

Informatik

12.Klasse

Arne Terkowski

I. FORMALE SPRACHEN	2
1. Natürliche und formale Sprachen.....	2
2. Erzeugung formaler Sprachen.....	4
3. Notationsformen	8
4. Erkennung formaler Sprachen.....	10
Endliche Automaten	10
Zusammenhang zwischen Grammatik und DEA	11
Grenzen endlicher Automaten	12
5. Endliche Automaten und deren Implementierung	14
II. KOMMUNIKATION UND SYNCHRONISATION VON PROZESSEN	14
1. Protokolle zur Kommunikation von Prozessen.....	14
2. Nebenläufigkeit.....	17
3. Implementierung.....	19
4. Topologie von Rechnernetzen	21
6. Kommunikation über Protokollschichten	23
7. Internet – Das Netz der Netze.....	25
III. FUNKTIONSWEISE EINES RECHNERS	26
1. Von-Neumann-Rechner	26
2. Registermaschine	27
IV. GRENZEN DER BERECHENBARKEIT.....	29
1. Laufzeit von Algorithmen	29

I. Formale Sprachen

1. Natürliche und formale Sprachen



natürliche Sprachen

- haben sich über eine lange Zeit entwickelt,
- entwickeln sich weiter,
- sind u.U. mehrdeutig
- Bedeutung von Wörter ist u.U. missverständlich
- Semantik wird u.U. trotz fehlerhafter Syntax erkannt
- Buchstabe → Wort → Satz

formale Sprachen

- künstliche / konstruierte Sprache
- i.a. sehr strenge Regeln
- Semantik spielt *keine* Rolle
- Buchstabe → Wort

Programmiersprache

- ähnlich einer formale Sprache
- Semantik ist von Bedeutung

Beispiele:

1. Heute Wetter schön

2. a0b1b1a0a1b0

3.

```
n=1
for i in range(1,10):
    n=n*i
print(n)
```

4. $(5+8)/5$ $5+(8/5)$

Dem Text in Beispiel 3 scheint folgende Regel zugrunde zu liegen:

- Das Wort besteht aus *Buchstaben* des *Alphabets* $\{a,b,0,1\}$.
- Die Konstruktionsregel lautet: Jedem a bzw. b folgt eine 0 oder 1

Begriffe (im Sinne der Informatik):

Die Syntax einer formalen Sprache umfasst sämtliche (grammatikalischen) Regeln, die zu korrekt gebildeten Zeichenketten führt.

Die Bedeutung eines Worts nennt man Semantik.

Das Alphabet Σ beinhaltet die zulässigen Zeichen, die zur Bildung der Wörter herangezogen werden dürfen.

Beispiel:

Satzbau im Englischen gemäß SPO (Subjekt–Prädikat–Objekt):

Subjekt	→	She
Subjekt	→	He
Prädikat	→	loves
Prädikat	→	kicked
Prädikat	→	likes
Objekt	→	soccer players
Objekt	→	swimming
Objekt	→	the ball

Anzahl der Sätze, die gemäß der SPO–Regel gebildet werden können: 18

Offensichtlich spielt die Semantik in dieser Sprache keine Rolle!

2. Erzeugung formaler Sprachen

langweilig: Aufzählen aller Wörter

Begriffe

- Unter Terminalen T versteht man die Symbole, die zum Alphabet Σ der formalen Sprache gehören.
- Nichtterminale N kommen im Wort nicht vor, sind aber zur Beschreibung der Regeln einer Sprache notwendig.
- Durch Anwendung der Produktionsregeln P wird ein Wort sukzessive aus einem Startsymbol s erzeugt. Dieses Startsymbol entstammt der Menge N .

Englisches Beispiel von oben:

$T = \{\text{He, She, loves, kicked, likes, soccer players, swimming, the ball}\}$

$N = \{\langle \text{Satz} \rangle, \langle \text{Subjekt} \rangle, \langle \text{Prädikat} \rangle, \langle \text{Objekt} \rangle\}$

$P = \{$
 $\langle \text{Satz} \rangle \rightarrow \langle \text{Subjekt} \rangle \langle \text{Prädikat} \rangle \langle \text{Objekt} \rangle ,$
 $\langle \text{Subjekt} \rangle \rightarrow \text{He} ,$
 $\langle \text{Subjekt} \rangle \rightarrow \text{She} ,$
 $\langle \text{Prädikat} \rangle \rightarrow \text{loves} ,$
 $\langle \text{Prädikat} \rangle \rightarrow \text{kicked} ,$
 $\langle \text{Prädikat} \rangle \rightarrow \text{likes} ,$
 $\langle \text{Objekt} \rangle \rightarrow \text{soccer players} ,$
 $\langle \text{Objekt} \rangle \rightarrow \text{the ball} \quad \}$

Der Satz "He likes swimming" ist mit dem Startsymbol $s = \text{"Satz"}$ ableitbar.

Kurzform für P mit dem oder-Operator $|$:

$P = \{$
 $\langle \text{Satz} \rangle \rightarrow \langle \text{Subjekt} \rangle \langle \text{Prädikat} \rangle \langle \text{Objekt} \rangle ,$
 $\langle \text{Subjekt} \rangle \rightarrow \text{He} \mid \text{She} ,$
 $\langle \text{Prädikat} \rangle \rightarrow \text{loves} \mid \text{kicked} \mid \text{likes} ,$
 $\langle \text{Objekt} \rangle \rightarrow \text{soccer players} \mid \text{the ball} \quad \}$

Noch ein Beispiel ("ausgewogene Bitmuster"):

Über den Terminalzeichen der Menge $\{0; 1\}$ wird eine Sprache betrachtet, bei der jedes Wort gleich viele Einsen wie Nullen enthalten muss. Die Worte 01, 10, 001011 oder 110010 gehören also dazu.

Als Nichtterminalzeichen dienen S, A und B, wobei S das Startzeichen ist.

Dabei sollen aus A alle Wörter herleitbar sein, die genau eine Eins mehr als Nullen enthalten und aus B soll man alle Wörter erhalten, die genau eine 0 mehr als Einsen besitzen.

Das führt zu folgenden Grammatikregeln:

- (1) S \rightarrow 0A | 1B
- (2) A \rightarrow 1 | 1S | 0AA
- (3) B \rightarrow 0 | 0S | 1BB

Das Wort 110010 kann dann wie folgt abgeleitet werden:

S \Rightarrow 1B \Rightarrow 11BB \Rightarrow 11B0 \Rightarrow 110S0 \Rightarrow 1100A0 \Rightarrow 110010

Anderes Beispiel von oben (a0b1b1a0a1b0)

$$T = \{a, b, 0, 1\}$$

$$N = \{ \langle \text{wort} \rangle, \langle \text{buchstabe} \rangle, \langle \text{zahl} \rangle \}$$

$$P = \{ \begin{array}{ll} \langle \text{wort} \rangle & \rightarrow \langle \text{wort} \rangle \langle \text{buchstabe} \rangle \langle \text{zahl} \rangle \mid \langle \text{buchstabe} \rangle \langle \text{zahl} \rangle, \\ \langle \text{buchstabe} \rangle & \rightarrow a \mid b, \\ \langle \text{zahl} \rangle & \rightarrow 0 \mid 1 \end{array} \}$$

$s = \langle \text{wort} \rangle$

Beachte, dass in obiger Produktion (in der ersten Produktionsregel) eine Rekursion auftritt.

Bemerkung:

Bei einer verschränkten Rekursion erstreckt sich die Rekursion über mehrere Regeln.

einfaches Beispiel:

$$\begin{array}{ll} \langle \text{wort} \rangle & \rightarrow \langle \text{ausdruck} \rangle \\ \langle \text{ausdruck} \rangle & \rightarrow \text{wort} \langle \text{buchstabe} \rangle \langle \text{zahl} \rangle \mid \langle \text{buchstabe} \rangle \langle \text{zahl} \rangle, \\ \langle \text{buchstabe} \rangle & \rightarrow a \mid b, \\ \langle \text{zahl} \rangle & \rightarrow 0 \mid 1 \end{array}$$

schwierigeres Beispiel:

$\langle \text{Ausdruck} \rangle = ['-'] \langle \text{Term} \rangle \{ ('+' \mid '-') \langle \text{Term} \rangle \}$
 $\langle \text{Term} \rangle = \langle \text{Faktor} \rangle \{ '*' \langle \text{Faktor} \rangle \}$
 $\langle \text{Faktor} \rangle = '(' \langle \text{Ausdruck} \rangle ')' \mid \langle \text{Zahl} \rangle$

Verwendet man die EBNF (Erweiterte Backus Naur Form), so sind Terme in eckigen Klammern [...] optional, Terme in geschweiften Klammern {...} können beliebig oft wiederholt werden.

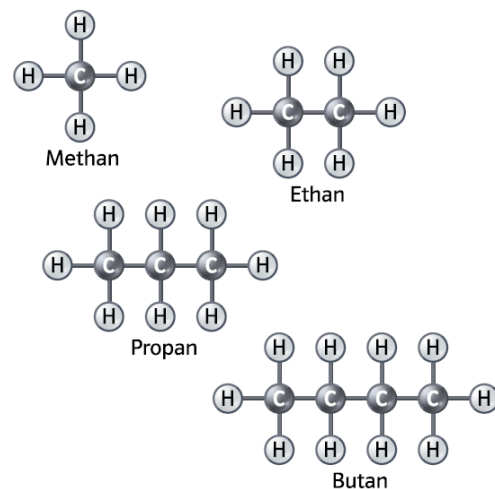
Ableitungsbäume am Beispiel der Alkane (Buch S.17)

Allgemeine Summenformel: $C_n H_{2n+2}$

genauer:

- CH_4 oder

- $CH_3 - \underbrace{CH_2 - \dots - CH_2}_{n\text{-mal}, n \in \mathbb{N}_0} - CH_3$



Grammatik $G = \{V, \Sigma, P, s\}$

Alphabet / Terminale: $\Sigma = \{C, H, -, \varepsilon\}$

Nonterminale: $N = \{\langle \text{Alkan} \rangle, \langle \text{Randgruppe} \rangle, \langle \text{Innengruppe} \rangle\}$

$P = \{$

$\langle \text{Alkan} \rangle \rightarrow 'CH_4' \mid \langle \text{Randgruppe} \rangle \langle \text{Innengruppe} \rangle '-' \langle \text{Randgruppe} \rangle ,$
 $\langle \text{Innengruppe} \rangle \rightarrow \varepsilon \mid 'CH_2' \mid \langle \text{Innengruppe} \rangle ,$
 $\langle \text{Randgruppe} \rangle \rightarrow 'CH_3' \}$

$s = \langle \text{Alkan} \rangle$

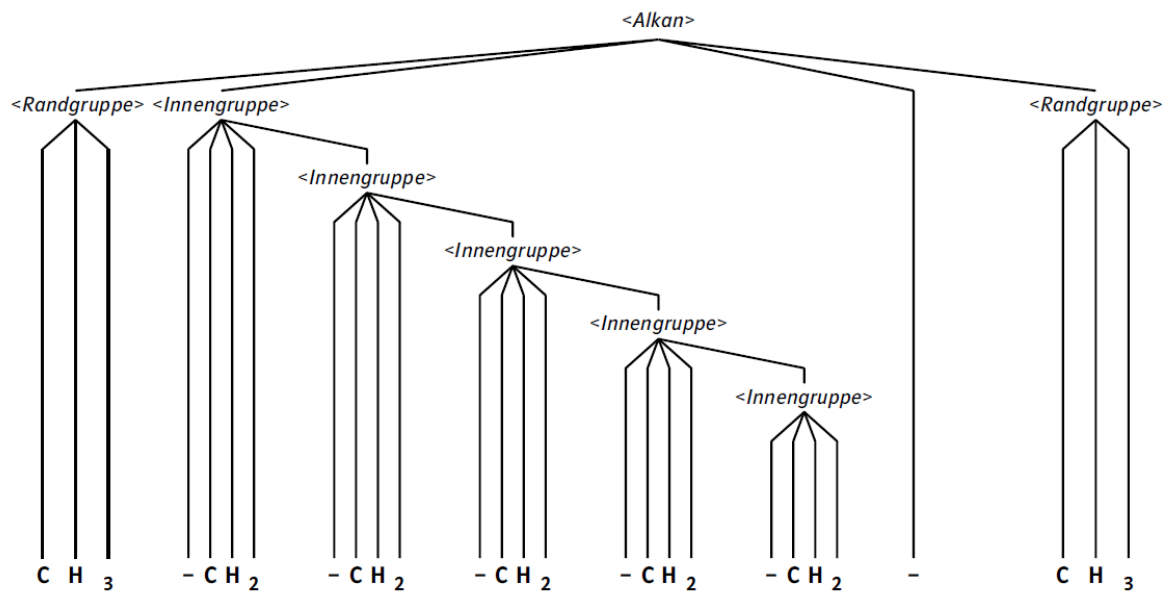
Bemerkungen:

