

**ETHIK**  
**Einführung in die**  
**Theoretische Informatik**

Prof. Dr. Markus Chimani

Theoretische Informatik, Uni Osnabrück

Sommersemester 2025

# Einleitung

# **Einleitung**

Thema der Vorlesung

## Fragestellungen

- ▶ **Beobachtung:** In der Praxis gibt es viele verschiedene Computermodelle... und doch, von Gimmicks wie Graphikleistung abgesehen, tun sie alle das selbe: sie **rechnen** irgendetwas
- ▶ Was ist eine „einfache“ Beschreibung eines Computers der (prinzipiell) **alles** kann, was ein Supercomputer, traditioneller PC, Notebook, Tablet, Smartphone kann?
- ▶ Was kann ein Computer alles theoretisch berechnen? Was kann **kein** Computer berechnen??
- ▶ Was kann ein Computer **effizient** berechnen? Was bedeutet Effizienz?

...dazu müssen wir leider ein bisschen ausholen...

# Themenübersicht

- ▶ **Informationstheorie**

Was ist **Information** überhaupt?

- ▶ **Formale Sprachen** und **Automaten**

Wie kann ich zu lösende **Probleme** überhaupt definieren?  
Wie kann ich **Lösungsmethoden** bzw. **Berechnungen** definieren?

- ▶ **Berechenbarkeitstheorie**

Was kann ein Computer prinzipiell (nie) berechnen?

- ▶ **Komplexitätstheorie**

Was bedeutet Effizienz?

Was kann ein Computer prinzipiell (nie) effizient berechnen?

# **Einleitung**

## Organisatorisches

# Vorlesung

## 2x pro Woche:

- ▶ Donnerstags, 14:15 – 15:45 @ 66/E33
- ▶ Freitags, 10:15 – 11:45 @ 66/E33

## Unterlagen:

- ▶ Diese Folien → **Stud.IP**
- ▶ Empfehlung: Bücher, die „Theoretische Informatik“ im Titel haben (fast beliebig), z.B.
  - ▶ **Uwe Schöning.**  
*„Theoretische Informatik – kurz gefasst“*  
Spektrum Akad. Verlag, 5. Aufl., 190 Seiten, 39.99€.

# Modulprüfung: Klausur

## Zulassungsvoraussetzung:

- ▶ Bestandene **Kreuzerl-Übung** (siehe nächste Folien) im Jahr  $X$  gilt für die Klausurtermine im Jahr  $X$  und  $X + 1$ .
- ▶ Gültigkeitsdauer kann durch gute Übungsleistung gestreckt werden (siehe später).
- ▶ **Ausnahme für Klausuren nur noch in diesem Jahr (SoSe 2024):** Beliebig-alte Zulassungen gelten.

## Prüfungsmodus

- ▶ **Schriftliche Klausur, 120min**  
**Do, 7. August, 14:00–17:00**
- ▶ Nachklausur am Ende der vorlesungsfreien Zeit
- ▶ Fragestellungen:
  - ▶ Definitionen, Zusammenhänge, Beweisideen, und
  - ▶ Aufgabenstellungen ähnlich der Übung
  - ▶ vrgl. alte Klausuren: <https://tcs.uos.de/teaching/theoinf>

# Übung

Übungsleitung: **Mirko Wagner**

+ A. Birkemeyer, M. Donzelmann, K. Gorenflo, R. Henne, L. Kindermann

## Für die Klausurzulassung:

- ▶ wöchentliche **Kreuzerl-Übung**

## Weitere nicht-verpflichtende Angebote:

- ▶ Tutorium
- ▶ Schriftliche Abgaben

Wenn Sie die Klausurzulassung für dieses Jahr schon haben, können Sie **nicht** an der Kreuzerl-Übung teilnehmen!

Sie dürfen aber **gerne** am **Tutorium** und an den **schriftlichen Abgaben** teilnehmen.

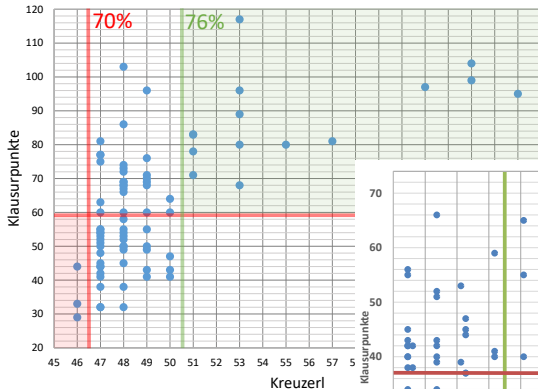
Sie können auf Ihre Klausurzulassung auch (irreversibel!) **verzichten**, um doch an der Kreuzerl-Übung teilzunehmen.

