



Algorithmen und Datenstrukturen

Kapitel 01: Algorithmen – Grundlagen

Prof. Dr. Adrian Ulges

B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain



Outline

1. Der Algorithmus-Begriff
2. Eigenschaften von Algorithmen
3. Beispiel: Primzahltest
4. Darstellung von Algorithmen
 - Pseudo-Code
 - Flussdiagramme
 - Struktogramme

Feindbild Algorithmus

Der Wissenschaftliche Dienst des Bundestages erkennt "Bedarf" an einer Regulierung der Algorithmen. Aber schon die seltsame Fixierung auf den Begriff ist irreführend.



Automatisierung

Hälften der EU-Bürger weiß nicht, was ein Algorithmus ist

Im Internet führt an ihnen zwar kaum ein Weg vorbei – dennoch wissen die Menschen wenig über Algorithmen. Die meisten Europäer verlangen nach einer wirksameren Kontrolle. ★ 44

Deutschland braucht mehr Algorithmenkompetenz

Künstliche Intelligenz verändert unsere Gesellschaft gewaltig. Leider verstehen die meisten Menschen nicht, wie Algorithmen funktionieren. Das muss sich dringend ändern!

Auswahl von Beiträgen

Facebook will Algorithmus erklären

Warum wird mir dieser Beitrag angezeigt? Diese Frage stellen sich viele Facebook-Nutzer. Der Konzern will die Auswahl der Beiträge jetzt besser erklären – auch die der Werbung. [Mehr](#)

„Die Muße ist ein Algorithmus“

Was passiert, wenn Algorithmen Bilder malen, neue Formen generieren und Skulpturen schaffen. Ist das wirklich Kunst oder nur eine Kopie? [Mehr](#)

Definieren Sie “Informatik”!

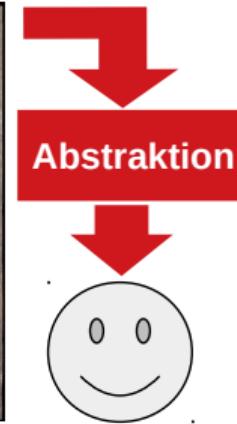
- ▶ Informatik = Systematische **Informationsverarbeitung**
(insbesondere mit Digitalrechnern)
- ▶ **Zwei Schlüsselschritte:** Abstraktion und Mechanisierung

“Computer Science = The Mechanization of Abstraction”

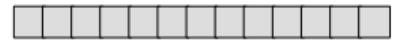
(Aho, Ullmann: Foundations of Computer Science)

Häufiges Vorgehen in der Informatik

1. Analyse des Problems
(Anforderungsanalyse)
2. Spezifikation der Lösung
(Pflichtenheft, System-Architektur)
3. Implementierung
(Programmiersprache)

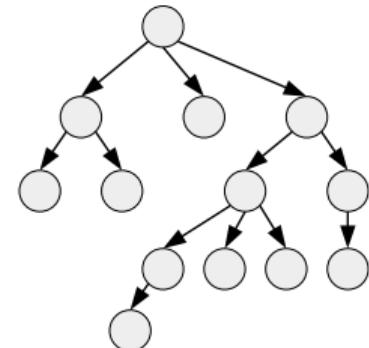


Weitere grundlegende Definitionen



Definition "Algorithmus" ?

- ▶ Algorithmus = “wohldefinierte” **Handlungsvorschrift, oft zur Lösung eines Problems** (*siehe nächste Seite*).
- ▶ Die **Ausführung** eines Algorithmus (auf einem *Prozessor*) erzeugt Ausgabeobjekte aus Eingabeobjekten.



Definition "Programm"

- ▶ Programm = **Umsetzung** eines Algorithmus in einer konkreten Programmiersprache (*z.B. Java*).

Definition "Datenstruktur"

- ▶ Organisation der Eingabe- und Ausgabe-Objekte.

Definition “Algorithmus”

Definition (Algorithmus (Saake, Sattler [4]))

Ein Algorithmus ist eine...

- ▶ präzise (d.h. in festgelegter Sprache)
- ▶ endliche Beschreibung eines
- ▶ allgemeinen Verfahrens unter Verwendung
- ▶ ausführbarer elementarer Schritte.

Beispiele

- ▶ Bedienungsanleitungen, Bauanleitungen (IKEA)
- ▶ Kochrezepte, Spielanleitungen
- ▶ Noten/Partituren
- ▶ Vorschriften zur Verarbeitung von Daten (*hier in ADS*).



“Algorithmus”: Beispiel (Ei kochen)

Ei kochen

1. Wasser in einem Topf zum Kochen bringen.
2. Ei in ein Sieb legen.
3. Sieb in Wasser senken.
4. Wenn Ei mittelgroß ist, dann 9 min kochen, wenn Ei groß ist, dann 10 min.
5. Sieb aus kochendem Wasser nehmen.
6. Sieb in kaltes Wasser tauchen.

Ist das ein Algorithmus?

Präzise?