- ε bezeichnet das leere Wort
- ohne ε könnte man auch schreiben:
 $\langle \text{Alkan} \rangle \rightarrow 'CH_4' \mid \langle \text{Randgruppe} \rangle \langle \text{Innengruppe} \rangle '-' \langle \text{Randgruppe} \rangle \mid \langle \text{Randgruppe} \rangle '-' \langle \text{Randgruppe} \rangle$
 $\langle \text{Innengruppe} \rangle \rightarrow 'CH_2' \mid \langle \text{Innengruppe} \rangle$

- In EBNF wären die Produktionsregeln noch einfacher, z.B. so:

$\langle \text{Alkan} \rangle \rightarrow 'CH_4' \mid \langle \text{Randgruppe} \rangle [\langle \text{Innengruppe} \rangle] '-' \langle \text{Randgruppe} \rangle ,$
 $\langle \text{Innengruppe} \rangle \rightarrow \varepsilon \mid \{ 'CH_2' \} ,$
 $\langle \text{Randgruppe} \rangle \rightarrow 'CH_3'$

Ableitungsbaum (für Heptan):



(2) als Folge von Regelanwendungen (Linksableitung):

<Alkan> → <Randgruppe><Innengruppe>'-'<Randgruppe>
 → 'CH₃'<Innengruppe>'-'<Randgruppe>
 → 'CH₃-CH₂'<Innengruppe>'-'<Randgruppe>
 → 'CH₃-CH₂-CH₂'<Innengruppe>'-'<Randgruppe>
 → 'CH₃-CH₂-CH₂-CH₂'<Innengruppe>-<Randgruppe>
 → 'CH₃-CH₂-CH₂-CH₂-CH₂'<Innengruppe>-<Randgruppe>
 → 'CH₃-CH₂-CH₂-CH₂-CH₂-CH₂'<Randgruppe>
 → 'CH₃-CH₂-CH₂-CH₂-CH₂-CH₂-CH₃'

3. Notationsformen

Grammatik:

Grammatik: $G = \{V, \Sigma, P, s\}$
 V bzw. N : Variablen (Nonterminale)
 Σ : Alphabet
 P : Produktionsregeln
 s : Startsymbol

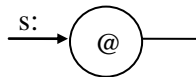
EBNF (Erweiterte Backus-Naur-Form):

- für die Produktionsregeln einer Grammatik.
- verwende Gleichheitszeichen = statt des Pfeils
- jede Zeile endet mit einem Strichpunkt (manchmal auch Punkt)
- die spitzen Klammern können weggelassen werden

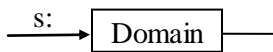
Die EBNF ist von der ISO standardisiert: *ISO/IEC 14977:1996(E)*. Es gibt aber verschiedene gebräuchliche "Dialekte".

Syntaxdiagramme:

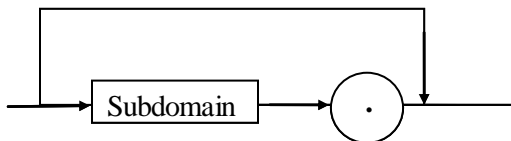
Terminal



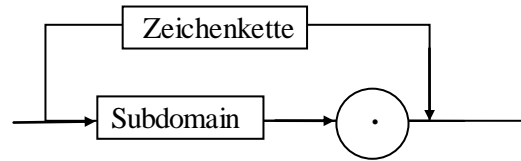
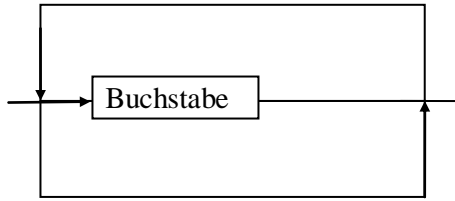
Nonterminal / Variable



Wiederholung

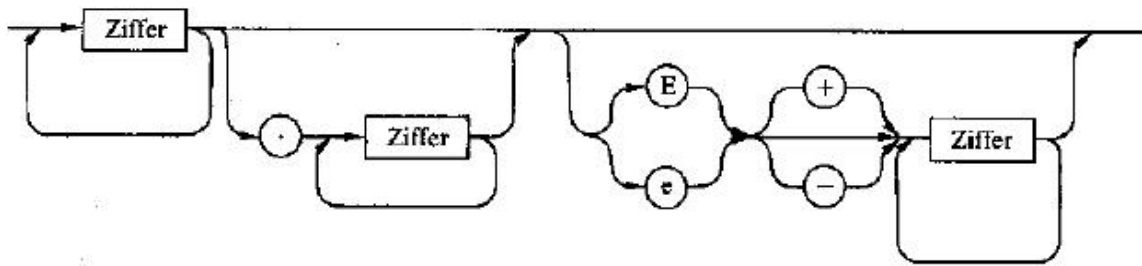


Alternative



Beispiel: Dezimalzahlen

Dezimalzahl = Zahl ["." Zahl] [Exponent] ;
 Zahl = Ziffer { Ziffer } ;
 Exponent = ("E"|"e") ["+"|"−"] Zahl ;
 Ziffer = "0" | "1" | ... | "9" ;

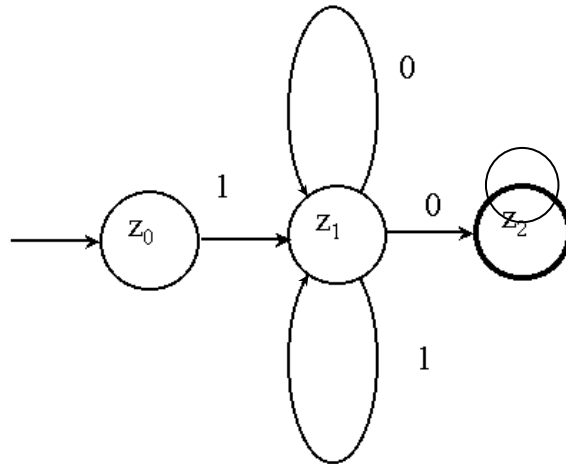


4. Erkennung formaler Sprachen

Endliche Automaten

Der im Bild rechts dargestellte Automat erkennt alle Dualzahlen, die mit 1 beginnen und mit 0 enden, also alle korrekt geschriebenen geraden Dualzahlen.

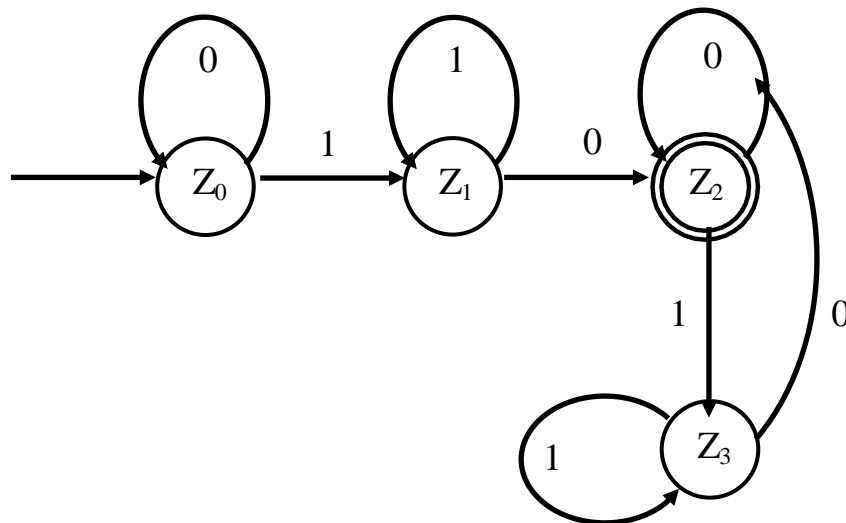
Zustandsmenge: Z
Eingabealphabet: 0,1
Startzustand: z_0
Endzustand : z_2



Der Automat liest ein Wort zeichenweise und geht in Abhängigkeit vom aktuellen Zustand und vom gelesenen Zeichen in einen Folgezustand über. Ist das Wort abgearbeitet und befindet sich der Automat in einem sogenannten Endzustand, so gilt das Wort als akzeptiert, andernfalls nicht.

Es handelt sich hier um einen nichtdeterministischen endlichen Automaten (NEA), da in Zustand z_1 beim Lesen einer 0 nicht klar ist, welches der Folgezustand ist. Der Automat ist endlich, weil seine Zustandsmenge endlich ist.

Ein fast gleichwertiger, deterministischer endlicher Automat (DEA) könnte so aussehen:



Deterministische Endliche Automaten (DEA) – Definition

1. Endliche Menge Z von Zuständen
2. Endliches Eingabealphabet
3. Definierter Startzustand
4. Menge E von Endzuständen
5. Übergangsfunktion δ , die jedem Paar aus aktuellem Zustand und gelesenen Zeichen wohldefiniert einen Folgezustand zuordnet.

Oft führt man sogenannte Fang- bzw. Fehlerzustände ein, in denen der Automat stehen bleibt, wenn ein nicht akzeptiertes Wort gelesen wurde. Hier könnte man dies z.B. bei einer führenden Null machen. Diese sind allerdings keine Endzustände.

Übergangsfunktion im Fall des DEA, der die geraden Dezimalzahlen akzeptiert:

Zustand gelesen	z_0	z_1	z_2	z_3
0	z_0	z_2	z_2	z_2
1	z_1	z_1	z_3	z_3

Schreibweise für Transitionen:

$z_1 \xrightarrow{a} z_2$ bzw. $\delta(z_1, 'a') = z_2$ bedeutet:

Ist das System im Zustand z_1 und erhält die Eingabe 'a', so erfolgt der Übergang in den Zustand z_2 .

Zusammenhang zwischen Grammatik und DEA

Es ist offensichtlich, dass jeder DEA die Wörter einer bestimmten Grammatik erkennt. Diese Grammatik lässt sich wie folgt bilden:

1. Die Zustandsmenge ergibt die Menge der Nichtterminale
2. Das Eingabealphabet Σ ergibt die Menge der Terminale
3. Der Startzustand geht über in das Startwort
4. Jede Transition $z_1 \xrightarrow{a} z_2$ der Übergangsfunktion findet ihre Entsprechung in der Produktion $z_1 \rightarrow 'a' z_2$
5. Für jeden Endzustand z_{end} wird zusätzlich Produktionsregel $z_{end} \rightarrow \varepsilon$ eingeführt.

Beispiel:

$$T = \{0, 1\}$$

$$N = \{ \langle z_0 \rangle, \langle z_1 \rangle, \langle z_2 \rangle, \langle z_3 \rangle \}$$

$$P = \{$$

$$\langle z_0 \rangle \rightarrow 0 \langle z_0 \rangle \mid 1 \langle z_1 \rangle ,$$

$$\langle z_1 \rangle \rightarrow 0 \langle z_2 \rangle \mid 1 \langle z_1 \rangle ,$$

$$\langle z_2 \rangle \rightarrow 0 \langle z_2 \rangle \mid 1 \langle z_3 \rangle \mid \varepsilon ,$$

$$\langle z_3 \rangle \rightarrow 0 \langle z_2 \rangle \mid 1 \langle z_3 \rangle \}$$

$$s = \langle z_0 \rangle$$

Grenzen endlicher Automaten

Es gilt: DEA \rightarrow Grammatik

Genauer: Grammatiken, die sich aus deterministischen endlichen Automaten ableiten, nennt man reguläre Grammatiken. Ihr Produktionen können immer in folgender Weise geschrieben werden:

$\langle \text{Nonterminal} \rangle \rightarrow 'a' \langle \text{Nonterminal}_2 \rangle$ oder
 $\langle \text{Nonterminal} \rangle \rightarrow 'a'$ oder
 $\langle \text{Nonterminal} \rangle \rightarrow \varepsilon$

Bemerkung:

Eine solche Grammatik heißt rechtsregulär. Analog gibt es auch linksreguläre Grammatiken.