## Kreuzerl-Übung

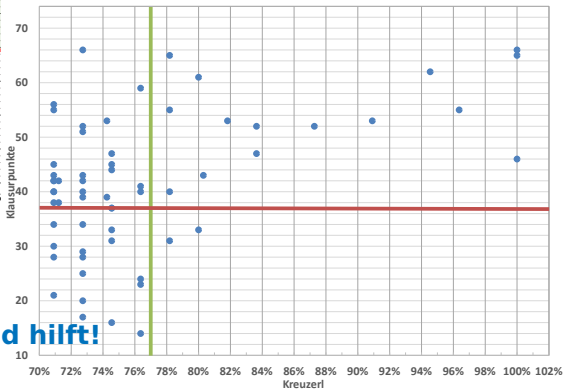
- ▶ **wöchentlich**, ca. 8 Übungsgruppen, **Aufgabenblätter** (11x 5 Aufgaben) immer mind. 1 Woche vorher verfügbar
- ▶ **Anmeldung:** Präferenzen in **StudIP** eintragen  
→ **ab Do, 17.4. bis Mo, 21.4.!**
- ▶ **Kreuzerl-Übung:**  
**Ankreuzen**, welche Lösungen Sie **erklären** können.  
Übungsbetreuer bittet per Pseudozufall jemanden zum Vorrechnen an der Tafel.
- ▶ **Voraussetzung für das Bestehen der Übung:**
  - ▶ Erfolgreiche Tafelleistungen
  - ▶ Mindestens 70% der Übungsaufgaben korrekt angekreuzt (= 39 der 55 Kreuzerl)
  - ▶ Bei zu unrichtig angekreuzter Aufgabe:
    1. Vergehen: Keine Bewertung des Übungsblatts.
    2. Vergehen: Kein Bestehen der Übung.

# Übung – Kreuzerl vs. Klausurpunkte

Haupttermin 2018 (falls auch Übung besucht)



Haupttermin 2024

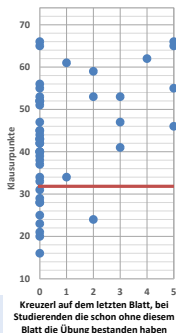


**Übung ist wichtig und hilft!**

# Übung – Verlängerung der Klausurzulassung

Durchschnittliche Kreuzerl pro Übungsblatt

	1	2	3	4	5	6	7	8	9	10	11
SoSe 2020	79%	92%	89%	83%	85%	85%	84%	74%	49%	29%	
SoSe 2021	80%	95%	93%	88%	81%	86%	86%	82%	68%	51%	16%
SoSe 2022	74%	90%	86%	87%	76%	85%	83%	74%	52%	29%	
SoSe 2023	77%	86%	96%	87%	80%	90%	83%	75%	75%	50%	20%
SoSe 2024	84%	88%	84%	88%	76%	88%	79%	86%	79%	59%	28%



Erfolgreiche Übungsteilnahme im Jahr  $X$ :  
Klausurzulassung für die Jahre  $X$  und  $X + 1$ .

## Zusätzlich:

Falls auch **70% der Kreuzerl** auf den **letzten 3 Übungsblättern** (=11 von 15 Kreuzerl):

Zulassung gilt auch noch im Jahr  $X + 2$  und  $X + 3$ .

# Übung – FAQ und Tutorium


## Kreuzerl-Übung (Mo,Di,Mi)

- ▶ **Fragen: Nach** dem Besprechen einer Aufgabe sind **beliebige** Fragen erlaubt — das gesetzte Kreuzerl bleibt!
- ▶ **Krankheiten und sonstige Ausnahmen:**
  - ⇒ möglichst **vorab** Bescheid geben
  - ⇒ Abgabe & Gespräch mit Übungsleiter (Mirko Wagner)
  - ⇒ Details siehe Ankündigung auf **StudIP**

## Tutorium: **Mittwochs, 16:15–17:45** (freiwillig!) 32/110

- ▶ Vorrechnen durch Übungsleiter (Mirko Wagner)
- ▶ Möglichkeiten für Fragen zu den Übungsaufgaben (inkl. der Aufgaben für folgende Woche!)
- ▶ Erster Termin schon **vor** der ersten Übung: **23. April!**

## Übung – Freiwillige(!) Abgaben

- ▶ **Ziel:** Saubere/formale Verschriftlichung üben  $\Rightarrow$  die Klausur ist schließlich auch schriftlich!
- ▶ **Freiwillig!**
- ▶ Vollkommen **losgelöst** vom Kreuzerlsystem: Sie können dadurch weder Kreuzerl gewinnen noch verlieren. Auch sonst kein Einfluss auf unsere Bewertungen!
- ▶ Mit „“ markierte Übungsaufgaben können digital abgegeben werden.  $\rightarrow$  Sie erhalten eine Korrektur, Hinweise zu Verbesserungen, etc. zurück.