- ▶ Eingabeobjekte genauer definieren, z.B. rohes Ei?

Endlich?

- ▶ ja (*6 Schritte*)

Allgemeines Verfahren?

- ▶ Lösung einer ganzen Problemklasse, allgemein anwendbar.
(Verhalten für kleine Eier ...?)

Ausführbare, elementare Schritte?

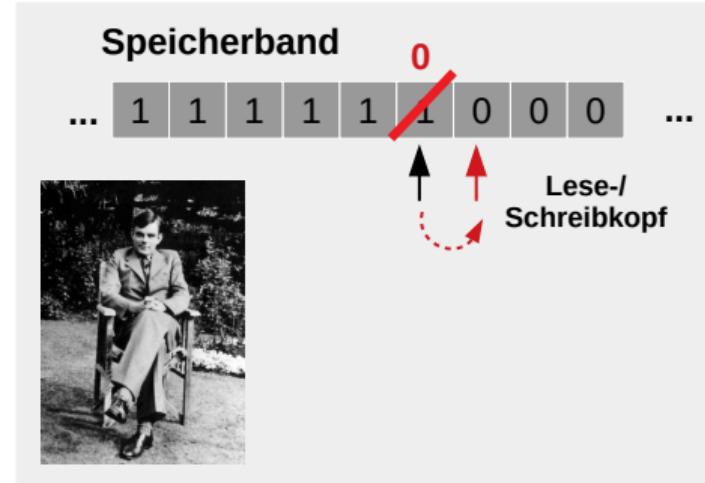
- ▶ Kommt auf den Prozessor an ;-)

“Algorithmus”: Beispiel (Turing-Maschine)

Bild: [1]

Andere Algorithmen-Darstellungen sind **deutlich formaler**, z.B. die sogenannte **Turing-Maschine**

- ▶ **Speicher:** ein unendliches Band.
- ▶ **Operationen:**
 - ▶ Lese 0/1 vom Band und wähle die nächste Aktion.
 - ▶ Bewege dich auf dem Band einen Schritt nach links/rechts.
 - ▶ Schreibe 0/1 auf das Band.



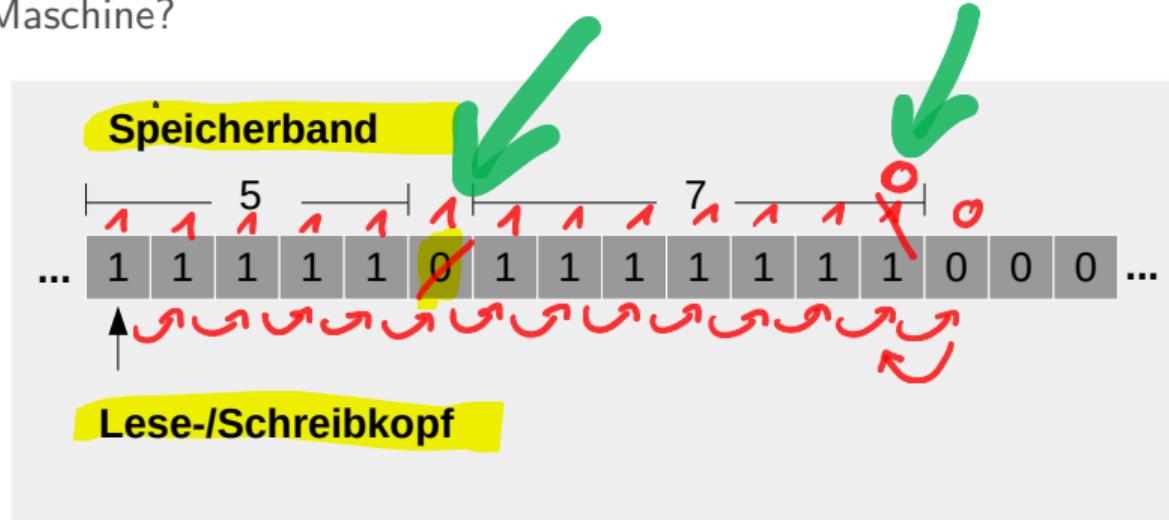
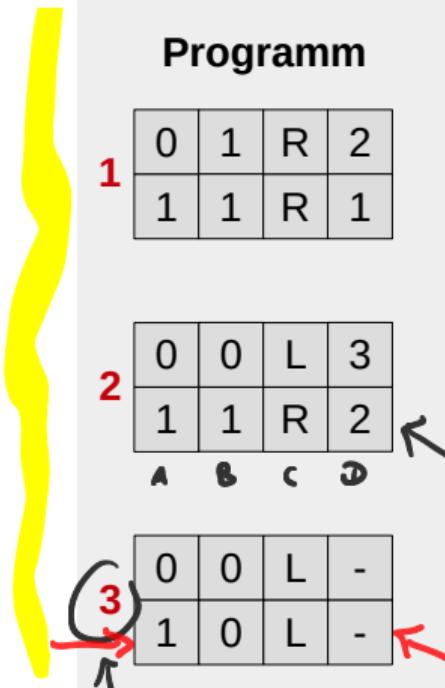
Church'sche These

- ▶ Die (Ausdrucks-)Mächtigkeit von Algorithmen in einer **beliebigen Programmiersprache** ist **gleich** der Mächtigkeit von **Turing-Maschinen**.
- ▶ **Algorithmen** = alles was programmierbar ist
= alles was eine Turing-Maschine ausführen kann.
- ▶ Diese nicht bewiesen (*schwierig!*), aber allgemein anerkannt.