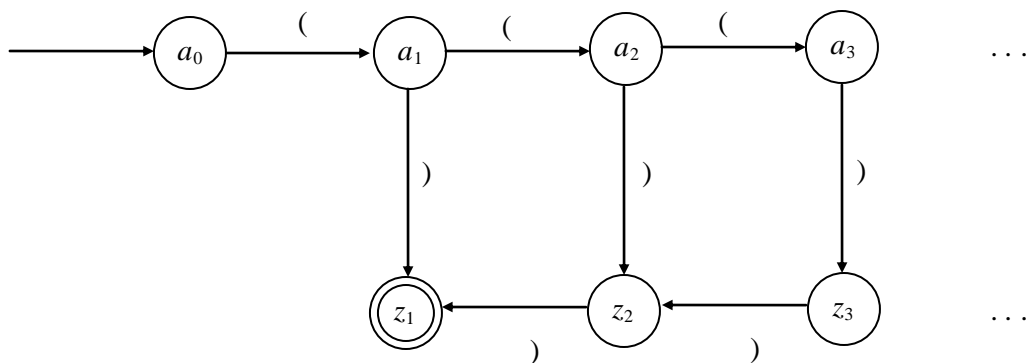
Folgende Aussage ist richtig:

DEA \Leftrightarrow reguläre Grammatik

Es gibt aber (somit nichtreguläre) Grammatiken, die keine Entsprechung in einem deterministischen endlichen Automaten haben.

Ein klassisches Beispiel beschäftigt sich mit geklammerten Ausdrücken.

Vereinfachend sollen zunächst nur Ausdrücke mit bis zu drei Klammerebenen betrachtet werden:



zugehörige Grammatik (aus dem Automaten konstruiert):

$$\begin{aligned}
 T &= \{ (,) \} \\
 N &= \{ \langle a_0 \rangle, \langle a_1 \rangle, \langle a_2 \rangle, \langle a_3 \rangle, \langle z_3 \rangle, \langle z_2 \rangle, \langle z_1 \rangle \} \\
 P &= \{ \\
 &\langle a_0 \rangle \quad \rightarrow \quad (\langle a_1 \rangle \\
 &\langle a_1 \rangle \quad \rightarrow \quad (\langle a_2 \rangle \quad | \quad \langle z_1 \rangle \\
 &\langle a_2 \rangle \quad \rightarrow \quad (\langle a_3 \rangle \quad | \quad \langle z_2 \rangle \\
 &\langle a_3 \rangle \quad \rightarrow \quad) \langle z_3 \rangle \\
 &\langle z_3 \rangle \quad \rightarrow \quad) \langle z_2 \rangle \\
 &\langle z_2 \rangle \quad \rightarrow \quad) \langle z_1 \rangle \\
 &\langle z_1 \rangle \quad \rightarrow \quad \varepsilon \\
 &\} \\
 s &= \langle a_0 \rangle
 \end{aligned}$$

Für eine höhere Klammerungstiefe muss man den Automaten bzw. die Grammatik lediglich über a_3 bzw. z_3 heraus analog fortsetzen. Dann wäre der Automat aber nicht mehr endlich!

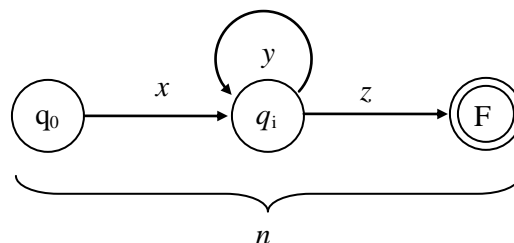
Eine Grammatik zur Beschreibung der beliebig tief geklammerten Ausdrücke wäre z.B.

$$\begin{aligned}
 T &= \{ (,) \} \\
 N &= \{ \langle \text{Ausdruck} \rangle \} \\
 P &= \{ \\
 &\langle \text{Ausdruck} \rangle \quad \rightarrow \quad (\langle \text{Ausdruck} \rangle) \quad | \quad () \\
 &\} \\
 s &= \langle \text{Ausdruck} \rangle
 \end{aligned}$$

Diese Grammatik ist aber nicht regulär. Man kann sich überlegen, dass es tatsächlich keine reguläre Grammatik geben kann, welche diese Ausdrücke beschreibt.

Der Beweis benutzt das Pumpig-Lemma, welches besagt, dass für gewisse Wörter w , die lang genug sind (Länge n), und von einem DEA erkannt werden folgendes gilt:

Es lässt sich in xyz zerlegen mit $|y| \geq 1$; $|xy| \leq n$ und es gilt: $xy^i z$ würde auch akzeptiert.



Wendet man dieses Lemma an, stellt man fest, dass obige Grammatik nicht regulär sein kann, da z.B. $((((()))$ aufgepumpt werden könnte zu $(((((()) () ()))))$.

5. Endliche Automaten und deren Implementierung

Dies ist vergleichsweise leicht (z.B. in Python) zu implementieren:

→ siehe das Programm "Automat_Zahltester.py"

II. Kommunikation und Synchronisation von Prozessen

1. Protokolle zur Kommunikation von Prozessen

Zur Kommunikation zweier Prozesse muss Einigkeit über folgende Dinge bestehen:

- Datenformat (festgelegt durch formale Sprache)
- Zeitlicher Ablauf des Kommunikationsvorgangs

Ein solches Regelwerk wird Protokoll genannt.

Beispiel 1:

SMTP Simple Mail Transfer Protocol) zum Versenden einer Email: LS S.47/48

Beispiel 2:

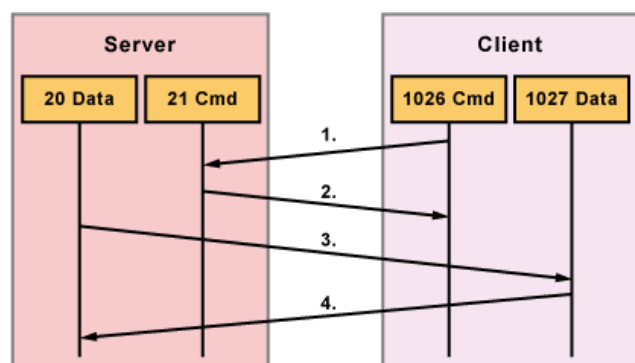
FTP (File Transfer Protocol) zur Übertragung von Dateien zwischen Rechnern (seit 1971)



Es werden 2 Verbindungen zwischen Server und Client aufgebaut: eine Steuerverbindung (für Kommandos) und eine Datenverbindung.

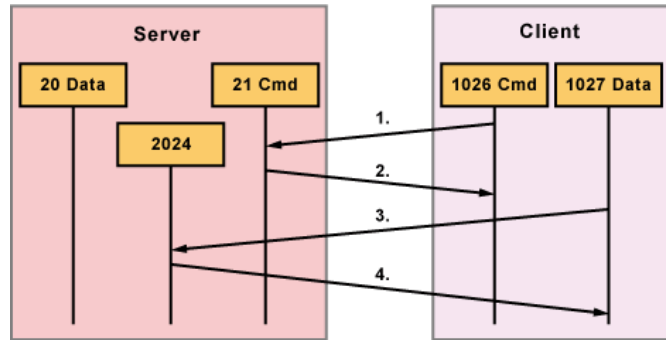
FTP Active Mode

In der Regel stellt der Client über den Steuerkanal eine Anfrage an den Server (– umgekehrt würde die Firewall des Clients blockieren). Er übermittelt ihm die Portnummer, auf der die Daten übertragen werden sollen. Der Server bestätigt und öffnet einen Datenkanal (sofern keine Firewall dies blockiert). Die Verbindung ist hergestellt.



FTP Passive Mode

Sitzt der Client hinter einer Firewall, so nennt der Server dem Client eine Portnummer, über die der Kontakt herzustellen ist. Nun initiiert der Client die Datenverbindung (weshalb sie von der Firewall nicht blockiert wird) und der Datenkanal kann geöffnet werden.



Beispiel (vgl. auch <http://www.elektronik-kompodium.de/sites/net/0902241.htm>)

```
1: ~ Verbinde...
2: ~ Verbunden mit 212.227.84.222, warte auf Antwort...
3: < 220 FTP Server ready.
4: > USER beispiel
5: < 331 Password required for beispiel.
6: > PASS *****
7: < 230 User beispiel logged in.
8: > REST 1
9: < 350 Restarting at 1. Send STORE or RETRIEVE to initiate transfer
10: > REST 0
11: < 350 Restarting at 0. Send STORE or RETRIEVE to initiate transfer
12: > SYST
13: < 215 UNIX Type: L8
14: > PWD
15: < 257 "/" is current directory.
16: ~ Login erfolgreich.
17: > PORT 192,168,168,12,4,182
18: < 200 PORT command successful
```

1. Verbindungsmeldung vom FTP-Client.
2. Verbindungsmeldung vom FTP-Client.
3. Der FTP-Server meldet mit dem Status-Code 220, dass die FTP-Verbindung hergestellt ist.
4. Der FTP-Client teilt dem FTP-Server mit, welcher Benutzer sich anmelden will.
5. Der FTP-Server fordert von diesem Benutzer das Passwort an.
6. Der FTP-Client schickt dem FTP-Server das Passwort.
7. Ist das Passwort richtig, meldet der FTP-Server dem Client den erfolgten Login (Status-Code 230).
8. Der FTP-Client teilt dem FTP-Server mit, dass er den Datentransfer neu starten soll.
9. Der FTP-Server meldet den Status-Code 350 zurück und erwartet weitere Angaben zur Ausführung des Befehls.
10. Der FTP-Client teilt dem FTP-Server mit, dass er den Datentransfer neu starten soll.
11. Der FTP-Server meldet den Status-Code 350 zurück und erwartet weitere Angaben zur Ausführung des Befehls.
12. Mit dem Befehl SYST fragt der FTP-Client nach dem Betriebssystem auf dem der FTP-Server läuft.
13. Der FTP-Server meldet das Betriebssystem zurück (Status-Code 215). In diesem Fall handelt es sich um ein UNIX-Betriebssystem.
14. Mit dem Befehl PWD fordert der FTP-Client das aktuelle Verzeichnis an.
15. Der FTP-Server meldet das aktuelle Verzeichnis (Status-Code 257). In diesem Fall lautet es "/".
16. Verbindungsmeldung des FTP-Client, dass der Login erfolgreich war.
17. Mit dem Befehl PORT teilt der Client dem Server mit, dass er den zweiten Port für den Datenkanal verwenden möchte.
18. Der FTP-Server meldet die erfolgreiche Ausführung des Kommandos (Status-Code 200).

Weitere Protokolle:

Schicht	Dienste / Protokolle / Anwendungen			
Anwendung	FTP	HTTP	DNS	SMTP
Transport	TCP		UDP	
Internet	IP (IPv4 / IPv6)			
Netzzugang	Ethernet, ...			

Ports:

Port	Dienst
20	ftp data
21	ftp control
25	Smtp
80	http
110	Pop3
443	https
3306	Mysql-Server

Das ISO-OSI-Schichtenmodell (Übersicht, vgl. auch Kapitel 5):

OSI-Architektur		TCP/IP-basierte Architektur		
Anwendung	Netzwerk- betriebs- system	Internet-Anwendungsprotokolle		
Darstellung		• Electronic Mail (Simple Mail Transfer Protocol, SMTP)		
Sitzung		• Datenübertragung (File Transfer Protocol, FTP)		
		• Telnet		
		• Hypertext (Hypertext Transfer Protocol, HTTP)		
Transport	Transmission Control Protocol, TCP User Datagram Protocol, UDP			
Vermittlung	Internet Control Message Protocol, ICMP	Internet Protocol, IP	Adress Resolution Protocol, ARP	
Sicherung	z.B. X.25, ATM, Frame Relay, ISDN			
Bitübertragung				

2. Nebenläufigkeit

Nebenläufige Prozesse können gleichzeitig ausgeführt werden. In der Regel muss man sie synchronisieren, wenn sie auf gemeinsame Ressourcen zugreifen. Dabei können Verklemmungen auftreten, die vermieden (generell ausgeschlossen) und verhindert (vorsichtige Zuteilung von Ressourcen) werden können.

Nebenläufige Prozesse:

- Sie können parallel auf verschiedenen Prozessoren ausgeführt werden, oder
- „pseudo-parallel“ auf einem Prozessor.

Prozess vs. Thread:

- Prozess: getrennte Adressräume, geschützt gegen andere Prozesse
- Thread („leichtgewichtiger Prozess“): gemeinsamer Adressraum, teilen geöffnete Dateien und andere Ressourcen

Grundlegende Fragestellung bei Nebenläufigkeit:

- Zugriff auf gemeinsame Ressourcen
- Koordination, Kommunikation, Synchronisation (zeitl. Abstimmung)

Bei Zugriff auf gemeinsame Ressourcen muss wechselseitiger Ausschluss sichergestellt werden – zum Beispiel beim gemeinsam verwendeten Speicher zweier Threads.

