 |

Ergebnis: Person $j = 6$ mit Alter $x = a_j = 17$ ist eine jüngste Person!

# Analyse – Termination, Korrektheit von Algorithmen

## 4. Korrektheit und Verifikation:

***Terminiert der Algorithmus und liefert er das gewünschte Ergebnis bzgl. der Problemspezifikation?***

**Behauptung:** Der Algorithmus MinAlter terminiert und liefert das geforderte Ergebnis

**Beweis:** Vollständige Induktion über n

Induktions**behauptung**:
x = minimum($a_1$ … $a_n$) = $a_j$ (wobei j minimal ist und Algorithmus terminiert nach n-1 Durchläufen)

Induktions**anfang** (**n=1**): x = minimum($a_1$) = $a_j$ (j = 1 und Algorithmus terminiert nach 0 Durchläufen)
Beweis durch Simulation:
(1) j := 1; x := $a_j$;
(2) i := 2;
(3) j = 1 ⇒ Die Induktionsbehauptung gilt für n = 1

Induktions**schritt** (**n → n+1**):
Induktionsannahme: Induktionsbehauptung gelte für n; betrachte nun n‘ = n+1:
(1) j := 1; x := $a_j$;
(2) i := 2;
(3) Nach n‘-1 Durchläufen gilt nach Induktionsvoraussetzung: $a_j$ = minimum($a_1$ … $a_n$) = x (sowie i = n‘ = n+1)
i = n + 1 ≤ n + 1 ⇒ Wiederholung wird erneut (zum ((n‘-1) + 1). Mal) durchlaufen (letztmals!):
Falls $a_{n+1} \geq x$ ⇒ Induktionsbehauptung gilt für n‘ = n+1
Ansonsten: j und x erhalten neue Werte und die Induktionsbehauptung gilt für n‘
(Beachte, dass j minimal ist, da j und x nur dann neue Werte erhalten, wenn $a_{n+1} < x$)

# Analyse – Berechnungsaufwand

5. Aufwandsanalyse (Komplexitätsanalyse):

   Hierunter versteht man eine Analyse des Zeit- (und ggf. Speicher-) Bedarfs bei der Ausführung des Algorithmus.

   In diesem Fall gilt: Die Laufzeit ist proportional zur Anzahl der auszuführenden Elementar-operationen (**E** bedeute hier eine Zeiteinheit); wir analysieren die Anzahl der auszuführenden Operationen anhand einer Tabelle

| Anweisungen | Aufwand | Wie oft ausgeführt? |
|---|---|---|
| Setze j := 1; x := $a_j$ | 2 E | 1 |
| Setze i := 2 | 1 E | 1 |
| Teste $i \leq n$ | 1 E | n |
| Teste $a_i < x$ | 1 E | n – 1 |
| Setze j := i; x := $a_j$ | 2 E | max. (n – 1) mal |
| Erhöhe i um 1 | 1 E | n – 1 |

**Ergebnis**: Günstigster Fall: $T_{min}(n) = (3n + 1)$ E

Schlechtester Fall: $T_{max}(n) = (5n - 1)$ E

# Realisierung

6. Programmkonstruktion:
   - *Wie kommen die Daten in den Rechner?*
   - *Welche Programmiersprache steht zur Verfügung?*
   - *Wie wird das Ergebnis ausgegeben?*
   - *Wie wird das Programm strukturiert?*

**Ausblick**: Mit diesen Fragestellungen befassen wir uns in späteren Vorlesungen.

# EIGENSCHAFTEN VON ALGORITHMEN

# Programmbeispiel – Quadratwurzel

```
import java.util.Scanner;

class Quadratwurzel {
        public static void main(String[] args)
        { // gibt die Quadratwurzel aus
                double v;
                double vSqrt;
                Scanner scanner = new Scanner(System.in);

                // Eingabe der Zahl
                System.out.print("Zahl: ");
                v = scanner.nextDouble();

                // Berechne die Quadratwurzel
                vSqrt = (double) Math.sqrt( v );
                System.out.print("Quadratwurzel von "+
                        v+"= "+ vSqrt);

                scanner.close();
        } // Ende der main Methode
} // end of class
```