### SoSe 2023: Korrelation Abgaben $\leftrightarrow$ Klausur

Anzahl <b>Abgaben</b> von 10	0	1	2-3	4-7	8-10
Anzahl Studenten	68	8	6	5	6
<b>Klausurpunkte</b> von 70 [Median]	33	39	42	46	59.5
<b>Klausurpunkte</b> von 70 [Durchschnitt]	34.6	39.5	42.8	45.8	58.5
<b>Note</b> des Medians (und Durchschnitts)	5,0	3,7	3,0	2,7	1,3

# Wichtige erste Termine (Übersicht)

## ▶ Vorlesungswoche 1 (14. – 18. April):

**Do:** Organisatorische VO (online) [dieses Video!]

Veröffentlichung von Übungsblatt 1

**Fr:** Vorlesung entfällt wg. Feiertag

## ▶ Vorlesungswoche 2 (21. – 25. April):

**bis Mo:** Präferenzen für Gruppeneinteilung abgeben

**Mi/Do:** Veröffentlichung der Gruppeneinteilung

**Mi:** Tutorium 0

**Do:** Vorlesung

**Fr:** Vorlesung, Veröffentlichung von Übungsblatt 2

**So:** Kreuzerl-Deadline Übungsblatt 1

## ▶ Vorlesungswoche 3 (28. April – 2. Mai):

**Mo/Di/Mi:** Kreuzerl-Übung Übungsblatt 1

**Mi:** Tutorium, Besprechung Übungsblatt 1

**Do:** Vorlesung entfällt wg. Feiertag

**Fr:** Vorlesung, Veröffentlichung von Übungsblatt 3

**So:** Kreuzerl-Deadline Übungsblatt 2

## ▶ u.s.w.

# **Einleitung**

## Formales Handwerkszeug

(für Sie zum Nachschlagen und ggf. im Tutorium besprechen)

# Griechische Buchstaben

A	$\alpha$	Alpha	N	$\nu$	Ny (Nü)
B	$\beta$	Beta	$\Xi$	$\xi$	Xi
$\Gamma$	$\gamma$	Gamma	O	o	Omikron
$\Delta$	$\delta$	Delta	$\Pi$	$\pi \varpi$	Pi
E	$\varepsilon$	Epsilon	P	$\rho \varrho$	Rho
Z	$\zeta$	Zeta	$\Sigma$	$\sigma \varsigma$	Sigma
H	$\eta$	Eta	T	$\tau$	Tau
$\Theta$	$\theta \vartheta$	Theta	$\Upsilon$	$\upsilon$	Ypsilon
I	$\iota$	Iota	$\Phi$	$\phi \varphi$	Phi
K	$\kappa$	Kappa	X	$\chi$	Chi
$\Lambda$	$\lambda$	Lambda	$\Psi$	$\psi$	Psi
M	$\mu$	My (Mü)	$\Omega$	$\omega$	Omega

# Mengen

$$\mathbb{N} = \{0, 1, 2, 3, \dots\}$$

$$\mathbb{N}_+, \mathbb{N}_g, \mathbb{N}_u$$

natürliche Zahlen, inklusive 0  
positive, gerade und  
ungerade natürliche Zahlen