Beispiel:

```
def eingabe():  
    x=input(„Zahl eingeben“)  
    print(x)
```

Angenommen, zwei Threads T1 und T2 führen (parallel) diesen Code aus. Die Steuerung obliegt dem Betriebssystem.

- T1 liest das Zeichen „9“ und wird dann unterbrochen
- T2 liest „2“ und wird dann unterbrochen.
- T1 wird mit der Ausgabe fortgesetzt und schreibt „2“ auf den Bildschirm
- Grund: gemeinsamer genutzter Speicher.

Beispiele für typische Probleme:

Wechselseitiger Ausschluss:

Nur genau ein einziger Prozess darf sich im sogenannten kritischen Abschnitt befinden. Hierbei handelt es sich um den Teil des Programms, der auf die kritische Ressource zugreift.

Anforderungen:

- a) Höchstens ein Prozess im kritischen Abschnitt
- b) Jeder Prozess ist nur endliche Zeit im kritischen Abschnitt
- c) Wartezeit bis zum Eintritt in den kritischen Abschnitt ist endlich
- d) Ist kein Prozess im kritischen Abschnitt, kann ein sich interessierender Prozess sofort eintreten
- e) Alles funktioniert unabhängig von der relativen Ausführungsgeschwindigkeit der Prozesse.

Deadlock / Verklemmung:

Situation:

A benötigt eine Ressource, die momentan von B gehalten wird. B seinerseits benötigt eine Ressource, die von A gehalten wird.

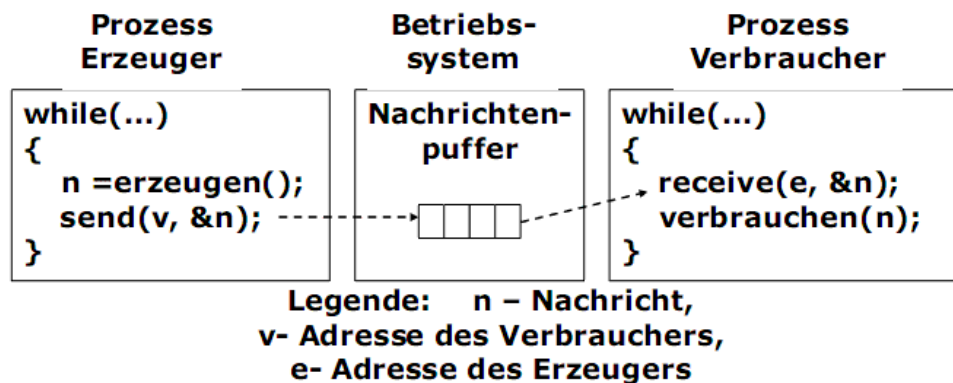
Folge:

A und B warten (ewig) bis die Ressource freigegeben wird.

Bedingungen für das Eintreten eines Deadlocks:

- Ressourcen können nur im wechselseitigen Ausschluss belegt werden
- Prozesse können mehrere Ressourcen anfordern
- Belegte Ressourcen sind ununterbrechbar zugeteilt.
- Es gibt mehrere Prozesse, die aus Ressourcen warten, die jeweils von anderen bereits belegt sind.

Erzeuger-Verbraucher-System



Probleme:

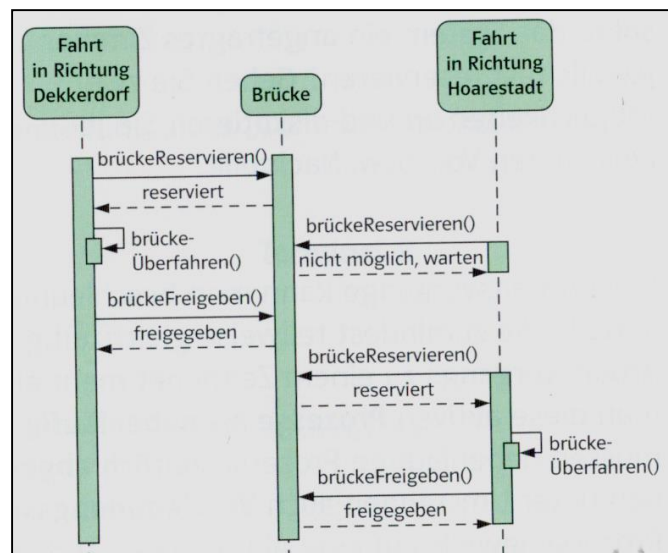
- Puffer zu klein bzw. Puffer wird zu langsam geleert: Ankommende Nachrichten müssen verworfen werden
- Puffer wird zu langsam gefüllt: Verbraucher müssen Wartezyklen einlegen

Ununterbrechbare Ressourcen

Beispiel Brücke

Ablauf:

- Ressource anfordern
- Benutzung der Ressource
- Ressource freigeben



3. Implementierung

Versuch 1:

- 2 Prozesse konkurrieren um eine Ressource
- Beide Prozesse können auf eine gemeinsame Variable *turn* zugreifen
- *turn* ist mit einem beliebigen Wert initialisiert (z.B. 0)

```
/* Prozess 0 */
wiederhole
{
    solange (turn ≠ 0) (*)
        tue nichts;
    /* kritischer Abschnitt */
    turn := 1;
    /* nichtkrit. Abschnitt */
}
```

```
/* Prozess 1 */
wiederhole
{
    solange (turn ≠ 1)
        tue nichts;
    /* kritischer Abschnitt */
    turn := 0; (**)
    /* nichtkrit. Abschnitt */
}
```

- Wechselseitiger Ausschluss ist garantiert.
- Nur abwechselnder Zugriff auf den kritischen Bereich.
- Busy waiting (aktives Warten) verschwendet Rechenzeit.

Versuch 2:

- Prozess 0 schreibt auf *flag[0]*, liest beide
- Prozess 1 schreibt auf *flag[1]*, liest beide
- Bedeutung von *flag[i] = true*: Prozess *i* will in den kritischen Bereich
- Initialisierung: *flag[0] := false* ; *flag[1] := false*

```
/* Prozess 0 */
wiederhole
{
    flag[0] := true; (*)
    solange (flag[1] = true)
        tue nichts;
    (**)
    /* kritischer Abschnitt */
    flag[0] := false;
    /* nichtkrit. Abschnitt */
}
```

```
/* Prozess 1 */
wiederhole
{
    flag[1] := true;
    solange (flag[0] = true)
        tue nichts;

    /* kritischer Abschnitt */
    flag[1] := false;
    /* nichtkrit. Abschnitt */
}
```

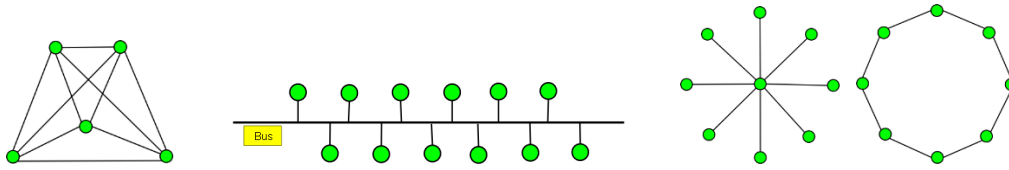
- Wechselseitiger Ausschluss ist garantiert.
- Auch nicht-alternierender Zugriff auf den kritischen Bereich.
- Busy waiting (aktives Warten) verschwendet Rechenzeit.

- Achtung: Deadlock ist möglich:
 - `flag[0] = flag[1] = false`
 - Prozess 0 setzt `flag[0] := true` und gibt CPU ab
 - Prozess 1 setzt `flag[1] := true`
 - Jetzt werden beide Prozesse ihre Schleife
„solange (`flag[i] ≠ true`) tue nichts;“ nie verlassen!

Stichworte: Monitorkonzept, Semaphore

Siehe Klett S.56 – 62

4. Topologie von Rechnernetzen



„Jeder mit Jedem“

Busnetz:

- Zugriffsverfahren CSMA-CD (Carrier Sense Multiple Access / Collision Detection)
Ein Sender muss prüfen, ob bereits ein Anderer Daten überträgt. Erst wenn der Bus frei ist, wird gesendet – und zwar automatisch an alle.
Über eine eindeutige MAC-Adresse (z.B. 10:A8:AF:01:CF:F1) weiß der Empfänger, dass ein Datenpaket für ihn bestimmt ist.

Sternnetz:

- Jeder Rechner ist direkt mit einem Zentralrechner verbunden.

Ringnetz (Token-Ring):

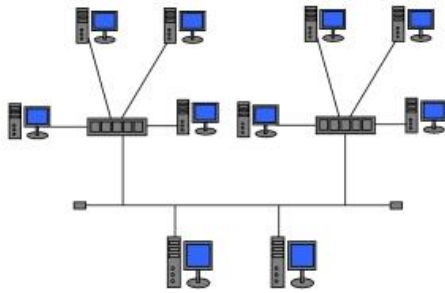
- Die Rechner sind ringförmig miteinander verbunden. Ein bestimmtes Datenpaket (Token) kreist im Ring. Nur der Rechner, der das Token hat, darf senden.

Kriterien zur Bewertung von Rechnernetzen:

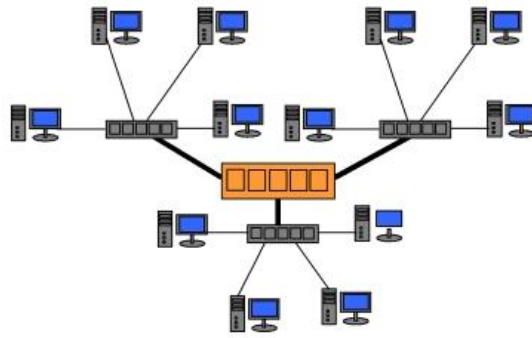
1. Störanfälligkeit
2. Systempflege
3. Kosten
4. Übertragungsrate
5. Sicherheit

	Bus	Stern	Ring
Störanfälligkeit	[+] Geräteausfall hat keine Auswirkungen [–] Ausfall des Hauptmediums (Kabelbruch) blockiert alles	[+] Endgeräteausfall hat keine Auswirkungen [–] Ausfall des Knotens legt Netz lahm	[+] bei Unterbrechung Umkehr der Laufrichtung des Tokens möglich
Kosten, Erweiterbarkeit	[+] geringe Kosten, keine aktiven Komponenten, leicht erweiterbar	[+] geringe Kosten wegen geringer Kabelmengen, leicht erweiterbar, wenige aktive Komponenten	[–] teure Komponenten
Sicherheit	[–] Abhören leicht möglich		[–] leicht abhörbar
Übertragungsrate	[–] Sinken der Übertragungsrate durch Kollisionen bei hoher Netzauslastung	[–] Bei vielen angeschlossenen Rechnern niedrige Übertragungsrate	[+] hohe rate, da garantiert nur einer sendet (wg. Token)
Beispiele	Kleine Netzwerke (Ethernet)	Computerräume, Heimnetze	früher: Uni-Netze

Kombinationen:

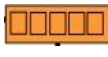


Stern-Bus



Stern – Stern

Je nach den verwendeten Komponenten verhalten sich obige Netze anders:

Beim „Verteiler“  kann es sich um einen Hub oder Switch handeln:

Hub:

Ein Hub ist eine nicht-intelligente Verteilungs- bzw. Verstärkereinheit. Er gibt das Eingangssignal inhaltlich unverändert an alle angeschlossenen Einheiten weiter. Sendet einer der angeschlossenen rechner, können also die anderen nicht senden („busartig“)

Switch:

Ein Switch wertet die (Ziel-)Adressen der Datenpakete aus und leitet sie an das richtige angeschlossene Gerät weiter. Es kann eine 1:1-(Point-to-Point)-Verbindung zwischen zwei Rechnern aufgebaut werden. Eine Kollision kann lediglich auftreten, wenn zwei Sender (gleichzeitig) an denselben Empfänger übertragen möchten.