**Problem beim Umgang mit negativen Zahlen!**

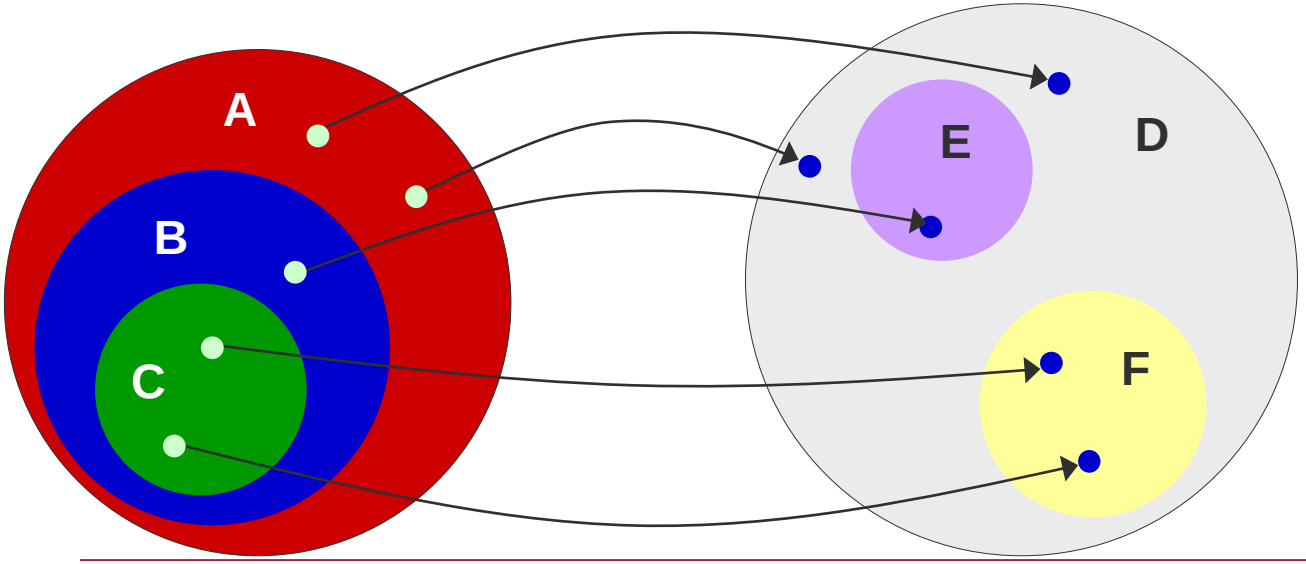
# Programmbeispiel – QuadratwurzelV2

```
import java.util.Scanner;

class QuadratwurzelV2 {
        public static void main(String[] args)
        { // gibt die Quadratwurzel aus
                double v;
                double vSqrt;
                Scanner scanner = new Scanner(System.in);

                // Eingabe der Zahl
                System.out.print("Zahl: ");
                v = scanner.nextDouble();

                if (v >= 0.0) {
                    // Berechne die Quadratwurzel
                    vSqrt = (double) Math.sqrt( v );
                    System.out.print("Quadratwurzel von "+
                        v + "= " + vSqrt);
                } // end of if

                scanner.close();
        } // Ende der main Methode
} // end of class
```

**Programmteil wird nur für positive Zahlen ausgeführt**

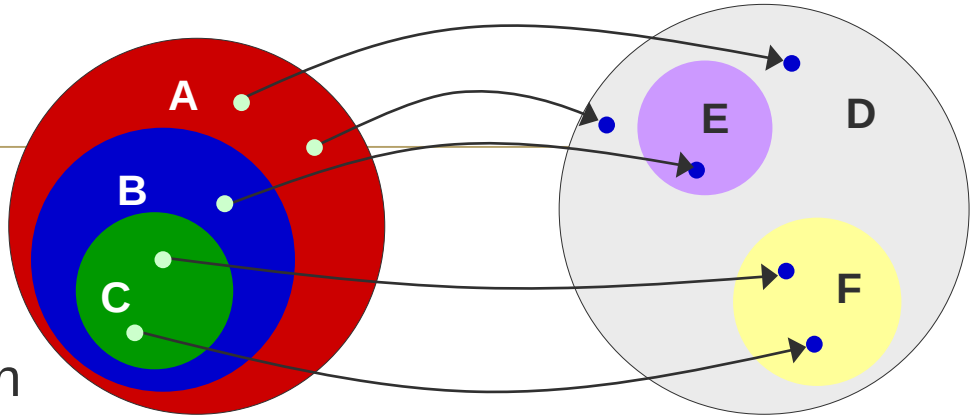
# Elementare Eigenschaften von Algorithmen

- **Korrektheit**: Ein Programm (Algorithmus) ist korrekt, wenn es für jede Eingabe aus dem Definitionsbereich der Funktion – deren Implementierung das Programm sein soll – die richtige Ausgabe erzeugt.




**A**: mögliche Eingaben
**B**: zulässige Eingaben (Definitionsbereich der Funktion)
**B-C**: als „Fehlerfall“ behandelte Eingaben
**D**: Ausgaben
**E**: Ergebnisse der „Fehlerbehandlung“
**F**: „normale“ Ausgaben

# Korrektheit



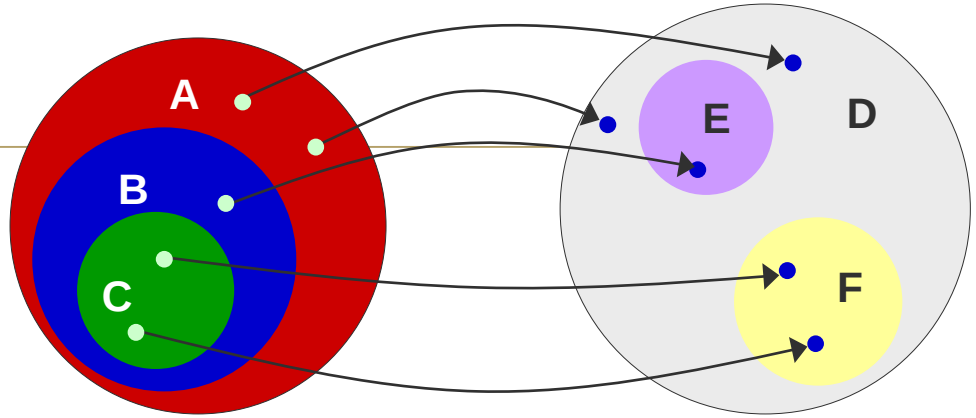

- Ein Programm ist **korrekt**, wenn
  B – C → E („Fehlerfall“) und
  C →F („Normalfall“)
  wie definiert implementiert sind.