Turing-Maschine: Beispiel

Addition zweier Zahlen! *

Was tut diese Turing-Maschine?



Wenn eine 1 gelesen wird (A), schreibe eine 1 (B),
gehe 1 nach Rechts(C) und wechsle in Zustand 2(D).

Beende.

Historie des Rechnens



300 v. Chr.: Euklid

- ▶ ältester bekannter Algorithmus (Bestimmung des größten gemeinsamen Teilers zweier Zahlen)



800 n. Chr.: Muhammed ibn Musa
abu Djafar **al-Chwarizmi**

- ▶ Buch "Al-Chwarizmi über die indischen Zahlen" →
"Algorithmi de Numero Indorum"



1550: Adam Riese

- ▶ Deutschsprachiges Rechenbuch¹
- ▶ Ablösung der lateinischen Ziffern.

¹<http://www.tinohempel.de/info/mathe/ries/ries.htm>

Historie des Rechnens



1703: G.W. Leibniz

- ▶ Duales Zahlensystem (*erste technische Nutzung: 1930er*)
- ▶ Differentialrechnung (*parallel zu Isaac Newton*).



1842: Charles Babbage

- ▶ Entwurf der *Analytical Engine*
- ▶ Erste Rechenmaschine äquivalent einer Turing-Maschine (*u.a. branching+loops*), programmierbar mit Lochkarten.



1931: Kurt Gödel

- ▶ Unvollständigkeitssatz für ausreichend mächtigen Rechensystemen (*z.B. Arithmetik über \mathbb{N}*).
- ▶ Hier gibt es Aussagen, die wir nicht automatisiert beweisen können.

Historie des Rechnens

ENIAC



1940er: Erste Computer

- ▶ Z3 (Zuse, Berlin)
- ▶ Colossus (Turing et al., Bletchley / UK), ENIAC (US Army).



1962: Donald Knuth

- ▶ Buch “The Art of Computer Programming” (TAoCP)
- ▶ Algorithmen in fiktiver Assembler-Sprache
- ▶ ADS-“Bibel” (*Suchen, Sortieren, Datenstrukturen, Scannen, ...*).



1973: Niklaus Wirth – Pascal

- ▶ **Strukturierte** Programmierung (*Zerlegung von Programmen in Prozeduren, Vermeidung von GoTos*).
- ▶ **Objektorientierte** Programmierung
→ seit den 1990ern.



Outline

1. Der Algorithmus-Begriff
2. Eigenschaften von Algorithmen
3. Beispiel: Primzahltest
4. Darstellung von Algorithmen
 - Pseudo-Code
 - Flussdiagramme
 - Struktogramme

Terminierung

Definition (Terminierung)

Wir nennen einen Algorithmus **terminierend** wenn er bei jeder erlaubten Eingabe nach **endlich vielen Schritten abbricht**.

Beispiele

- ▶ “Ei kochen” (*siehe oben*) ist terminierend.
- ▶ Algorithmen mit **Endlosschleife** sind nicht terminierend.
- ▶ Addition von natürlichen Zahlen $n_1 + n_2$ mittels Nachschauen in einer **Tabelle**...
 - ▶ ist **terminierend** (*Nachschauen in Tabelle = 1 Schritt*)
 - ▶ ist aber **kein Algorithms** (*Beschreibung/Tabelle wäre unendlich*).

| | 1 | 2 | 3 | 4 | ... |
|-----|-----|-----|-----|-----|-----|
| 1 | 2 | 3 | 4 | 5 | ... |
| 2 | 3 | 4 | 5 | 6 | ... |
| 3 | 4 | 5 | 6 | 7 | ... |
| ... | ... | ... | ... | ... | ... |

Determinismus und Determiniertheit

Definition (Determinismus)

Wir nennen einen Algorithmus **deterministisch** wenn bei gleicher Eingabe exakt dieselben **Schritte/Zustände** durchlaufen werden.

Definition (Determiniertheit)

Wir nennen einen Algorithmus **determiniert** wenn bei gleicher Eingabe immer dasselbe **Ergebnis** erzielt wird.

Beispiele

- ▶ Algorithmus “**Ei kochen**”: deterministischer Ablauf, determiniertes Ergebnis.
- ▶ Algorithmus “**Karten sortieren**” (*rechts oben*): nicht-deterministisch, determiniertes Ergebnis.

S sei ein
unsortierter
Kartenstapel

S' := ein leerer
Kartenstapel

wiederhole:

k := eine **zufällige Karte** aus S

Entferne k aus S.