6. Kommunikation über Protokollschichten

Blackbox-Sicht:

Man beauftragt ein Transportunternehmen und übergibt

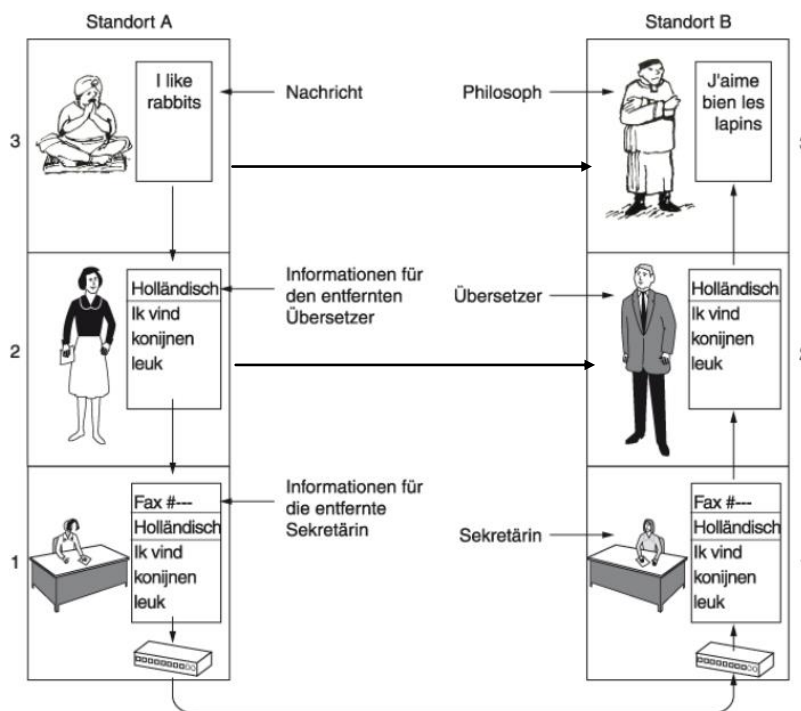
- Absender
- Empfänger
- 5 Paket

Der Kunde nimmt einen Dienst des Transportunternehmens in Anspruch, und teilt ihm (Schnittstelle) die relevanten Daten mit. Dabei ist es für den Kunden irrelevant, welchen Weg genau die Pakete nehmen, wie der Fahrer heißt bzw. ob die Waren im selben Fahrzeug reisen.

Der Unternehmer seinerseits nutzt u.U. auf einer Teilstrecke einen Dienst der Bahn und lässt die Sendung im Nachtzug transportieren. In welchem Waggon das Paket ist, interessiert ihn u.U. nicht.

Dienst und Schnittstelle:

Jede Schicht bietet der nächst höher gelegenen Dienste an. Wie diese Dienste genau abgearbeitet werden, ist für die obere Schicht nicht von Belang. Die Grenze zwischen benachbarten Schichten wird Schnittstelle genannt.



In gewisser Weise wird hier zwischen den Schichten 3 – 3 bzw. 2 – 2 kommuniziert. Die Philosophen müssen nicht wissen, wie ein Faxgerät funktioniert.

Zwischen gleichartigen Schichten (z.B. 3–3) geschieht die Kommunikation über ein festgelegtes Protokoll. Transporteinheit sind dabei gewisse Pakete.

3: Nachrichteninhalte in eigener Sprache, persönliche Anrede

2: Holländische Sprache, formale Anrede

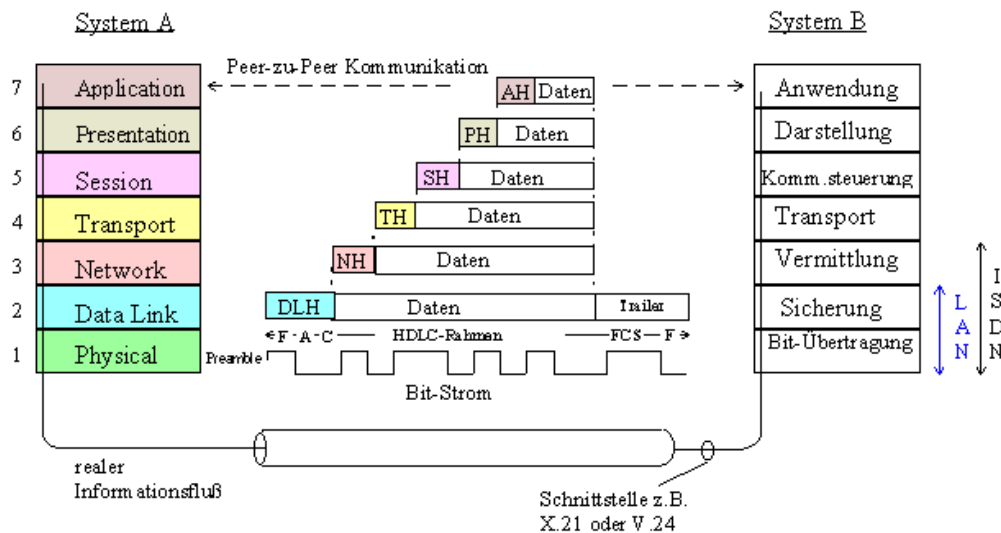
1: FAXprotokoll, ggf. Bits/Bytes, Telefonnummer/IP-Adresse.

Softwareschichten – Zweischichtarchitektur

- Clientschicht
- Serverschicht

Softwareschichten – Dreischichtarchitektur

- Präsentationsschicht
- Logikschicht
- Datenerhaltungsschicht



Schicht 1: Bitübertragung

wichtig sind Kabeleigenschaften, Übersprechen, MAC-Adresse des Endgeräts

Schicht 2: Sicherungsschicht:

Fehlerkorrektur durch Prüfsummen, etc.

Schicht 3: Vermittlungsschicht

IP-Adressen dienen zur Wegfindung (Routing). Geht ein Datenpaket verloren, so wird dies nicht erkannt.

Schicht 4: Transportschicht

Sicherung des Datenaustausch; Es wird geprüft, ob alle gewünschten Pakete angekommen sind.

Protokolle: TCP bzw. UDP

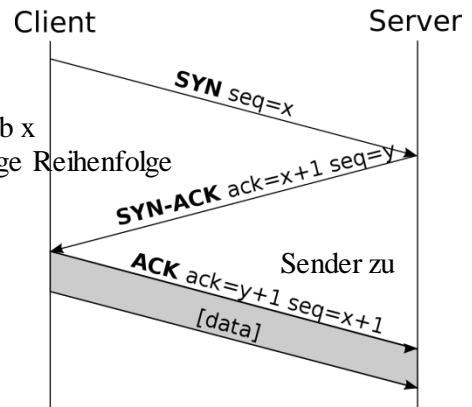
TCP stellt eine Verbindung zwischen Endpunkten einer Netzverbindung (Sockets) her.

UDP ist nicht verbindungsorientiert.

Aufbau einer TCP-Verbindung:

Bei x bzw. y handelt es sich um (zunächst) zufällig generierte Sequenznummern. Die gesendeten Pakete werden aufsteigend ab x durchnummeriert, sodass sie beim Empfänger ggf. in die richtige Reihenfolge gebracht werden können.

Dies ist wichtig, da IP-Pakete unterschiedlich lange Wege von Empfänger zurücklegen können.



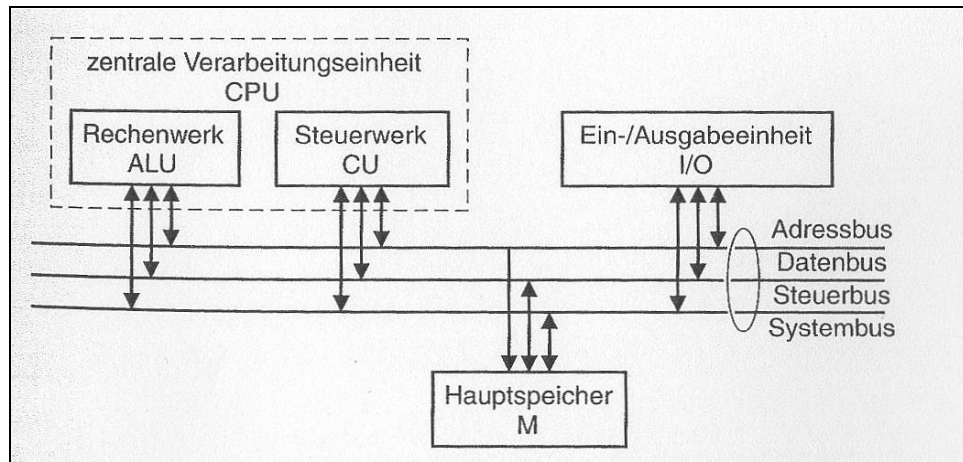
7. Internet – Das Netz der Netze

LS S.71ff

III. Funktionsweise eines Rechners

1. Von-Neumann-Rechner

Struktur:



Das von-Neumann-Rechnerkonzept stellt (ca. seit 1946) das grundlegende Operationsprinzip von Computern dar, wie es heute, wenn auch mit einigen Abwandlungen, noch in fast allen modernen Computern genutzt wird. Es basiert auf folgenden Punkten:

1. Hauptkomponenten des Rechners sind Rechenwerk, Steuerwerk, Hauptspeicher, Eingabe-/Ausgabeeinheit und die Verbindungen (Datenwege, Busse). Steuerwerk und Rechenwerk bilden zusammen die zentrale Recheneinheit (Prozessor, CPU).
2. Die intern verwendete Signalmenge ist binär codiert. Es werden Worte fester Länge parallel verarbeitet. Die Verarbeitung erfolgt taktgesteuert. Der Rechner arbeitet nach einem Start automatisch.
3. Programmbefehle und Daten werden im einheitlichen Hauptspeicher ohne Kennzeichnung gespeichert. Der Hauptspeicher besteht aus fortlaufend adressierten Speicherworten, deren Inhalt nur über die Adresse angesprochen werden kann.
4. Der Rechner verarbeitet extern eingegebene Programme und Daten, die intern gespeichert werden, sequenziell. Die normale Verarbeitung erfolgt in der Reihenfolge der Abspeicherung der Programmbefehle. Die sequenzielle Abarbeitung kann durch Sprungbefehle oder datenbedingte Verzweigungen unterbrochen werden.

(nach "Taschenbuch der Informatik", ...)

Befehlszyklus des von-Neumann-Rechners:

1. Befehl aus dem Hauptspeicher holen und im Steuerwerk interpretieren.
2. Operand aus dem Hauptspeicher holen und Befehl ausführen (und Resultat in den Hauptspeicher schreiben).

Engpässe (Flaschenhalse) der von-Neumann-Architektur:

1. Befehle und Daten stehen gleichzeitig nebeneinander im Hauptspeicher und werden über den gleichen Weg gelesen bzw. geschrieben (Speicherschnittstelle) und behindern sich damit gegenseitig.
Ausweg: Getrennte Speicher und Busse für Befehle und Daten (getrennte Caches, Harvard-Architektur).
2. Sequenzielle Abarbeitung der einzelnen Programmschritte in jeweils zwei Phasen. Keine Möglichkeit der gleichzeitigen Ausführung mehrerer Schritte.
Ausweg: Nutzung von Parallelität: ILP (Instruction Level Parallelism), MT (Multithreading) und MP (Multiprocessor).
3. Feste Befehlsabfolge und Steuerung in nur einer Ebene (nur ein Befehlszähler, ein Befehlsregister) bereitet Probleme bei Unterprogrammtechnik, Multitasking, Multiuserbetrieb.
Ausweg: Einführung eines Universalregistersatzes (register renaming, register windows) und vereinfachte Steuerungsabläufe für die Befehlsausführung (RISC-Architekturen).

2. Registermaschine

Moderne Rechner basieren nicht vollständig auf dem von-Neumann-Konzept.

Aktuelle Mikroprozessoren verwenden das Modell der Registermaschine. Die dabei verwendeten Prozessoren beinhalten sehr schnell zugängliche Speicherzellen auf dem Chip selber (im Ggs. zum Hauptspeicher). Solche Zellen heißen Register und kommen in verschiedenen Ausprägungen vor.

Registermaschinen

Eine Registermaschine besteht aus einem Arbeitsspeicher für die Programme sowie einer Reihe von Registern. Die Register haben folgende Aufgaben:

1. Der Befehlszähler *BZ* enthält die Speicheradresse des nächsten zu bearbeitenden Befehls.
2. Das Statusregister *SR* enthält Informationen über das Ergebnis der letzten (Rechen-) Operation.
3. Die Datenregister *A*, *R*₁, *R*₂, ... dienen zur Ablage von Daten. Dabei spielt das Datenregister *A* (Akkumulator) eine besondere Rolle: Es enthält einen Eingabewert für den folgenden Rechenbefehl und nimmt das Rechenergebnis auf.