$$A = \{a, b, c\}; \quad B = \{c, d\};$$

$$C = \{a, b, c, \dots, z\}; \quad D_1, \dots, D_k$$

endliche Mengen

$$\emptyset$$

leere Menge

$$A \subseteq C, \quad A \subset C, \quad A \not\subseteq B$$

$A$  ist (echte/keine) **Teilmenge**

$$C \supseteq A$$

$C$  ist **Übermenge**

$$a \in A, \quad a \notin B$$

$a$  ist **Element** von  $A$ , nicht von  $B$

$$A \cup B = \{a, b, c, d\}, \quad \bigcup_{1 \leq i \leq k} D_i$$

Vereinigung

$$A \cap B = \{c\}, \quad \bigcap_{1 \leq i \leq k} D_i$$

Schnittmenge

$$C \setminus A = \{d, \dots, z\}, \quad A \setminus B = \{a, b\}$$

„ $C$  ohne  $A$ “, „ $A$  ohne  $B$ “

$$A \triangle B = \{a, b, d\}$$

Symmetrische Differenz

$$A \times B = \{(a, c), (b, c), (c, c), \\ (a, d), (b, d), (c, d)\}$$

Kartesisches Produkt

# Logik

$true$ , ( <b>T</b> , $\top$ )	Symbol für „wahr“
$false$ , ( <b>F</b> , $\perp$ )	Symbol für „falsch“
$x, y, x_1, x_2, x_3, \dots$	Boolsche Variablen, nehmen den Wert $true$ oder $false$ an
$x \rightarrow y$	Implikation. Falls $x = true$ , dann $y = true$ ; sonst $y$ beliebig
$x \leftrightarrow y$	Äquivalenz, „ $x = y$ “. Falls $x = true$ , dann $y = true$ ; sonst $y = false$
$\neg x$	Negation. $false$ falls $x = true$ und umgekehrt
$x \wedge y$ , $\bigwedge_{1 \leq i \leq k} x_i$	Konjunktion („logisches Und“). $true$ nur falls $x$ und $y true$
$x \vee y$ , $\bigvee_{1 \leq i \leq k} x_i$	Disjunktion („logisches Oder“). $true$ nur falls mindestens $x$ oder $y true$
$x \oplus y$	Exklusive Disjunktion („exklusives Oder“). $true$ nur falls genau eines von $x$ und $y true$

## Logik (Fortsetzung)

**Belegung** einer Variablen: Zuweisung eines festen Werts (*true* oder *false*) zu einer Variablen.

Variable  $x$ : kann *true* oder *false* sein.

Belegung  $x'$  von  $x$  ist (beispielsweise) *true* („Einsetzen“).

### Formeln

$$A := x_1 \vee (x_2 \wedge x_3)$$

$$B(x) := (\neg x \rightarrow \text{true})$$

---

$\forall x: B(x)$	Allquantor. Für alle mögl. Belegungen $x'$ von $x$ gilt $B(x')$
$\exists x: B(x)$	Existenzquantor. Es gibt mindestens eine Belegung $x'$ von $x$ , sodass $B(x')$ gilt
$C \vdash D$ $B(x) \vdash \text{true}$	Deduktion. „Aus Formel $C$ folgt Formel $D$ “

---

## Arten von Aussagen

▶ **Aussage.**

Besteht aus **Voraussetzungen** (Grundannahmen, Hypothesen) und der **Konklusion** (Folgerung, Resultat, ...).

**Beispiel:** Sei  $n \geq 8$  eine Primzahl (*Voraussetzung*), dann ist  $n$  ungerade (*Konklusion*).

▶ **Theorem (=Satz).** Wichtige Aussage.

▶ **Lemma.** „Kleine“ Aussage, wird i. d. R. als Hilfsmittel benutzt, um ein Theorem beweisen zu können.

▶ **Beobachtung.** Sehr einfache Aussage.

▶ **Korollar (=Folgerung).** Aussage, die sich direkt aus einer anderen bewiesenen Aussage (Theorem, Lemma, etc.) ergibt.

## Arten von Beweisen

### Direkter Beweis (= deduktiver Beweis).

Kette von logischen Folgerungen, die von den Voraussetzungen bis zur Konklusion gelangt.

**Aussage.** Sei  $n \in \mathbb{N}_g$ ,  $n \geq 2$ . Es gilt  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ .

**Beweis.** Betrachte die kleinste und die größte Zahl; deren Summe ist  $1 + n$ . Allgemein ist die Summe aus der  $k$ -kleinsten und der  $k$ -größten Zahl immer  $k + (n - k + 1) = n + 1$ .

Wegen  $n \in \mathbb{N}_g$  ist die  $\frac{n}{2}$ -kleinste Zahl der direkte Vorgänger der  $\frac{n}{2}$ -größten Zahl. Es gibt also genau  $\frac{n}{2}$  viele Paare von  $k$ -kleinster/ $k$ -größter Zahl.

$\Rightarrow \frac{n}{2}$ -mal die Paar-Summe  $n + 1$  aufaddieren:  $\frac{n}{2}(n + 1)$ .  $\square$

## Arten von Beweisen

### Indirekter Beweis (= Beweis durch Widerspruch).

Nehmen Sie an, dass die Konklusion nicht gelten würde.  
Zeigen Sie, dass dies im Widerspruch zu den Voraussetzungen steht.