Sortiere k an **passender Stelle** in S'
ein, so dass S' sortiert ist.

bis S leer. # S' enthält nun die
sortierten Karten.



Spezifikation und Korrektheit

Die meisten Algorithmen in ADS sind **determiniert und terminierend**. Wir können uns einen solchen Algorithmus als eine **Funktion f^{Alg}** vorstellen:

$$f^{\text{Alg}} : \text{Eingabewerte} \rightarrow \text{Ausgabewerte.}$$

Beispiel

Sortierverfahren: $\mathcal{A} = \text{Menge aller möglichen Arrays}$
 $= \{ [3,1,2,4], [1,2,3,4,5], [], \dots \}$

$$f^{\text{Alg}} : \mathcal{A} \rightarrow \mathcal{A}$$

$$\text{mit } f^{\text{Alg}}([3,1,2,4]) = [1,2,3,4]$$

$$f^{\text{Alg}}([1,2,1]) = [1,1,2]$$

...



Spezifikation und Korrektheit

Spezifikation

- ▶ In der Praxis sollen Algorithmen ein **konkretes Problem lösen**. Wir **spezifizieren** dieses Problem, indem wir angeben für welche Eingaben wir welche Ausgaben erwarten.
- ▶ Dieses Soll-Verhalten entspricht ebenfalls einer **Funktion**. Wir bezeichnen Sie als die **Spezifikation** des Algorithmus:

$$f^{Spez} : \text{Eingabewerte} \rightarrow \text{Ausgabewerte.}$$

Definition (Korrektheit)

Wir nennen einen Algorithmus genau dann **korrekt** wenn er für **jede mögliche Eingabe x** mit der durch die **Spezifikation** geforderten Ausgabe terminiert, d.h. $f^{Alg}(x) = f^{Spez}(x)$.

Korrektheit: Beispiel

- ▶ Prüfe ob a ein Teiler von b ist.
- ▶ $f^{Spez} : \mathbb{Z} \times \mathbb{Z} \rightarrow \{\text{true}, \text{false}\}$ mit

$$f^{Spez}(a, b) = \begin{cases} \text{true} & \text{falls } a|b \\ \text{false} & \text{sonst} \end{cases}$$

- ▶ **Algorithmus:** Prüft $1 \cdot a, 2 \cdot a, 3 \cdot a, \dots$ auf Gleichheit mit b .
- ▶ Ist der Algorithmus korrekt?

```
# prüft ob a ein
# Teiler von b ist.

test = a

while test <= b:

    if test == b:
        return true

    test += a

return false.
```



Korrektheit und Testen

```
# prüft ob a ein
# Teiler von b ist.

test = a

while test <= b:
    if test == b:
        return true
    test += a

return false.

#####
##### Testfälle #####
#####

# a ist Teiler
assert check(3, 72) == true

# a ist kein Teiler
assert check(3, 73) == false

# a == b
assert check(10, 10) == true

# a == 1, b groß
assert check(1, 10000) == true

# a > b
assert check(10, 9) == false
```

Wie zeigen wir Korrektheit?

1. Testen

- ▶ In der Praxis sehr verbreitet.
- ▶ Programmierung von **Testfällen**, **Prüfung** dass in diesen Fällen das gewünschte Ergebnis erzielt wird.
- ▶ Tests können **Nicht-Korrektheit zeigen**
(wenn ein Test fehlschlägt).
- ▶ Tests zeigen im Allgemeinen keine Korrektheit!
(wir können im Allgemeinen nicht alle Eingaben prüfen).

Beispiel (links)

- ▶ Die Tests sind alle erfolgreich, aber negative Werte wurden vergessen 😞
*(nicht-terminierend für $(a < 0 \wedge b > a)$ oder $(a = 0 \wedge b < a)$).
(nicht korrekt für $(a < 0 \wedge b < a \wedge a \neq b)$).*



2. Verifikation

Beweise Korrektheit (*händisch oder automatisiert*).

Beispiel: Der Hoare-Kalkül [3]

- Wir definieren Vorbedingungen P (*engl. “pre-conditions”*) an die Eingabe.
- Beim Ausführen imperativer Rechenoperationen S gelten bestimmte *Axiome*.
- Hieraus leiten wir Nachbedingungen Q (*post-conditions*) für die Ausgabe her.

$$\underbrace{\{x < 42\}}_P \quad y := x + 1 \quad \underbrace{\{y < 43\}}_Q$$



Outline

1. Der Algorithmus-Begriff
2. Eigenschaften von Algorithmen
3. Beispiel: Primzahltest
4. Darstellung von Algorithmen
 - Pseudo-Code
 - Flussdiagramme
 - Struktogramme

Beispiel: Analyse von “Primzahltest”

Spezifikation

- Eine natürliche Zahl $n \geq 2$ ist **prim** wenn sie (außer sich selbst) keine Teiler ≥ 2 besitzt.
- Gegeben eine Zahl $n \geq 2$, prüfe ob n prim ist.