**A**: mögliche Eingaben
**B**: zulässige Eingaben (Definitionsbereich)
**B-C**: als „Fehlerfall“ behandelte Eingaben
**D**: Ausgaben
**E**: Ergebnisse der „Fehlerbehandlung“
**F**: „normale“ Ausgaben

Anmerkungen:

- Ein Programm reagiert irgendwie auf jede denkbare Eingabe (die nicht im Definitionsbereich der Funktion liegen).
  - Dies ist die Abbildung A – B → D – (E ∪ F).
  - Dieser Bereich könnte auch leer sein.
- Für die Korrektheitsbetrachtung ist es unwichtig, ob Teile der definierten Eingaben (B – C) als „falsch“ angesehen werden

# Robustheit




**A**: mögliche Eingaben
**B**: zulässige Eingaben (Definitionsbereich)
**B-C**: als „Fehlerfall“ behandelte Eingaben
**D**: Ausgaben
**E**: Ergebnisse der „Fehlerbehandlung“
**F**: „normale“ Ausgaben

- Die **Robustheit** ist eine Aussage darüber, wie viele (vom problemlösenden Algorithmus aus gesehen) falsche Eingaben vom Programm aus **erkannt und gemeldet** werden (|B – C| ≡ card(B – C)).

- „Extremwerte“:
  - Nicht robust ist ein Programm, in dem eine Behandlung von Eingaben, die eine Fehlermeldung als Ausgabe verursachen müssen, nicht vorgesehen ist (B = C und A – B nicht leer).
  - Ein Programm ist maximal robust, wenn es keine Eingabe gibt, die das Programm zu Fehlreaktionen veranlassen kann (A = B; Idealfall).

# Termination

- Ein Algorithmus heißt **terminierend**, wenn er – bei jeder erlaubten Eingabe – nach endlich vielen (elementaren) Schritten zu seinem Ende kommt.
- Gegenbeispiel:

```
import java.util.Scanner;

class BisZehnV1 {  // gibt die Zahlen von n bis 10 aus
        public static void main(String[] args) {
                int n;
                // Eingabe der Zahl
                Scanner scanner = new Scanner(System.in);
                System.out.print("Zahl: ");
                n = scanner.nextInt();
                // wiederhole bis die Bedingung eingetreten ist
                while (n != 11) {
                        System.out.println(n);
                        n = n+1;
                } // end of while
                scanner.close();
        } // end of main
} // end of class
```

**Problem für n > 11 !**

# Determinismus und Determiniertheit

- Ein **deterministischer Algorithmus** ist ein Algorithmus, der eine eindeutige Vorgabe für die Folge der auszuführenden Schritte festlegt.
  - Dadurch folgt auf eine bestimmte Anweisung innerhalb des Algorithmus unter den gleichen Voraussetzungen stets die gleiche Anweisung.
  - Ein **nicht-deterministischer Algorithmus** besitzt keinen eindeutig vorgeschriebenen Ablauf.
- Ein Programm bildet (im Sinne einer Funktion) eine Menge von Eingabewerten auf eine Menge von Ausgabewerten ab, deren Umfang nicht gleich mächtig sein muss (z.B. andere Eingabe kann gleiche Ausgabe erzeugen).
- Ein **determinierter** Algorithmus ist gegeben, wenn
  - diese Abbildung eine mathematische Funktion definiert und somit
  - jeder Eingabewert auf **genau einen** Ausgabewert abgebildet wird.

# Determinismus (des Rechenwegs) und Determiniertheit (des Resultats)

## Ein Alltagsalgorithmus: Wegbeschreibung

- **Deterministisch** und **determiniert**
  Die Wegbeschreibung ist eindeutig; alle Nutzer fahren den selben Weg und erreichen somit das selbe Ziel.
- **Nicht-deterministisch** und **determiniert**
  Nur das Ziel ist eindeutig; Nutzer fahren verschiedene Wege, erreichen aber das selbe Ziel.
- **Nicht-deterministisch** und **nicht-determiniert**
  „Herumirren“, „Fahrt in‘s Blaue“; die Nutzer fahren verschiedene Wege und erreichen verschiedene Ziele (… Exploration eines Gebiets)
- **Deterministisch** und **nicht-determiniert**
  Gibt es eigentlich nicht, denn falls die Wegbeschreibung eindeutig ist, dann auch das Ziel!
  [ Allerdings könnte zum Beispiel durch Interaktionen mit der Umwelt oder einer randomisierten Simulationsumgebung durchaus auch diese Kombination möglich sein – in diesem Fall wäre die Umwelt bzw. die Simulation nicht-deterministisch. ]

## Beispiel: Nicht-deterministischer aber determinierter Algorithmus

1. *Nehmen Sie eine beliebige Zahl x ungleich 0.*
2. ***Entweder**: Addieren Sie das Dreifache von x zu x hinzu und teilen Sie das Ergebnis durch x*
   *(3 x + x) / x.*

   ***Oder**: Subtrahieren Sie die Zahl 4 von x und subtrahieren Sie das Ergebnis von x*
   *x – (x – 4).*
3. *Schreiben Sie das Ergebnis auf.*