Die Registermaschine im Ordner "seRMS" verwendet denselben Befehlssatz wie in LS S.101 und ist auf allen Rechnern lauffähig, die JAVA unterstützen.

Achtung:

Der Einfachheit halber wird bei dieser Maschine davon ausgegangen, dass alle Daten in den Registern stehen. Ein Arbeitsspeicher ist also in diesem speziellen Fall und im Gegensatz zur "Realität" nicht vorhanden.

Einfache Assemblersprache:

Befehl	Auswirkung
DLOAD i	lädt unmittelbar die Zahl i in A, erhöht den Wert in BZ um 1
LOAD x	kopiert den Wert in Rx nach A, erhöht den Wert in BZ um 1
STORE x	kopiert den Wert in A nach Rx, erhöht den Wert in BZ um 1
ADD x	addiert den Wert in Rx zum Wert in A, legt das Ergebnis in A ab, erhöht den Wert in BZ um 1
SUB x	subtrahiert den Wert in Rx vom Wert in A, legt das Ergebnis in A ab, erhöht den Wert in BZ um 1
MULT x	multipliziert den Wert in Rx mit dem Wert in A, legt das Ergebnis in A ab, erhöht den Wert in BZ um 1
DIV x	dividiert den Wert in A durch den Wert in Rx (Ganzzahldivision ohne Rest), legt das Ergebnis in A ab, erhöht den Wert in BZ um 1
JUMP n	lädt die Zahl n in BZ, Programm wird mit Befehl in Speicherzelle n fortgesetzt (unbedingter Sprung)
JGE n	lädt die Zahl n in BZ, falls der Wert in A größer oder gleich 0 ist, erhöht ansonsten den Wert in BZ um 1 (bedingter Sprung)
JGT n JLE n JLT n JEQ n JNE n	entsprechende bedingte Sprünge für die Fälle "größer als 0" "kleiner oder gleich 0" "kleiner als 0" "gleich 0" "ungleich 0"
END	erhöht den Wert in BZ um 1 und beendet den Programmablauf

Begriff (Assembler):

- Ein Assembler ist ein Programm, das ein (z.B. in obiger) in Assemblersprache verfasstes Programm in Maschinencode (Bitfolgencodes) übersetzt.
- Umgangssprachlich meint man mit Assembler aber manchmal auch die Sprache selber.

Beispiel:

The image displays two side-by-side screenshots from a software application. The left screenshot shows a text editor window titled 'Kap_III_LE_2_Sitzplaetze - Editor'. It contains a list of assembly instructions with comments: 1: DLOAD 8 --Beispiel n = 8; 2: STORE 1 --Produkt p in R1; 3: STORE 2 --Zählvar. m in R2; 4: DLOAD 1; 5: STORE 3 --konstante 1 in R3; 6: LOAD 2; 7: SUB 3; 8: JLE 13 --falls m <= 0; 9: STORE 2 --m = m - 1; 10: MULT 1; 11: STORE 1 --p = p*m; 12: JUMP 6; 13: LOAD 1 --Ergebnis in A; 14: END. The right screenshot shows a window titled 'sehr einfache Registermaschinen-Simulation'. It displays a grid of 16 registers (R0 to R15) and two special registers, A and BZ. Register A contains the value 5, and BZ contains 8. Register R1 contains 336, R2 contains 6, and R3 contains 1. All other registers are 0. Below the registers are three buttons: 'Anfang' (red), 'Schritt' (yellow), and 'Durchlauf' (green). The window also shows the same assembly instructions as the editor, with some lines highlighted in red.

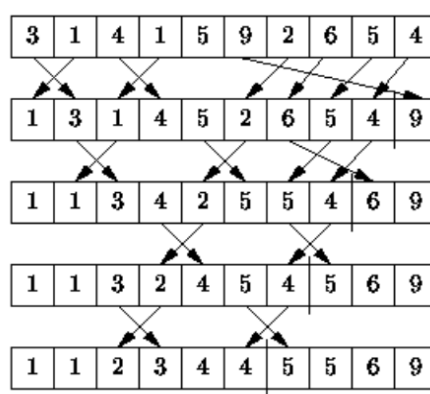
IV. Grenzen der Berechenbarkeit

1. Laufzeit von Algorithmen

Vergleich von Sortieralgorithmen

Bubblesort:

- Vergleiche von links nach rechts jeweils zwei Nachbarelemente und vertausche deren Inhalt, falls sie in der falschen Reihenfolge stehen.
- Wiederhole dies, bis alle Elemente richtig sortiert sind.
- Analogie: die kleinsten Elemente steigen wie Luftblasen zu ihrer richtigen Position auf (je nachdem, ob man aufsteigend oder absteigend sortiert)



```
def bubblesort(a):  
    for k in range(0, len(a)-1):  
        for i in range(0, len(a)-1):  
            if a[i] > a[i+1]:  
                a[i], a[i+1] = a[i+1], a[i]
```

Vertauschen zweier Variablen

Korrektheitsbeweis und Bemerkungen:

1. Nach dem ersten Durchlauf befindet sich das größte Element an der richtigen Stelle (hier ganz hinten)
2. Nach dem zweiten Durchlauf gilt dies für das zweigrößte Element, usw.
3. Nach $N-1$ Durchgängen ist das Array sortiert.
4. Da bei jedem Durchlauf auch andere Elemente ihre Position verbessern, ist das Array oft auch schon vor dem $N-1$. Durchlauf sortiert.
5. Verbesserung: Test auf vorzeitiges Ende.

```
def bubblesort(a):  
    for k in range(0, len(a)-1):  
        sortiert = True  
        for i in range(0, len(a)-1):  
            if a[i] > a[i+1]:  
                a[i], a[i+1] = a[i+1], a[i]  
                sortiert = False  
        if sortiert:  
            break
```

➔ vgl. http://cgvr.cs.uni-bremen.de/teaching/info2_11/folien/10_sortieren-2.pdf

Laufzeitbetrachtung:

Worst Case: das kleinste Element steht ganz hinten bzw. die Liste ist „verkehrt herum“ sortiert.

0															14	
S	O	R	T	I	E	R	B	E	I	S	P	I	E	L		Original-Array
O	R	S	I	E	R	B	E	I	S	P	I	E	L	T		nach 1. BubbleUp
O	R	I	E	R	B	E	I	S	P	I	E	L	S	T		nach 2. BubbleUp
O	I	E	R	B	E	I	R	P	I	E	L	S	S	T		
I	E	O	B	E	I	R	P	I	E	L	R	S	S	T		
E	I	B	E	I	O	P	I	E	L	R	R	S	S	T		... etc. ...
E	B	E	I	I	O	I	E	L	P	R	R	S	S	T		
B	E	E	I	I	I	E	L	O	P	R	R	S	S	T		
B	E	E	I	I	E	I	L	O	P	R	R	S	S	T		
B	E	E	I	E	I	I	L	O	P	R	R	S	S	T		nach 10. BubbleUp
B	E	E	E	I	I	I	L	O	P	R	R	S	S	T		Sortiert !

Hier läuft die äußere Schleife 14 mal (bzw. 11 mal mit early exit), die innere jeweils 13,12,11,10, 9,8,7, ... mal.

Anzahl der Vergleiche/potentiellen Vertauschungen: $14+13+12+10+\dots+3+2+1$

Allgemein bei einer Listenlänge von n : $n+(n-1)+\dots+3+2+1 = \frac{n \cdot (n-1)}{2} = \frac{n^2 - n}{2} \rightarrow n^2$

Die Komplexität des Algorithmus im worst case ist also: $O(n^2)$

```
def bubblesort(a):
    for k in range(0, len(a)-1):
        for i in range(0, len(a)-1):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
```

$\left. \begin{array}{l} O(1) \\ O(n) \cdot O(1) = O(n) \end{array} \right\} O(n) \cdot O(n) = O(n^2)$

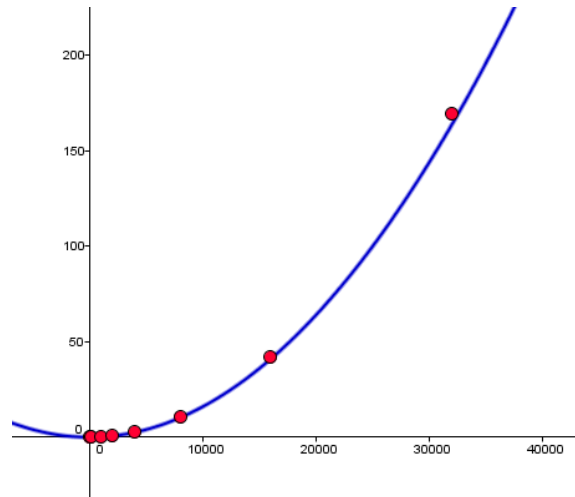
Best Case: die Liste ist schon sortiert.

Hier ist (bei der Variante mit „early exit“) nur ein Schleifendurchlauf nötig). Also: $O(n)$

average case: liegt irgendwo dazwischen. Ohne Beweis: $O(n^2)$

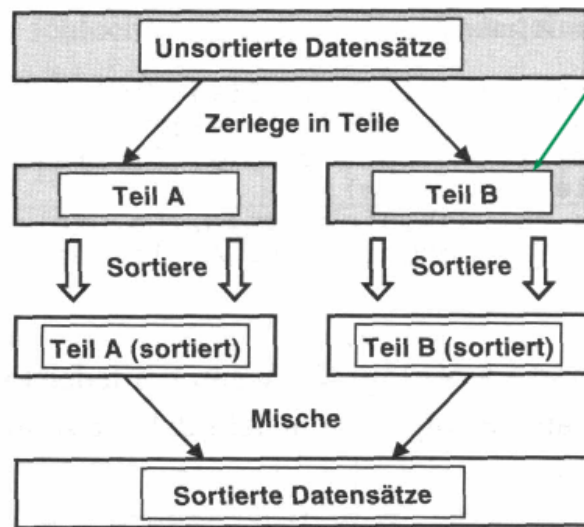
Laufzeitanalyse des in Bubblesortalgorithmus

Die dargestellte Funktion hat einen Term der Form $f(n) = k \cdot n^2$

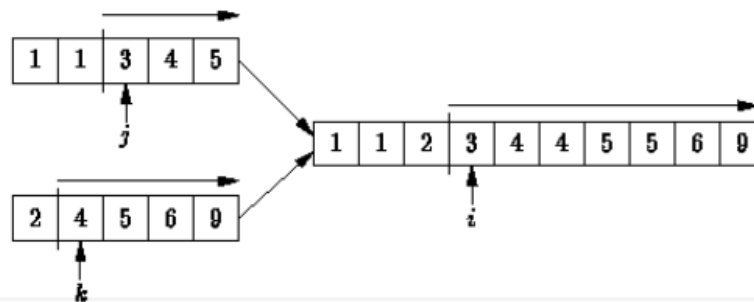
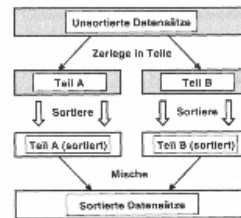


Mergesort

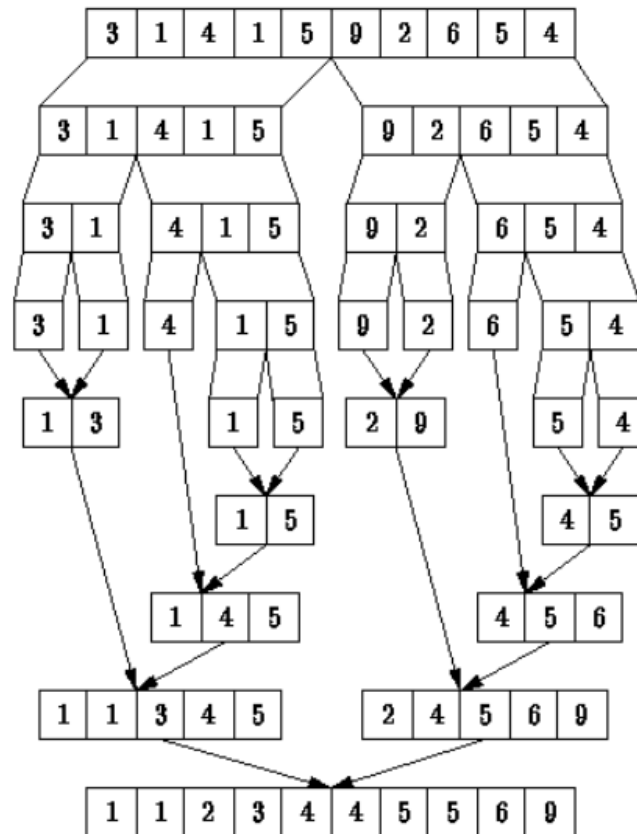
- Teile die ursprüngliche Menge an Daten in zwei Hälften
- Sortiere die beiden Teilmengen
- Mische die beiden sortierten Hälften wieder zusammen (engl. merge) – wähle dazu das kleinere der beiden Elemente, die an der jeweils ersten Stelle der beiden Datensätze stehen
- Wende das Verfahren rekursiv auf die beiden Hälften an, um diese zu sortieren.