**Aussage.** Sei  $n \geq 8$  eine Primzahl, dann ist  $n$  ungerade.

**Beweis.** Angenommen  $n$  sei gerade. Dann ist  $n$  durch 2 teilbar. Um eine Primzahl zu sein, müsste also  $n = 2$  gelten. Widerspruch zu  $n \geq 8$ . □

# Arten von Beweisen

## Beweis durch Induktion.

Für Aussagen bzgl. einer Zahl, einer Menge,... (allgemein: Halbordnung).

**Induktionsanfang (IA):** Zeigen Sie, dass die Aussage für kleine Basisfälle (z. B. Zahl=1, einelementige Menge,...) gilt.

**Induktionsschritt (IS):** Betrachten Sie eine beliebige Zahl (Mengengröße)  $X$ . Zeigen Sie, dass die Aussage für  $X$  gilt, falls die Aussage für alle kleineren Zahlen (Mengen) gilt

(**Induktionshypothese (IH)**).

**Begründungslogik:** Angenommen, es gibt Zahlen (Mengen) für die die Aussage nicht stimmt. Wähle  $X$  als die kleinste dieser Zahlen (Mengen). Für alle kleineren Zahlen (Mengen) gilt die Aussage also, also die **IH**. Der **IS** beweist dann die Gültigkeit für  $X \rightarrow$  Widerspruch.

# Arten von Beweisen

## Beweis durch Induktion.

**Aussage.** Sei  $n \in \mathbb{N}_+$ . Es gilt  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ .

### Beweis.

*Induktionsanfang (IA):*

Die Aussage gilt für  $n = 1$  ✓

*Induktionshypothese (IH):*

Angenommen, die Aussage gilt für jedes  $n' < n$ .

*Induktionsschritt (IS):*

$$\begin{aligned} \sum_{i=1}^n i &= \sum_{i=1}^{n-1} i + n \stackrel{IH}{=} \frac{(n-1)n}{2} + n = \\ &= n \cdot \left( \frac{n-1}{2} + 1 \right) = n \cdot \frac{n-1+2}{2} = n \frac{(n+1)}{2}. \quad \square \end{aligned}$$

## Arten von Beweisen

### Beweis durch Autorität.

Dem Vortragenden vorbehalten.

In Lösungen der Übungsaufgaben nicht zulässig.

**Aussage.** Sei  $z$  eine reelle oder komplexe Zahl. Es gilt:

$$\lim_{n \rightarrow \infty} \left(1 + \frac{z}{n}\right)^n = e^z.$$

**Beweis.** Das gilt. Glauben Sie mir. Der Beweis wäre jetzt zu lang, kompliziert und irrelevant. □

# Arten von Beweisen

## Beweis durch Einschüchterung.

THE AXIOM OF CHOICE ALLOWS  
YOU TO SELECT ONE ELEMENT  
FROM EACH SET IN A COLLECTION

AND HAVE IT EXECUTED AS  
AN EXAMPLE TO THE OTHERS.



Bildquelle: [xkcd.com/982](http://xkcd.com/982)

# Informationstheorie

# Informationstheorie

## Information

# Informatik?

Deutsch:	<b>Informatik</b>	1956, Karl Steinbuch Buch: „Informatik: Automatische Informationsverarbeitung“
Englisch:	<b>Computer science</b> aber <b>bio-, geo-informatics</b>	1956/59, Louis Fein

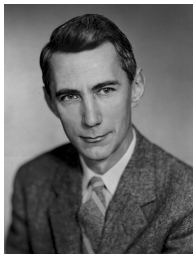
*„In der Informatik geht es nicht mehr um Computer, als in der Astronomie um Teleskope.“ – E. W. Dijkstra?*

Französisch:	<b>Informatique</b>	1962, Philippe Dreyfus
Dänisch:	<b>Datalogi</b>	1966, Peter Naur
Holländisch:	<b>Informatica</b>	
Spanisch:	<b>Ciencias de la computacion; informática</b>	
Portugisisch:	<b>Ciência da computação</b>	
Italienisch:	<b>Informatica</b>	
Russisch:	<b>Информатика</b>	
Polnisch:	<b>Informatyka</b>	
Türkisch:	<b>Bilgisayar bilimi, enformatik</b>	
Kurdisch:	<b>Informatîk</b>	
Arabisch:	<b>(eulim alhasub ?)</b>	
Chinesisch:	Übersetzung von <b>Computerwissenschaften</b>	
Japanisch:	Übersetzungen von <b>Informatik (jōhō-gaku)</b> , aber auch <b>konpyūta-saiensu</b>	
Hindi, Urdu,...:	Übersetzungen von <b>Computerwissenschaften</b>	

## Information

Was ist das überhaupt?