- Warum nicht-deterministisch? Der Rechenweg (2.) ist nicht festgelegt, sondern ist durch eine zufällige Auswahl spezifiziert.
- Warum determiniert? Dieselbe Eingabe für x liefert immer das selbe Resultat, nämlich: **4**

## Beispiel: Nicht-determinierter (nicht-deterministischer) Algorithmus

1. *Nehmen Sie eine beliebige Zahl x ungleich 0.*
2. *Entweder: Addieren Sie das Dreifache von x zu x hinzu und teilen das Ergebnis durch x*

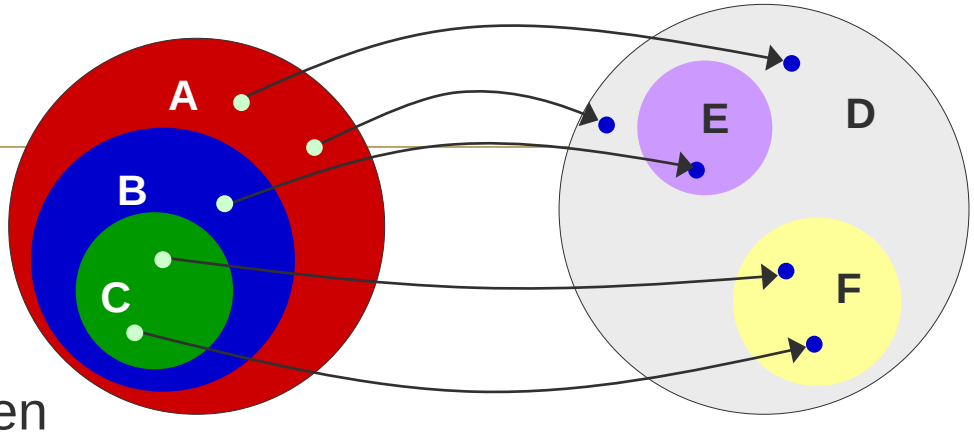
   $(3x + x) / x.$

   *Oder: Subtrahieren Sie die Zahl 5 von x und subtrahieren das Ergebnis von x*

   $x – (x – 5).$
3. *Schreiben Sie das Ergebnis auf.*

- Was macht die Nicht-Determiniertheit aus?
  - Je nachdem, welche Anweisung in Schritt 2 ausgewählt wird, ist das Ergebnis entweder 4 oder 5.
  - Damit bildet der Algorithmus bei gleicher Eingabe x auf Ausgabewerte 4 oder 5 ab.

# Weitere Betrachtungen




- **Vor- und Nachbedingungen**
  - Unter welchen Bedingungen arbeitet der Algorithmus?
    - Menge der erlaubten Eingaben (|B|)
    - Menge aller möglichen Ausgaben bei erlaubter Eingabe (E ∪ F)
  - Was geschieht bei falscher Eingabe?
- **Termination**
  - Endet der Algorithmus für alle möglichen Eingaben?
  - Ist es möglich, den Algorithmus in einen nicht-endenden Zyklus zu bringen?
  - Ein vollständiger **formaler Beweis** kann – theoretisch – durchgeführt werden, wenn die Funktion mathematisch definiert ist;
  - Ist sie nicht formal beschrieben bzw. beschreibbar, so müssen **Plausibilitätsbetrachtungen und Tests** angestellt werden.

# Weitere Betrachtungen: Aufwand / Effizienz

- Ausführungszeit:
  - Wie lange dauert das Programm?
- Speicherplatzbedarf:
  - Wie viel Speicher (RAM etc.) benötigt das Programm?

- Ist die Ausführungszeit / der Speicherplatzbedarfs abhängig von der Eingabe? – in wie weit?

- Kann die verfügbare Hardware noch besser ausgenutzt werden?
  - max. Performanz (MFLOPs)
  - Parallelisierung
    - Ausnutzung von mehreren Recheneinheiten – multi-cores
    - Ausnutzung von Graphics Processing Units (GPUs – also Grafikkarten)
  - Caching
    - Optimale Zwischenspeicherung von Daten abhängig davon wie häufig sie gebraucht werden und wie sie sequentiell bearbeitet werden.
  - …

# DARSTELLUNG VON ALGORITHMEN

# Darstellung von (einfachen) Algorithmen

**Was muss dargestellt werden?**

**Wichtigste Elemente:**

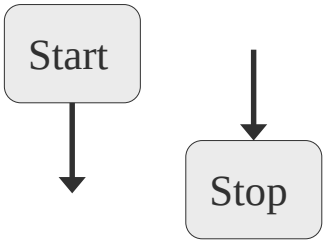
- **Ausführungsschritte, Algorithmusbausteine**
  - Anweisungen – Tätigkeiten, Handlungen
  - Bedingungen
  - Einfache und zusammengesetzte Anweisungen

- **Ausführungsformen** und **–reihenfolgen**
  - Sequentiell
  - Alternativ (d.h. Auswahl und Entscheidung)
  - Iterativ (d.h. wiederholt)

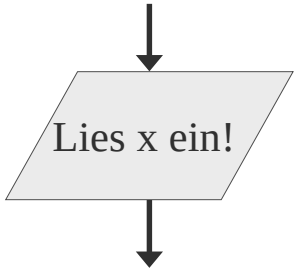
- **Zusammensetzung** von Ausführungsschritten und -formen