rekursive Anwendung
des Algorithmus' auf
die Teile



Beispiel:



```
def mergesort(A):
    return rek_mergesort(A, 0, len(A)-1)
```

```
def rek_mergesort(A, l, r):
    if r <= l:
        return
    mid = (l + r) // 2
    A1 = rek_mergesort(A, l, mid)
    A2 = rek_mergesort(A, mid+1, r)
    return merge(A1, A2)
```

$O(\log(n))$

```
def merge(a, b):
    if len(a) == 0: return b
    if len(b) == 0: return a
    result = []
    i=j=0
    while i<len(a) and j<len(b):
        if a[i] <= b[j]:
            result.append(a[i])
            i+= 1
        else:
            result.append(b[j])
            j+= 1
    while i<len(a):
        result.append(a[i])
        i+= 1
    while j<len(b):
        result.append(b[j])
        j+= 1
    return result
```

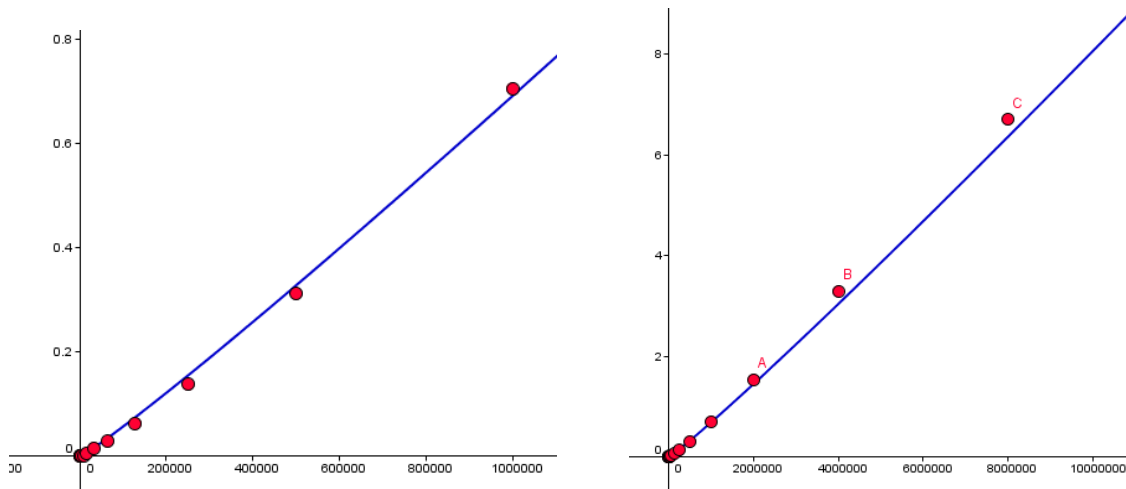
$O(n)$

Bemerkungen:

- Algorithmus ist sehr übersichtlich und einfach zu implementieren
- Aufwand: $O(n \cdot \log(n))$
- $\log(n)$ viele „Etagen“, Aufwand pro Etage in $O(n)$, gilt auch im *worst case*
- nicht besonders schnell, da viel umkopiert wird
- Optimierung:
 - ständiges Anlegen und Aufgeben von Hilfsarrays kostet Zeit.
 - besser ein großes Hilfsarray anlegen und immer wieder benutzen
- In-place-Sortierung (ohne Hilfsarray) möglich, aber sehr kompliziert

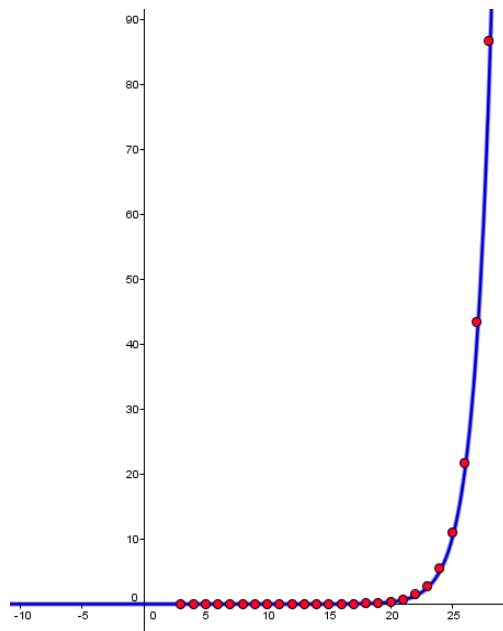
Laufzeitanalyse des in Python eingebauten Mergesortalgorithmus

Die dargestellte Funktion hat einen Term der Form $f(n) = k \cdot n \log(n)$



Beispiel: Türme von Hanoi

- Implementiere den Algorithmus zur Lösung des Problems „Türme von Hanoi“ in Python.
- Untersuche das Laufzeitverhalten in Abhängigkeit von der Anzahl n der Scheiben



siehe auch das Dokument "hanoi.doc".

Dort ergibt sich, dass für n Steine $2^n - 1$ Umsetzungen nötig sind.

Exkurs: Wie erkennt man Exponentialfunktionen?

- Verdopplungsintervalle sind gleich groß
- Quotient $\frac{f(x+a)}{f(x)}$ ist für festes a immer gleich groß
- halblogarithmische Auftragung liefert eine Gerade
→ Vgl. "halblogarithmisch.xls"

zurück zum Beispiel:

Für 60 Steine wären $1,15 \cdot 10^{18}$, bei 1000 schon $2 \cdot 10^{90}$ Züge nötig.

Würde man die 60er Variante spielen und pro Sekunde einen Stein umsetzen, dauerte das Spiel 36 MRD Jahre. Beachte auch, dass man die Zahl der Atome im Universum auf 10^{79} schätzt.

Beispiel: Fibonacci-Zahlen

Die Fibonacci-Folge beginnt so: 0,1,1,2,3,5,8,13,21,34, ...

Rekursive Definition:

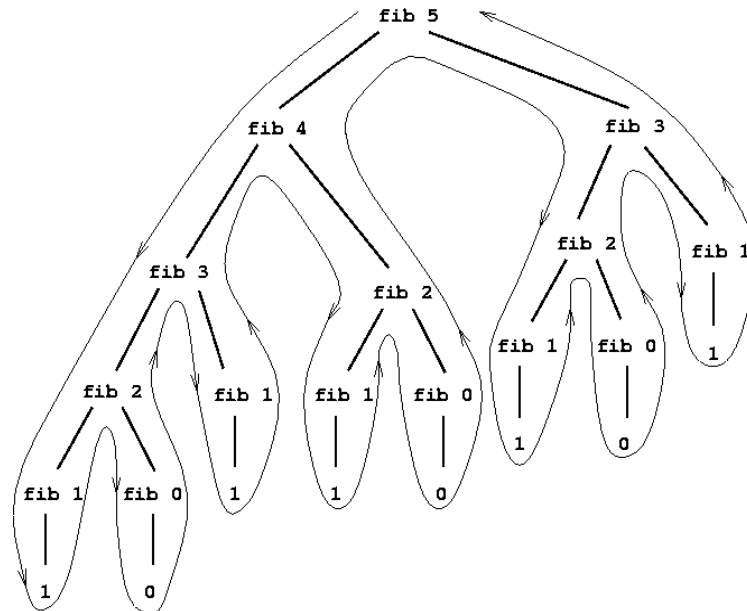
$$fib(n) = \begin{cases} 0 & \text{für } n=0 \\ 1 & \text{für } n=1 \\ fib(n-1) + fib(n-2) & \text{für } n \geq 2 \end{cases}$$

Umgesetzt in Python sieht das z.B. so aus:

```
def fib(n):  
    if n==0:  
        return 0  
    elif n==1:  
        return 1  
    else:  
        return fib(n-1)+fib(n-2)
```

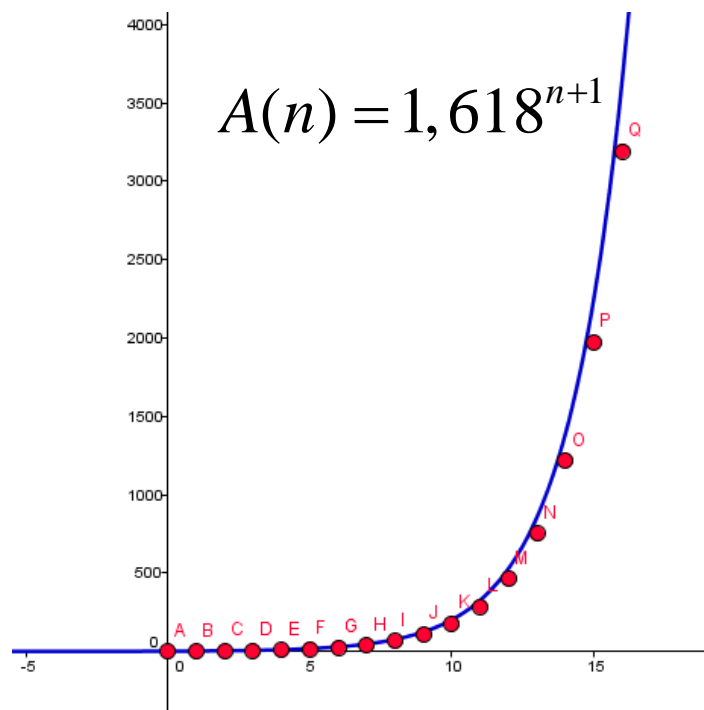
Laufzeitverhalten:

<i>n</i> -te Fibonacci-Zahl	Wert	rekursive Aufrufe	# <i>A</i> (<i>n</i>)	$\frac{A(n)}{A(n-1)}$
0	0	0	0	
1	1	0	0	
2	1	fib(1), fib(0)	2	
3	2	fib(2), fib(1)	2 + 2+0=4	2
4	3	fib(3), fib(2)	2 + 4+2=8	2
5	5	fib(4), fib(3)	2 + 8+4=14	1,75
6	8	fib(5), fib(4)	2 + 14+8=24	1,71
7	13	fib(6), fib(5)	2 + 24+14=40	1,67
8	21	fib(7), fib(6)	2 + 40+24=66	1,65
...
				?



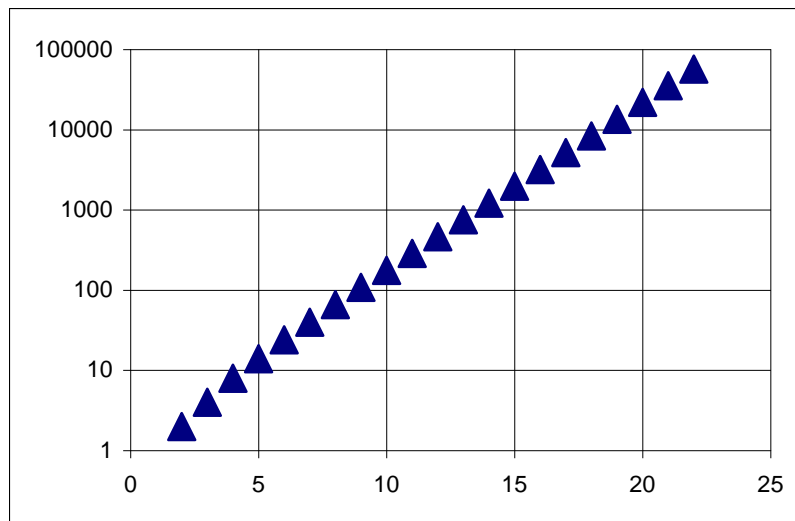
Bemerkungen:

- Die Zahl der rekursiven Aufrufe ist wieder so etwas Ähnliches wie eine Fibonacci-Folge.
- Der Algorithmus ist von exponentieller Ordnung.
- Das Verhältnis $\frac{A(n)}{A(n-1)}$ der Rekursionszahlen von einem Schritt zum Nächsten nähert sich 1,618...
Hierbei handelt es sich um die charakteristische Zahl des goldenen Schnitts.