Eigenschaften des Algorithmus

- ✓ ► deterministisch (es werden bei gleicher Eingabe immer die gleichen Schritte durchlaufen).
- ✓ ► determiniertes Ergebnis.
- ? ► Terminierend?
- ? ► Korrekt?

Ist 17 Prim? 2|17?

nein

3|17? 4|17 ... $x=17$
nein nein Abbruch
in Schritt 2 ✓

Primzahltest

1. Weise x den Wert 2 zu.
2. Wenn x gleich n ist, dann gebe **true** zurück und terminiere.
3. Dividiere n durch x mit Rest
4. Wenn diese Division den Rest 0 ergibt, dann gebe **false** zurück und terminiere.
5. Erhöhe den Wert von x um 1 und **gehe zu Schritt 2**.

Beispiel: Analyse von "Primzahltest"

Terminieren? (Endlosschleife in Schritt 2-5?)

- x wird in Schritt 5 immer weiter erhöht.
- Fall(a): Wir finden einen Wert $x < n$ mit $x | n$

→ Abbruch ✓

- Fall(b): Wir finden kein solches x

→ x erreicht irgendwann den Wert n

→ Abbruch (Schritt 2) ✓

Primzahltest

1. Weise x den Wert 2 zu.
2. Wenn x gleich n ist,
dann gebe **true** zurück und terminiere. (b)
3. Dividiere n durch x mit Rest
4. Wenn diese Division den Rest 0 ergibt, dann gebe **false** zurück und terminiere. (a)
5. Erhöhe den Wert von x um 1 und **gehe zu Schritt 2**.

Korrektheit? Zeige: (a) n ist prim \rightarrow Ergebnis ist true.
(b) n "nicht prim" \rightarrow " " false.

Beispiel: Analyse von "Primzahltest" $n=5 \cdot 7$

(b) n ist nicht prim.

→ Es gibt einen kleinsten Teiler $a \geq 2$ (und $a < n$) mit $a|n$.

→ In Schritt 5 erreicht x den Wert a .

→ In Schritt 4 wird der Wert false zurückgegeben. ✓

(a) n ist prim.

→ Es gibt keinen Teiler a so dass $a \geq 2$ und $a < n$ und $a|n$.

→ In Schritt 4 gilt wie $x|n$ und es wird wie false zurückgegeben.

→ x erreicht den Wert n , und in Schritt 2 wird true zurückgegeben. ✓

Primzahltest

1. Weise x den Wert 2 zu.

2. Wenn x gleich n ist,
dann gebe **true** zurück
und terminiere.

3. Dividiere n durch x mit Rest

4. Wenn diese Division den
Rest 0 ergibt, dann gebe **false**
zurück und terminiere.

5. Erhöhe den Wert von x um 1
und **gehe zu Schritt 2**.



Outline

1. Der Algorithmus-Begriff
2. Eigenschaften von Algorithmen
3. Beispiel: Primzahltest
4. Darstellung von Algorithmen

Pseudo-Code

Flussdiagramme

Struktogramme

Bausteine von Algorithmen



Elementare Operationen

... sind Schritte die nicht weiter zerlegt werden müssen.

- ▶ *“Schneide Fleisch in kleine Würfel”*
- ▶ **Im Rechner:** Zuweisungen, Additionen, Vergleiche, ...

Sequenzen von Operationen

... sind Schritte die nacheinander ausgeführt werden.

- ▶ *“Bringe Wasser zum Kochen und lege dann das Ei hinein.”*
- ▶ **Im Rechner:** Anweisungssequenzen, z.B. $y = x+1$; $z = 2*y$;

Parallele Ausführung

... mehrere Prozessoren arbeiten gleichzeitig an Teilschritten:

- ▶ *“Ich schneide Fleisch, Du Gemüse. Anschließend geben wir beides in die Pfanne.”*
- ▶ **Im Rechner:** Threads, Prozesse.

Bausteine von Algorithmen (cont'd)

Bedingte Operationen

... Schritte werden nur ausgeführt wenn Bedingungen erfüllt sind.

- ▶ *“Wenn die Soße zu dünn ist, dann füge Mehl hinzu”.*
- ▶ **Im Rechner:** if-else-Konstrukte.



Schleifen

... Wiederholung einer Ausführung bis Endbedingung erreicht.

- ▶ *“Rühre, bis die Soße braun ist”*
- ▶ **Im Rechner:** z.B. while $x \geq 10$: $x = x - 1$;

Unterprogramm/Prozedur/Funktion/Methode

... Ausführen eines benannten Teilalgoritmus.

- ▶ *“Koche Nudeln (siehe Seite X)”*
- ▶ **Im Rechner:** z.B. `abs(x)`

Bausteine von Algorithmen (cont'd)

Rekursion

... reduziert das Problem auf ein kleineres, ähnliches Teilproblem.