# Grafische Darstellung – Flussdiagramme

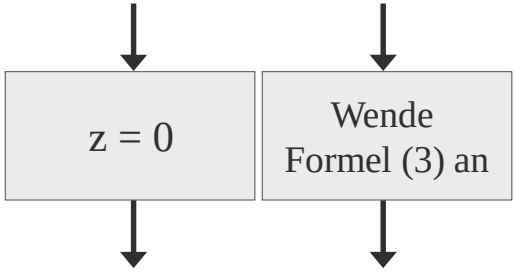
**Flussdiagramme** sind eine Methode zur informellen Beschreibung von Algorithmen



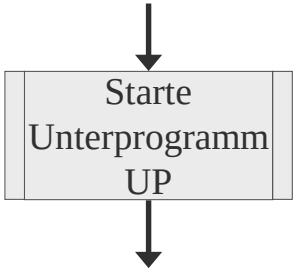

Anfang und Ende eines Programms



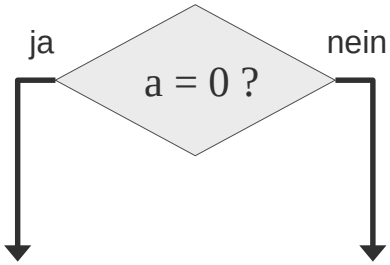

Ein- / Ausgabe

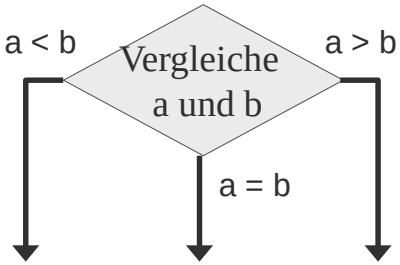



Elementare Anweisungen

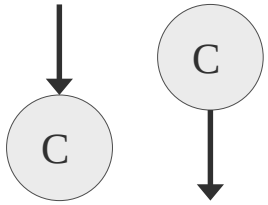



Unterprogramm
Teil- / Subalgorithmus
(UP: Name)




Verzweigung aufgrund einer Bedingung




Konnektor zur Verbindung von Diagrammen

**Hinweis**: Flussdiagramme und weitere grafische Darstellungsformen für Algorithmen werden später noch genauer vorgestellt und angewendet

# Größter gemeinsamer Teiler

- Bestimmung des größten gemeinsamen Teilers ggT(a, b)
- Definition eines Algorithmus
- Aufgabe: Bestimmung des Wertes für ggT(a, b) für zwei natürliche Zahlen a und b
- Algorithmus 1 beruht auf folgender Behauptung:

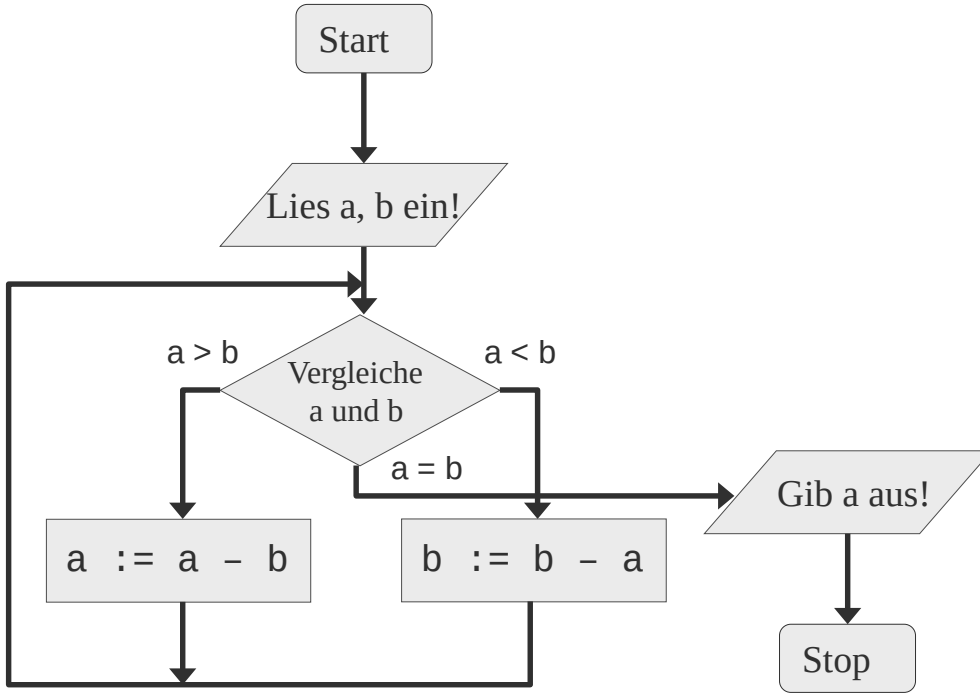
$$ggT(a, b) = \begin{cases} a & \text{falls } a = b \\ ggT(a - b, b) & \text{falls } a > b \\ ggT(a, b - a) & \text{falls } a < b \end{cases}$$

(*)

Bsp.: ggT(20, 15) → 20 > 15 : ggT((20 – 15) = 5, 15)
5 < 15 : ggT(5, (15 – 5) = 10)
5 < 10 : ggT(5, (10 – 5) = 5)
5 = 5 : 5