# Claude Elwood Shannon, 1916 – 2001 (USA)



## Vater der Informationstheorie

1936: BSc Elektrotechnik [U. Michigan] +  
BSc Mathematik [U. Michigan]

1937/38: MSc Elektrotechnik [MIT]  
*„A Symbolic Analysis of Relay and  
Switching Circuits“*: digitale Logik-  
schaltkreise = Boolsche Algebra,  
4-bit full adder

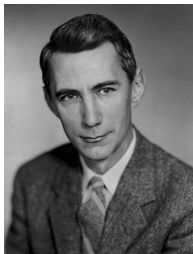
1940: PhD Mathematik [MIT]  
zu theoretischer Genetik

Forschungen @ [Princeton]

(J. von Neumann, K. Gödel, H. Weyl, A. Einstein)



# Claude Elwood Shannon, 1916 – 2001 (USA)



## Vater der Informationstheorie

1941ff: AT&T Bell Labs → WW2!

Kryptographie (one-time pad)

Signalflussgraphen ('42)

Treffen mit Alan Turing ('43)

„*A Mathematical Theory of Cryptography*“ ('45)

→ geheim, aber Vorläufer von:

1948: „*A Mathematical Theory of Communication*“

## Geburt der Informationstheorie

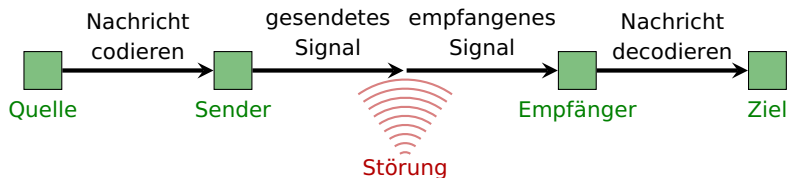
1949: Erstes Paper über ein Schachprogramm

1950: Mechanische Maus in Labyrinth (AI, Learning)

1956–1978: Professor am MIT



# Anwendungen der Informationstheorie



- ▶ Codierung, Kompression
- ▶ Fehlererkennung
- ▶ Datenübertragung mit/ohne Noise

## Eigenschaften von Information, 1/3

Welcher Satz liefert mehr Information?

- ▶ Heute ging die Sonne auf.
- ▶ Heute: Sonne ging auf.
- ▶ Heute besuchten uns Außerirdische.

„überraschender“ → mehr Information

Nicht (nur) abhängig von der Länge, sondern auch von

**Wahrscheinlichkeit.**

## Eigenschaften von Information, 2/3

Ich denke mir einen Satz aus.

- ▶ Was ist der erste Buchstabe des Satzes?
- ▶ Was ist der nächste Buchstabe bei diesem Satzanfang?

**GUTEN MOR\_**

Der **selbe** Buchstabe liefert unterschiedlich viel Information, je nachdem wo/wann er auftritt.

→ (Bedingte) Wahrscheinlichkeit

## Eigenschaften von Information, 3/3

Ich erhalte von Personen **A** und **B** jeweils eine Datei mit  $i_A$  bzw.  $i_B$  „viel“ Information (was auch immer das bedeutet).

Wieviel Information habe ich insgesamt erhalten?

$$\max\{i_A, i_B\} \leq i_{\text{insgesamt}} \leq i_A + i_B$$

Wenn die beiden Informationen **unabhängig** sind, dann

$$i_{\text{insgesamt}} = i_A + i_B$$

Unabhängige Information ist **additiv**.

# Würfeln

**Informationsgewinn** durch ein Ereignis?

**Vorher:**

W'keit für jede Zahl

normaler Würfel:

$1/6$

2-seitiger Würfel (=Münze):

$1/2$

16-seitiger Würfel:

$1/16$

**Nachher:** Es ist klar, was gewürfelt wurde.

„Überraschung“, dass man eine **1** gewürfelt hat?

gar nicht	klein	mittel	groß	unmöglich
Würfel mit lauter 1en	Münze	Würfel	16-s. W.	Würfel mit lauter 6en

**Informationsgewinn des Würfels?**

0

1

?