- ▶ **würfleKäse(käse):**
 - Teile Käse in zwei Hälften.
 - Falls Käse noch nicht klein genug:
 - würfleKäse(linkeHälfte)
 - würfleKäse(rechteHälfte).
- ▶ **Im Rechner:** rekursive Funktionen

```
int fakultaet(n):  
    if n>0:  
        return n * fakultaet(n-1)  
    else:  
        return 1
```



Outline

1. Der Algorithmus-Begriff
2. Eigenschaften von Algorithmen
3. Beispiel: Primzahltest
4. Darstellung von Algorithmen

Pseudo-Code

Flussdiagramme

Struktogramme

Pseudo-Code...

... stellt das **Wesentliche eines Algorithmus** prägnant dar.

- ▶ unabhängig vom Prozessor (*Abstraktion!*)
- ▶ **Details** der maschinellen Verarbeitung werden **ausgelassen** (z.B. Variablen-Deklarationen, einfache Subroutinen wie `sqrt`).
- ▶ **Mischung** von
 - (1) Elementen von Programmiersprachen
 - (2) natürlicher Sprache
 - (3) mathematischer Notation.

Pseudo-Code dient der **Kommunikation zwischen Informatikern!**

- ▶ **Ziel:** Vollständige und eindeutige Beschreibung, die **einfach** in eine Programmiersprache **überführbar** ist.
- ▶ Es existiert **keine formale Syntax**
("... Pseudo-Code ist das was angemessen ist").

S sei ein
unsortierter
Kartenstapel

S' := ein leerer
Kartenstapel

wiederhole:

k := eine **zufällige Karte** aus S

Entferne k aus S.

Sortiere k an **passender Stelle** in S'
ein, so dass S' sortiert ist.

bis S leer. # S' enthält nun die
sortierten Karten.



Pseudo-Code: Konventionen

Anmerkungen

- Pseudo-Code ist gemäß guter Praxis eng an **Programmiersprachen** angelehnt.
Optionen sind z.B. ...

```
// Java  
if (a==b) { a = c; }
```

```
# Python  
if a==b :  
    a = c
```

```
// Pascal  
if a = b then begin a := c; end
```

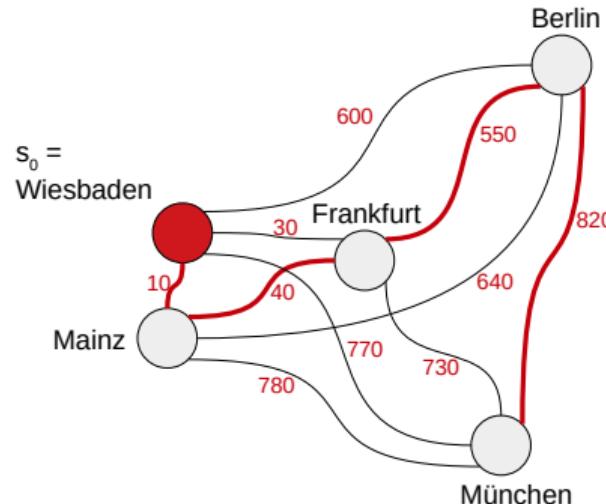
- Pseudo-Code ist oft genauer und klarer als **natürliche Sprache** (*u.a. eindeutige Benamung der verwendeten Objekte, eindeutige Abbruchkriterien für Schleifen ...*).

Pseudo-Code: Beispiel “Routenplaner” (schlecht)

```
# Routenplaner (Version 1)

Beginne mit der Route in Stadt  $s_0$ 

wiederhole:
    finde die nächstgelegene Stadt  $s$ .
    Füge  $s$  der Route hinzu.
bis alle Städte besucht.
```



Version 1

Hier existieren **Mehrdeutigkeiten / Unklarheiten**:

- ▶ Definition “Route”? Wie sind Ein- und Ausgabe strukturiert?
- ▶ Naive Interpretation des Codes führt zu Endlosschleife
($Wiesbaden \rightarrow Mainz \rightarrow Wiesbaden \rightarrow Mainz \rightarrow \dots$)



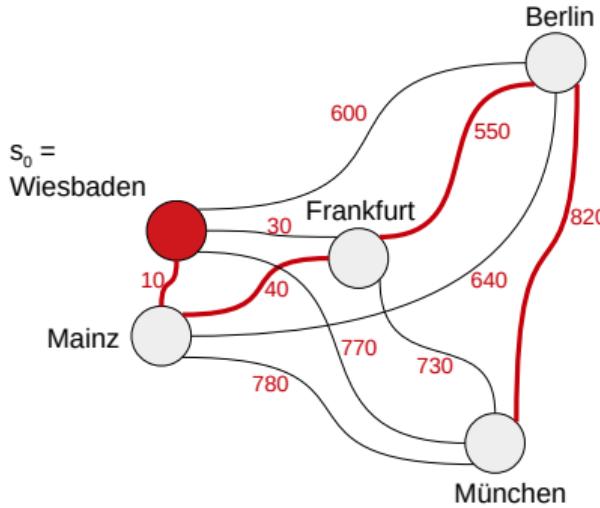
Pseudo-Code: Beispiel “Routenplaner” (gut)

```
# Routenplaner (Version 2)
# Gegeben:
# - S: Die Menge zu besuchender Städte
# -  $s_0$ : Die Anfangsstadt der Route

stadt :=  $s_0$ 
route := [] # leere Liste

while S <> {}:
    stadt := argmin $_{s \in S}$  cost(stadt,s)
    route := route + [stadt]
    S     := S\{stadt}

return route
```