# Flussdiagramm für ggT Algorithmus

$$ggT(a, b) = \begin{cases} a & \text{falls } a = b \\ ggT(a - b, b) & \text{falls } a > b \\ ggT(a, b - a) & \text{falls } a < b \end{cases}$$




## Partielle vs. totale Korrektheit

- **Partielle Korrektheit**: Falls der Algorithmus terminiert, sind die Ergebnisse korrekt; sie entsprechen der formalen Spezifikation des Algorithmus ( hier: Resultat = ggT(a, b) )
- **Totale Korrektheit** = Partielle Korrektheit + Termination

# Beweis der Korrektheit des ggT Algorithmus (*)

$$ggT(a, b) = \begin{cases} a & \text{falls } a = b \quad (1) \\ ggT(a - b, b) & \text{falls } a > b \quad (2) \\ ggT(a, b - a) & \text{falls } a < b \quad (3) \end{cases}$$

Beweis:

1. Gilt trivialerweise (!)
2. Ist zu beweisen
3. Geht dann durch Vertauschen von a und b aus (2) hervor

Es genügt somit zu zeigen:

$ggT(a, b) = ggT(a - b, b)$ falls $a > b$ (2)

Zu zeigen:

$$ggT(a, b) = ggT(a - b, b) \quad \text{falls} \quad a > b \qquad (2)$$

- Beweis von (2) (sei a > b): [g.T.: gemeinsamer Teiler]
- Wir zeigen: { z | z ist g.T. von a und b } = { v | v ist g.T. von a – b und b }
  (denn dann sind auch die Maxima dieser beiden Mengen gleich)

" $\subseteq$ ": Sei z in der linken Menge $\Rightarrow$ $z / a$ und $z / b$ (*) ( / bedeutet "teilt")
Zu zeigen: $z / (a - b)$

Wegen (*) gilt: $a = n \bullet z$ für ein $n \in \mathbb{N}$ und
$b = m \bullet z$ für ein $m \in \mathbb{N}$
$\Rightarrow$ $a - b = n \bullet z - m \bullet z = (n - m) \bullet z$
$\Rightarrow$ $z / (a - b)$

" $\supseteq$ ": Sei v in der rechten Menge $\Rightarrow$ $v / (a - b)$ und $v / b$ (**)
Zu zeigen: $v / a$

Wegen (**) gilt: $a - b = k \bullet v$ für ein $k \in \mathbb{N}$ und
$b = l \bullet v$ für ein $l \in \mathbb{N}$
$\Rightarrow$ $a = (a - b) + b = k \bullet v + l \bullet v = (k + l) \bullet v$
$\Rightarrow$ $v / a$

# Partielle Korrektheit von ggT Algorithmus

## Beweis der Korrektheit des Resultats

- Die partielle Korrektheit von Algorithmus 1 ergibt sich unmittelbar aus der Behauptung (*):

  (*) besagt, dass der ggT von a und b sich von Schleifendurchlauf zu Schleifendurchlauf nicht ändert, auch wenn sich a und b selbst ändern

- Falls sich also irgendwann einmal die Situation a = b ergibt, terminiert der Algorithmus – in diesem Fall gilt dann:

  $$ggT(a, b) = a = b = ggT(a_0, b_0)$$

  wobei $a_0$ und $b_0$ die ursprünglich eingegebenen Werte seien

- Die **partielle Korrektheit** des ggT Algorithmus steht somit fest
- **ABER**: Es ist noch **unbewiesen**, dass Algorithmus 1 wirklich **terminiert.**

# Totale Korrektheit des ggT Algorithmus

**Beweis der Termination**

- Es bleibt noch zu zeigen, dass Algorithmus 1 stets terminiert:

  Setze s = a + b

- Es ist klar, dass sich in jedem Durchlauf der Wert von s um mindestens 1 verringert.
- Da stets a, b $\geq$ 1 gilt, ist immer s $\geq$ 2 gegeben.
- Da der Wert von s in jedem Durchlauf abnimmt, aber stets s $\geq$ 2 gilt, muss Algorithmus 1 terminieren, und zwar spätestens nach (a + b) – 2 Schritten.
- **Ergebnis**:
  - Damit ist die **partielle Korrektheit** von Algorithmus 1 sowie
  - dessen **Termination** gezeigt;
  - ➢ … und damit schließlich auch dessen **totale Korrektheit.**
- **Wichtige Voraussetzungen**:
  - natürliche Zahlen,
  - nur positive Zahlen!

## Größter gemeinsamer Teiler ggTE(a, b) – Euklid‘scher Algorithmus

Aufgabe: Bestimmung des Wertes für ggT(a, b) für zwei natürliche Zahlen a und b in einem alternativen Ansatz nach Euklid

$$ggTE(a, b) = \begin{cases} b & \text{falls } a \text{ MOD } b = 0 \\ ggTE(b, a \text{ MOD } b) & \text{sonst} \end{cases}$$