?

$\infty$

Normierung  
binär

← **Informationseinheit Bit (bit)**

# Informationsgewinn bei 16-seitigem Würfel

W'keit eine **1** zu würfeln =  $1/16$ .

Oje, wir haben keinen 16er Würfel zur Hand, aber eine Münze!

→ **Simuliere** einen 16er-Würfel mit mehreren Münzwürfen:

Ergebnisse des 16-Würfels:  $0, 1, 2, \dots, 15$

Binär:  $0000, 0001, 0010, \dots, 1111$

→ Erwürfle das Ergebnis des 16er-Würfels mittels 4  
**unabhängiger** Münzwürfe (ein Münzwurf pro Stelle)

Jeder Münzwurf liefert **1 bit** Informationsgewinn.

Unabhängige Informationen sind additiv.

⇒ die Simulation liefert **4 bit** Information

⇒ der 16er-Würfel liefert **4 bit** Information

## Informationsgewinn

Sei  $\mathcal{I}(p)$  der Informationsgewinn eines Ereignisses  $E$ , das mit W'keit  $p := p_E$  ( $0 \leq p \leq 1$ ) eintritt.

### Welche Funktion kann $\mathcal{I}$ sein?

Eigenschaften:

- ▶  $\mathcal{I}(p) \geq 0, \forall p$
- ▶  $\mathcal{I}(p)$  steigt für kleiner werdende  $p \Rightarrow$  streng monoton fallend
- ▶ Kleine Änderung in  $p \rightarrow$  kleine Änd. in  $\mathcal{I}(p) \Rightarrow$  stetig
- ▶  $\mathcal{I}(0) = \infty \quad \mathcal{I}(1/2) = 1 \quad \mathcal{I}(1) = 0$
- ▶  $\mathcal{I}(p_E \cdot p_F) = \mathcal{I}(p_E) + \mathcal{I}(p_F)$ , für unabh. Ereignisse  $E, F$

Informationsgewinn durch ein Ereignis mit W'keit  $p$ :

$$\mathcal{I}(p) := \log_2 \frac{1}{p} = -\log_2 p$$

Wir definieren für  $p = 0$  konsistent mit  $\lim_{p \rightarrow 0}$ :

$$\begin{array}{ll} \log_2 \frac{1}{p}: & \log_2 \frac{1}{0} = -\log_2 0 := \infty \\ p \log_2 \frac{1}{p}: & 0 \log_2 \frac{1}{0} = -0 \log_2 0 := 0 \end{array}$$

# Würfeln (nochmal)

Würfel mit lauter 1en	Münze	Würfel	16-s. W.	Würfel mit lauter 6en
<b>Überraschung, eine 1 zu würfeln:</b>				
gar nicht	klein	mittel	groß	unmöglich
<b>W'keit <math>p</math>, eine 1 zu würfeln:</b>				
1	1/2	1/6	1/16	0
<b>Informationsgewinn <math>\log_2 1/p</math> des Würfels in bit:</b>				
$\log_2 1$ 0	$\log_2 2$ 1	$\log_2 6$ $\approx 2.585$	$\log_2 16$ 4	$\log_2 \frac{1}{0}$ $\infty$

# Informationstheorie

## Entropie

# Entropie = Erwarteter Informationsgewinn

Sei  $\Sigma$  ein endliches **Alphabet** (= Menge von Symbolen).

Jedes Symbol  $\sigma \in \Sigma$  hat eine W'keit  $p_\sigma \Rightarrow$  **Quelle**  $(\Sigma, p)$

**Bsp:**  $\Sigma = \{a, b, c, \dots, z\}$

$$p_a = 0.6, p_b = 0.1, p_c = 0.3, p_d = \dots = p_z = 0.0$$

**Physik: Entropie** = Maß der **Unordnung**

**Informatik: Entropie** = **Erwarteter Informationsgewinn**

Entropie  $H_{\Sigma, p}$  (griechisches **Eta**)

= **Erwarteter** Inf'gewinn durch **ein** Symbol

$$H_{\Sigma, p} = \mathbb{E}[I] = \sum_{\sigma \in \Sigma} p_\sigma \cdot I(p_\sigma) = - \sum_{\sigma \in \Sigma} p_\sigma \log_2 p_\sigma$$

$$H_{\Sigma, p} = -0.6 \log_2 0.6 - 0.1 \log_2 0.1 - 0.3 \log_2 0.3 - 0 \dots - 0 \\ \approx \mathbf{1.3 \text{ bit}}$$