Version 2

- **Formal:** Städte als Menge S , Route als Liste, Kostenfunktion $cost : S \times S \rightarrow \mathbb{R}^+$
- **Präziser, eindeutiger (mathematische Notation).**
- Unmittelbar in ein Programm **umsetzbar** (*wenn geeignete Datenstrukturen für Listen und Mengen bekannt → später*).



Outline

1. Der Algorithmus-Begriff
2. Eigenschaften von Algorithmen
3. Beispiel: Primzahltest
4. Darstellung von Algorithmen

Pseudo-Code

Flussdiagramme

Struktogramme

Flussdiagramme

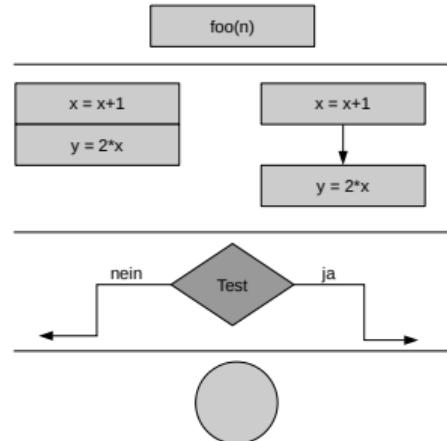
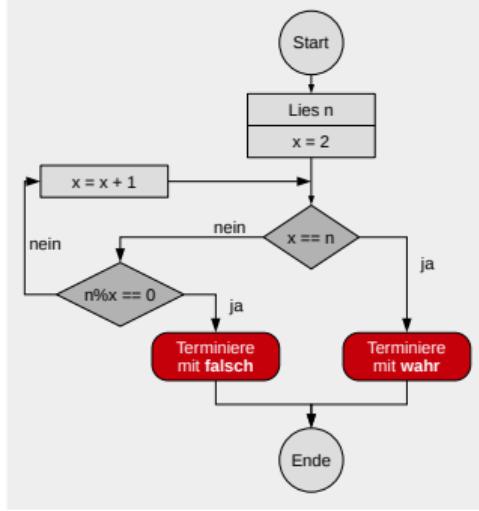
... stellen den Kontrollfluss eines Algorithmus graphisch (*mittels Pfeilen*) dar.

Elemente von Flussdiagrammen (rechts unten)

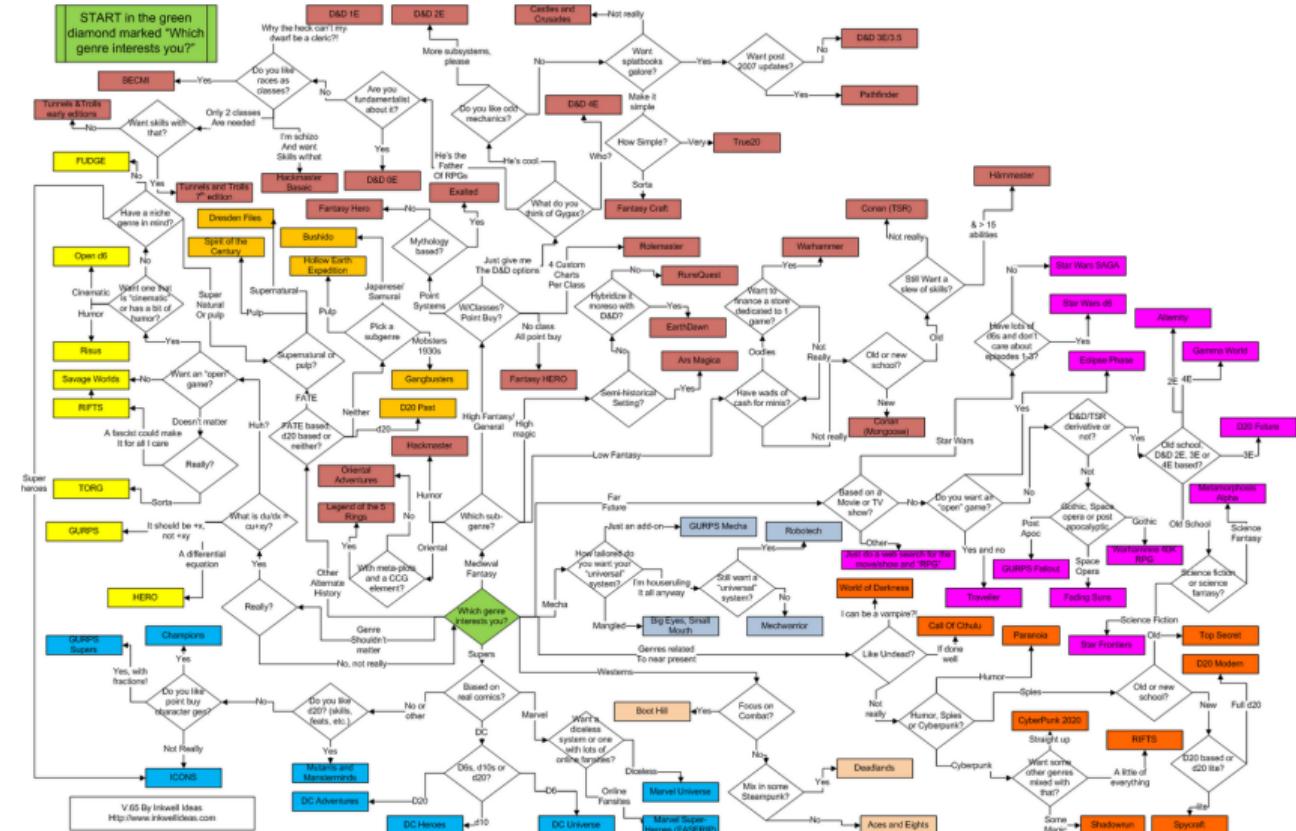
- ▶ Elementare Anweisungen,
Unterprogrammaufrufe
- ▶ Sequenz
- ▶ Verzweigung
- ▶ Start, Ende