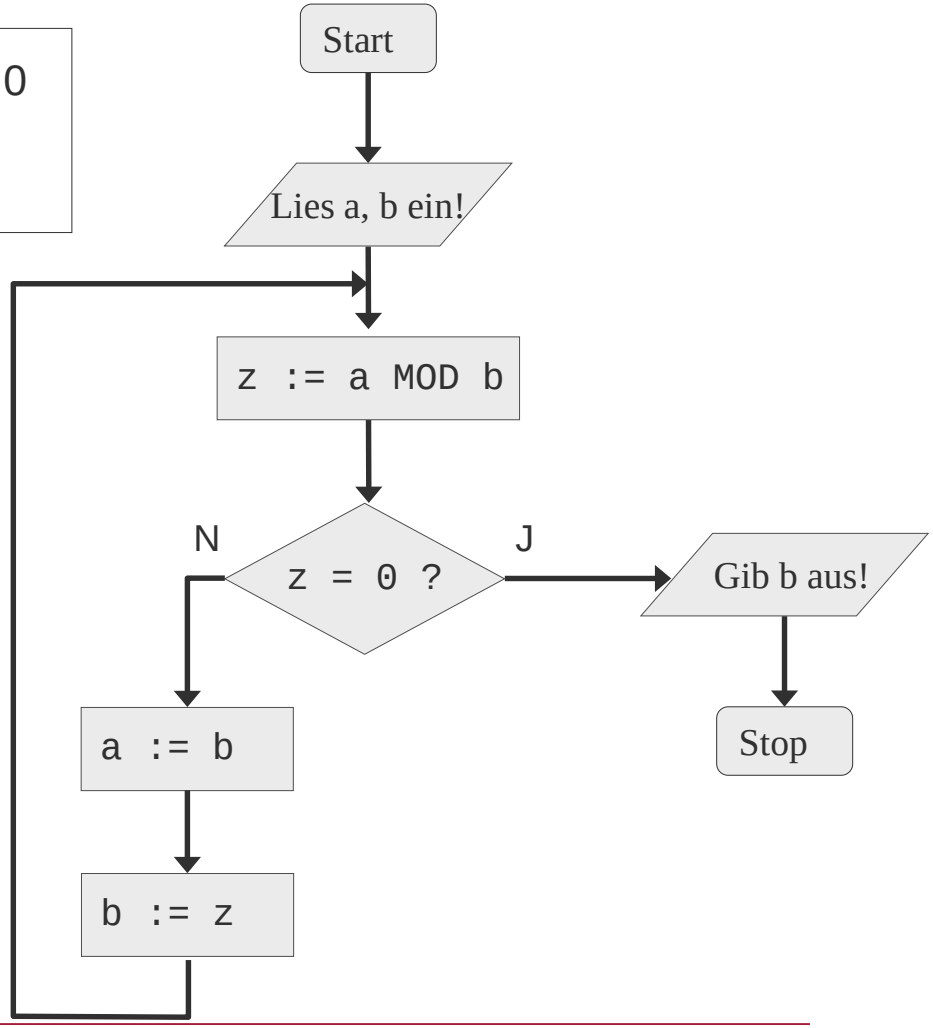
**Erläuterung**: Die MOD-Operation definiert den Rest einer ganzzahligen Division; Bsp.: 13 MOD 6 = 1 (entsprechend 2 • 6 + 1)

Bsp.: ggTE(20, 15)→ 20 MOD 15 ≠ 0: ggT(15, (20 MOD 15) = 5)
15 MOD 5 = 0: 5

Beobachtung: Algorithmus 1 hat bei gleicher Eingabe für die Bestimmung 4 Schritte benötigt, Algorithmus 2 benötigt nur 2! *Gilt das für alle Eingaben?*

# Flussdiagramm für Algorithmus ggTE

$$ggTE(a, b) = \begin{cases} b & \text{falls } a \text{ MOD } b = 0 \\ ggT(b, a \text{ MOD } b) & \text{sonst} \end{cases}$$




**Erläuterungen:**

- **Auswahl**: Welcher Pfad wird ausgeführt (zwei Möglichkeiten; bei einer liegt das Ergebnis vor)
- **Wiederholungen**: Vorschrift wird (mit verschiedenen Werten für a und b) mehrmals durchlaufen; es wird eine **Variable** z zur Speicherung von Zwischenwerten verwendet

# Beweis der Korrektheit von Algorithmus ggTE

" $\subseteq$ ": Sei v in der linken Menge $\Rightarrow$ $v / a$ und $v / b$ **(*)**
Zu zeigen: $v / (a \text{ MOD } b)$

Wegen **(*)** gilt: $a = m \bullet v$ für ein $m \in \mathbb{N}$ und
$b = n \bullet v$ für ein $n \in \mathbb{N}$
Sei r = a MOD b, d.h. $a = k \bullet b + r$ für ein $k \in \mathbb{N}$ und $r < b$

Einsetzen ergibt: $a = m \bullet v = k \bullet b + r = k \bullet n \bullet v + r$
$\Rightarrow$ $r = m \bullet v - k \bullet n \bullet v = (m - k \bullet n) \bullet v$
$\Rightarrow$ $v / r$ (d.h. v ist durch r teilbar)

" $\supseteq$ ": Sei z in der rechten Menge $\Rightarrow$ $z / (a \text{ MOD } b)$ und $z / b$ **(**)**
Zu zeigen: $z / a$

Wegen **(**)** gilt: $a \text{ MOD } b = m \bullet z$ für ein $m \in \mathbb{N}$ und
$b = n \bullet z$ für ein $n \in \mathbb{N}$,
wobei $a = k \bullet b + (a \text{ MOD } b)$ für ein $k \in \mathbb{N}$