→ Dateien: "fibonacci.py", "fibonacci_anpassung.ggb", "fibonacci_anzahl_reks.py"

Einfach logarithmische Auftragung (y-Achse ist log):



Erweiterungen:

Programmiere die *linear rekursive* Version der Fibonacci-Funktion (Buch S.132/6) und eine *iterative* Variante. Untersuche jeweils das Laufzeitverhalten.

→ Programmdateien: "fibonacci_iterativ.py", "fibonacci_linear.py"

weitere Beispiele:

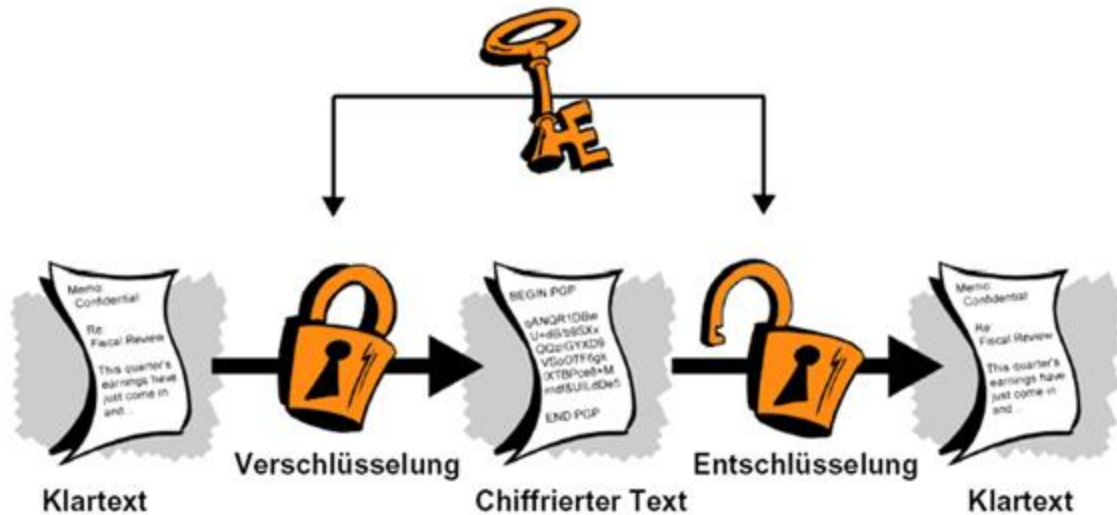
- Fakultät (Buch S. 130/1)
→ Programmdateien: "fakultaet.py"
- Binomialkoeffizient rekursiv
→ Programmdateien: "binomial_rekursiv.py"
- Binomialkoeffizient (Pascalsches Dreieck)
→ Programmdateien: "binomial_dreieck.py"
- Suchen in binären Bäumen

Merke:

rekursive Algorithmen können sehr schnell sein (Quicksort, ggT), aber im Vergleich zu iterativen Varianten aber auch sehr langsam (Binomialkoeffizienten, Fibonaccizahlen).

2. Laufzeitaufwand und Entschlüsselung

Symmetrische Verschlüsselung

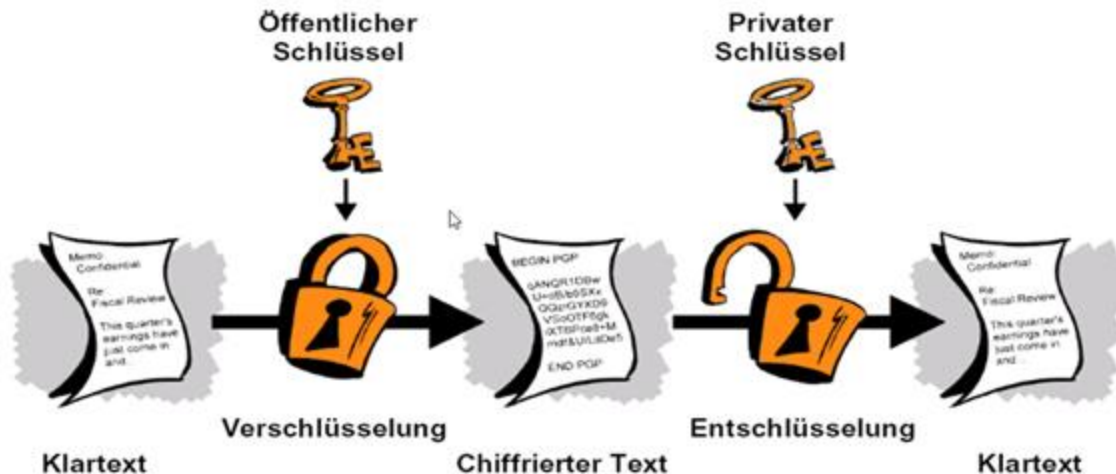


Zum Ver- und Entschlüsseln ist derselbe Schlüssel nötig. Dieser muss Sender und Empfänger bekannt sein und vor der Übermittlung ggf. ausgetauscht werden. Dies stellt einen u.U. unangemessenen Aufwand dar.

Beispiele:

DES, AES (Schlüssellänge: 128Bit bzw. 256Bit), IDEA, Blowfish, RC4

Asymmetrische Verschlüsselung



Sender und Empfänger besitzen Schlüsselpaare. Mit dem einen Schlüssel verschlüsselte Nachrichten können nur mit dem zugehörigen anderen Schlüssel entschlüsselt werden. I.A. sind die Schlüssellängen größer als bei der symmetrischen Verschlüsselung und somit ist das Verfahren langsamer als dort.

Beispiel: RSA

Spezialfall: Public Key

Ein öffentlicher Schlüssel wird verteilt. Jeder kann mit diesem Schlüssel Nachrichten verschlüsseln und versenden. Das Entschlüsseln ist ausschließlich mit dem privaten Schlüssel möglich. Dieser muss natürlich geheim gehalten werden.

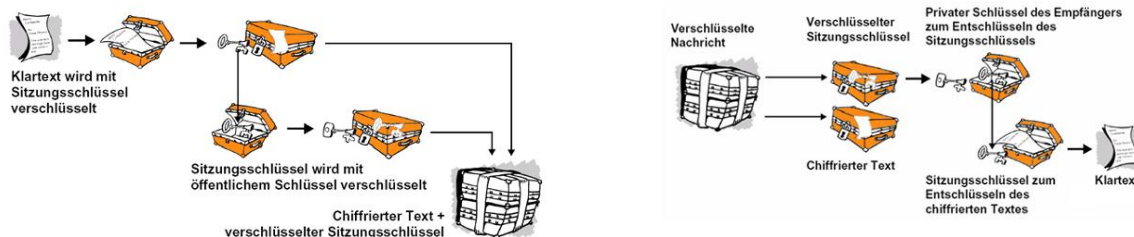
Natürlich muss der Absender sicherstellen, dass der ihm vorliegende öffentliche Schlüssel auch tatsächlich zum Absender gehört. Dies wird durch Zertifikate sichergestellt.

Anwendung:

Jemand möchte von verschiedenen Personen verschlüsselte Nachrichten erhalten.

Kombinationsverfahren (Hybridverfahren):

Die Kommunikation funktioniert über einen sog. Session-Key, der nur für ein einziges Mal verwendet wird. Dieser wird asymmetrisch mit dem öffentlichen Schlüssel des Empfängers verschlüsselt und versendet. Der Empfänger entschlüsselt diesen und verwendet ihn als Schlüssel für die anschließende symmetrische Verschlüsselung.



Beispiele: S/MIME, PGP

Mögliche Referate:

- RSA
- Hashfunktionen
- PGP
- WLAN-Verschlüsselung
- Digitale Signatur
- Zertifikate

Methoden zur (unerlaubten) Entschlüsselung und Auswege

- Brute-Force-Methode (systematisches Probieren)
Bemerkung: Es gibt $26! = 10^{26}$ Permutationen des Alphabets.
- Häufigkeitsanalyse
- Homophone Verschlüsselung (ein Klartextzeichen kann durch verschiedene Chiffrezeichen kodiert sein)
- Polyphone Verschlüsselung (mehrere Klartextzeichen können durch dasselbe Chiffrezeichen kodiert sein)

Unterlagen im Krypto_Ordner:

- Monoalphabetische_V.doc
- RSA.doc
- Ausstellung.doc
- Symm_asymm.doc
- Blinde_Kuh_Kryptologie_Seiten.doc

Beispiel zur Häufigkeitsanalyse:

**grrk skotk Ktzinkt yincosskt gal jiks Ykk, Quklvlinkt
ot jgy Cgyyxx, Yincgktfinkt ot jok Nukn.**

3. Prinzipielle Grenzen der Berechenbarkeit

Die Ulam-Funktion:

```
def ulam(n):  
    if n==1:  
        return 1  
    elif n/2==n//2:  
        return ulam(n/2)  
    else:  
        return ulam(3*n+1)
```

Beispiele:

21 → 64 → 32 → 16 → 8 → 4 → 2 → 1
20 → 10 → 5 → 16 → 8 → 4 → 2 → 1
19 → 58 → 29 → 88 → 44 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 (s.o.)
31 → 94 → 47 → 142 → 71 → 214 → 107 ... (98 weitere Schritte) ... → 4 → 2 → 1

Alle Startwerte, die Zweierpotenzen sind, lassen die Funktion terminieren („zum Ende kommen“).

Es ist aber bis heute unbewiesen, ob die Ulamfunktion für alle natürlichen Zahlen terminiert.

Halteproblem

Es wäre „schön“, wenn man einem Algorithmus bzw. einem Programm ansehen würde, ob er für eine bestimmte (oder alle) Zahlen terminiert. Noch besser wäre es, wenn man diese Aufgabe von einer Maschine (einem Programm) erledigen lassen könnte.

Diese Fragestellung bezeichnet man als *Halteproblem*.

Angenommen, es gäbe ein solches Superprogramm namens HALT. Diesem würde man den Code eines anderen Programms TESTPROG und einen Eingabewert EINGABE übergeben und es würde sagen „JA“ oder „NEIN“:

$$(I) \quad \text{HALT}(\text{TESTPROG}, \text{EINGABE}) = \begin{cases} \text{JA} & \text{wenn Testprog bei Eingabe terminiert} \\ \text{NEIN} & \text{wenn Testprog bei Eingabe nicht terminiert} \end{cases}$$

Beispiel:

HALT(ULAM, Zahl) kann nach heutigem Kenntnisstand nicht beantwortet werden. Dies kann an unseren fehlenden mathematischen Fähigkeiten liegen, oder ein prinzipielles Problem darstellen (vgl. die Kontinuumshypothese).

Nun konstruieren wir uns ein spezielles Testprogramm, das auf Terminierung untersucht werden soll. Dieses Programm verwendet HALT:

```
def TEST(Programm):  
    wiederhole solange HALT(Programm, Programm) = „JA“  
    *wiederhole
```

Das Programm TEST soll nun untersucht werden. Fest steht:

TEST terminiert nicht, wenn HALT mit „JA“ terminiert. Sollte HALT nicht terminieren, so terminiert auch TEST nicht. Nur, wenn HALT mit „NEIN“ terminiert, terminiert auch TEST.

Was passiert, wenn TEST mit sich selber aufgerufen wird, also TEST(TEST):

Vorbemerkung:

TEST(TEST) verwendet HALT(TEST, TEST).

1. Fall: Angenommen, HALT(TEST, TEST) wäre „JA“

Wegen (I) würde TEST terminieren, weil das Programm HALT genau dann den Wert „JA“ liefert, wenn TEST für die Eingabe TEST terminiert. Wegen (II) würde TEST aber in diesem Fall nicht terminieren (Endlosschleife).

Widerspruch!

2. Fall: Angenommen, HALT(TEST, TEST) wäre „NEIN“

Wegen (I) würde TEST nicht terminieren, weil das Programm HALT genau dann den Wert „NEIN“ liefert, wenn TEST für die Eingabe TEST nicht terminiert. Wegen (II) würde TEST aber in diesem Fall sehr wohl terminieren (Schleifenabbruch).

Widerspruch!

Wegen der Widersprüche kann es einen Algorithmus, der für alle Programme feststellt, ob sie terminieren (oder nicht), nicht geben. Bitter, aber wahr!

Angenommen, das Halteproblem wäre lösbar gewesen, dann wäre der Beweis der Goldbachschen Vermutung einfach: Man schreibe einen Algorithmus, der alle goldbachzahlen aufschreibt. Terminiert das zugehörige Programm, ist die Goldbachsche Vermutung falsch, ansonsten ist sie richtig. Dies wird so wohl nicht passieren, oder doch?