Diskussion

- ▶ keine dedizierten Schleifenkonstrukte
- ▶ häufig unübersichtlich
(für komplexere Algorithmen vermeiden).



Flussdiagramm: Beispiel





Outline

1. Der Algorithmus-Begriff
2. Eigenschaften von Algorithmen
3. Beispiel: Primzahltest
4. Darstellung von Algorithmen

Pseudo-Code

Flussdiagramme

Struktogramme

Struktogramme ("Nassi-Schneiderman"-Diagramme)

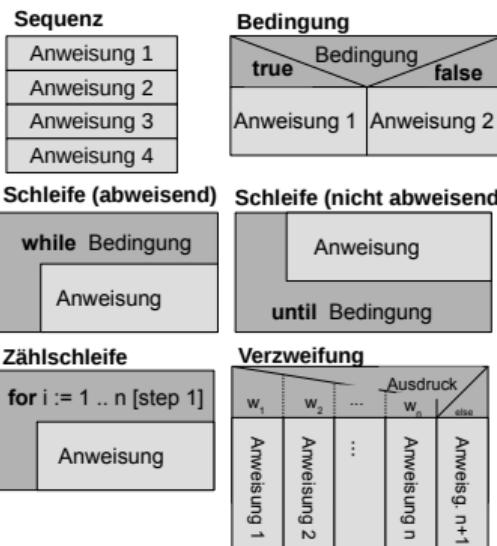
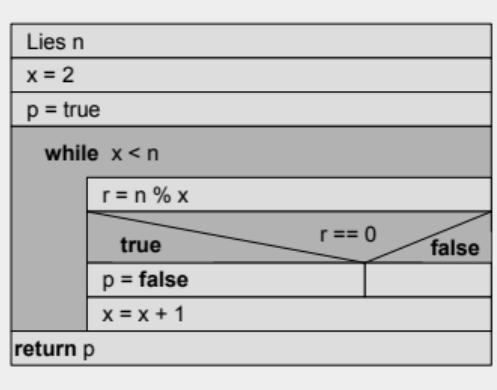
- sind (ähnlich wie Pseudo-Code und und Flussdiagramme) eine halbformale Darstellung.
- Zerlegung des Gesamtalgorithmus in **Teilprobleme**, (*strukturierte Programmierung*). Separates Struktogramm je Teilproblem.

Elemente

- Sequenzen, Verzweigungen
- Schleifen: Abweisend, nicht-abweisend, Zählschleifen

Diskussion

- Einfach erständlich.
- Jede Struktur hat einen klar definierten Eingang+Ausgang, Sprünge nicht erlaubt.





References I

- [1] Alan Turing (ca. 1938).
https://de.wikipedia.org/wiki/Alan_Turing#/media/File:Alan_Turing_az_1930-as_%C3%A9vekben.jpg (retrieved: Apr 2019).
- [2] Sir Charles Antony Richard Hoare giving a talk at the EPFL on 20th of June 2011.
https://de.wikipedia.org/wiki/Tony_Hoare#/media/File:Sir_Tony_Hoare_IMG_5125.jpg (retrieved: Apr 2019).
- [3] C. A. R. Hoare.
An axiomatic basis for computer programming.
Commun. ACM, 12(10):576–580, October 1969.
- [4] G. Saake and K.U. Sattler.
Algorithmen und Datenstrukturen: Eine Einführung mit Java.
Dpunkt.Verlag GmbH, 2013.

Bisschen eng?