Einsetzen ergibt: $a = k \bullet b + m \bullet z = k \bullet n \bullet z + m \bullet z = (k \bullet n + m) \bullet z$
$\Rightarrow$ $z / a$

# Totale Korrektheit von Algorithmus ggTE

## Beweis der Termination

- Es bezeichnen $a_i$, $b_i$ die Werte von a, b nach dem i-ten Aufruf
  Wir zeigen zunächst: $b_i > 2 \bullet b_{i+2}$

  | | | | |
  |---|---|---|---|
  | Stufe i : | $a_i = k \bullet b_i + b_{i+1}$ | $b_{i+1} < b_i$ |
  | Stufe i+1 : | $a_{i+1} = l \bullet b_{i+1} + b_{i+2}$ | $b_{i+2} < b_{i+1}$ |
  | Stufe i+2 : | … | |

- Ferner gilt: $a_{i+1} = b_i$ ; eingesetzt ergibt dies:
  $b_i = a_{i+1} = l \bullet b_{i+1} + b_{i+2}$

- Falls: $l = 0$ wäre, würde folgen:
  $b_i = a_{i+1} = b_{i+2} \rightarrow$ Widerspruch zu den obigen Ungleichungen
  Daher muss $l \geq 1$ gelten
  Daraus folgt: $b_i = a_{i+1} = l \bullet b_{i+1} + b_{i+2} \geq b_{i+1} + b_{i+2} > 2 \bullet b_{i+2}$

- Seien $b_0, b_1, \ldots, b_n$ die Zahlenwerte, welche die Variable b durchläuft
  Mit obige Abschätzung ergibt sich (n sei o.B.d.A. als gerade angenommen):
  $b_0 > 2 \bullet b_2 > 4 \bullet b_4 > 8 \bullet b_6 > 16 \bullet b_8 > \ldots > 2^{n/2} \bullet b_n \geq 2^{n/2}$ (da $b_n \geq 1$)
  Logarithmieren ergibt:
  $\log_2 b_0 > n / 2 \Rightarrow n < 2 \bullet \log_2 b_0$ (ähnliche Abschätzung für n ungerade)

- Es folgt nicht nur die Termination von **Algorithmus ggTE**, sondern man erhält eine gute Abschätzung für die Zahl n (Anzahl der Schleifendurchläufe). Diese Abschätzung ergibt außerdem, dass im Unterschied zu Algorithmus ggT für Algorithmus ggTE sogar (sehr) große Eingaben a, b effizient handhabbar sind.

# Bemerkungen zur Analyse der totalen Korrektheit

- Im Allgemeinen ist **unentscheidbar**, für einen gegebenen Algorithmus festzustellen, ob er terminiert.
  - Diese Thematik wird Gegenstand der Theoretischen Informatik sein.
- Eine derartige Analyse (wie in den vorangehenden Beispielen zur Bestimmung des ggT) gelingt nur in Einzelfällen und kann nicht „mechanisch“ ausgeführt werden.
  - Hierzu werden später in der Vorlesung Ansätze zur Korrektheitsanalyse betrachtet.

# Implementierung in Java

- Der Vollständigkeit halber sind im Folgenden zwei Java-Programme als Implementierung für **Algorithmus ggT** bzw. **Algorithmus ggTE** angegeben.

- Die genaue Syntax und Funktionsweise dieser Programme werden wir später behandeln.
  - Die Details der Programme und ihrer Konstrukte müssen an dieser Stelle noch nicht umfassend verstanden werden …

# Implementierung der ggT-Algorithmen in Java – Algorithmus 1

```
import java.util.Scanner;

public class ggT {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);

      System.out.print("Gib erste Zahl ein:  ");
      int a = sc.nextInt();
      System.out.print("Gib zweite Zahl ein: ");
      int b = sc.nextInt();

      while (a != b) {
         if (a > b)
            a = a - b;
         else
            b = b - a;
      }

      System.out.println(“Der ggT ist: “ + a);
   } // end main
} // end class ggT
```

Eingabe von zwei Zahlen a, b

# Implementierung der ggT-Algorithmen in Java – Algorithmus 2

```
import java.util.Scanner;

public class ggTE {
   public static void main(String[] args) {
      int z;
      Scanner sc = new Scanner(System.in);

      System.out.print("Gib erste Zahl ein:  ");
      int a = sc.nextInt();
      System.out.print("Gib zweite Zahl ein: ");
      int b = sc.nextInt();

      z = a % b;  // Rest nach Division von a und b
      while (z != 0) {
         a = b;
         b = z;
         z = a % b;
      }
      System.out.println("Der ggT ist: " + b);
   } // end main
} // end class ggTE
```

Eingabe von zwei Zahlen a, b

# Zusammenfassung

- Ein **Algorithmus** ist eine präzise endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer, elementarer Verarbeitungsschritte mit einer endlichen Gesamtausführungsdauer.

- Elementare Eigenschaften
  - Ausführbarkeit
  - Teilalgorithmen und elementare Verarbeitungsschritte
  - Ausführungsreihenfolge
  - Endlichkeit der Beschreibung
  - Termination
  - Determinismus und Determiniertheit
  - Korrektheit und totale Korrektheit

# Ausblick

- Datentypen
- Programmstruktur