# Maximale und zusammengesetzte Entropie

Entropie  $H_{\Sigma,p}$  = Erwarteter Inf'gewinn durch ein Symbol

$$H_{\Sigma,p} = \mathbb{E}[\mathcal{I}] = \sum_{\sigma \in \Sigma} p_{\sigma} \cdot \mathcal{I}(p_{\sigma}) = - \sum_{\sigma \in \Sigma} p_{\sigma} \log_2 p_{\sigma}$$

Entropie maximal  $\leftrightarrow$  alle Symbole **gleich** wahrscheinlich

Erw. Inf'gewinn  $H_{\Sigma,p}(n)$  einer Nachricht aus  $n$  Zeichen, wenn die einzelnen Zeichen gemäß  $p$  und unabhängig sind?

→ Inf'gewinn  $\mathcal{I}$  ist additiv wenn Symbole unabhängig sind

→ gilt auch für dessen Erwartungswert  $H_{\Sigma,p}(n)$

⇒  $H_{\Sigma,p}(n) = n \cdot H_{\Sigma,p}$

# Apropos Entropie: Information vs. Physik

bit  
↓

$$H_{\text{Information}} = - \sum_i p_i \log_2 p_i$$

Joule/Kelvin  
↓

$$H_{\text{Physik}} = - k_B \sum_i p_i \ln p_i$$

↑  
Boltzmannkonstante:  $1.38e-23$  J/K

**Information** ist (auch) ein **physikales Phänomen!**

- ▶ Auflösung für den Maxwellschen Dämon
- ▶ Informationsparadoxon schwarzer Löcher
- ▶ Landauer-Prinzip:

## Irreversibel arbeitende Computer:

Betrachte den Codebefehl  $i=j$  für zwei 64bit-Variablen  $i, j$ .  
Durch diesen Befehl wird Information **vernichtet!**

→ Entropie nimmt **um 64 bit zu**:

$$\rightarrow 64 \cdot k_B \ln 2 \approx 6.125 \cdot 10^{-22} \text{ J/K}$$

⇒ Destruktive Variablenzuweisungen können **nie**  
energieeffizienter sein!

## Entropie einer Sprache

**Entropie einer Sprache** = Durchschnittlicher Informationsgewinn eines einzelnen Symbols in einem zufälligen Text der in der betracht. Sprache geschrieben ist.

**Beispiel: Englischer Text**, 27 Symbole (26 Buchst. + Space)  
(ohne Satzzeichen, Groß-/Kleinschreibung,..)

Shannon: **0.6–1.3 bit**

Dieser Wert lässt sich nur schätzen!

Warum so wenig? **Redundanz!**  $u$  nach  $q$ ; Vokale nach 1–2 Konsonanten; Häufigkeit von  $e = 12.6\%$ ;...

Einfachste Art, Texte zu **codieren**: **Blockcode**  
= lege eine feste Bitlänge fest, und codiere jedes Zeichen mit einer Zahl innerhalb dieses Wertebereichs.

⇒ 27 Symbole: **5 bit** pro Symbol! ( $2^5 = 32$  versch. Werte)  
In ASCII-Codierung sogar **8 bit**.

# Präfix-Code

**Idee:** Symbole mit unterschiedl. langen Bitmustern codieren.  
Häufige Symbole benötigen weniger Bits als seltene.

$$p_a = 0.6, \quad p_b = 0.1, \quad p_c = 0.3$$

$$\mathbb{C}(a) = 1, \quad \mathbb{C}(b) = 00, \quad \mathbb{C}(c) = 01$$

**Binärer Präfix-Code**  $\mathbb{C}$  für ein Alphabet  $\Sigma$ :

Codierung von Symbolen  $\sigma \in \Sigma$  in Binärzahlen, so dass kein Symbol  $\sigma \in \Sigma$  existiert, dessen Codierung  $\mathbb{C}(\sigma)$  genau der Anfang eines  $\mathbb{C}(\sigma')$ , für irgendein  $\sigma' \neq \sigma$ , ist.

**Vorteil:** Kein Trenner zwischen Symbolen notwendig!

$$\mathbb{C}(r) = 101, \quad \mathbb{C}(t) = \underline{101}10 \quad \times$$

## Quellencodierungstheorem (Shannon)

Betrachte Quelle  $(\Sigma, p)$ . Sei  $\mathbb{C}$  ein binärer Präfix-Code für  $\Sigma$ , und  $|\mathbb{C}(\sigma)|$  die Länge der Codierung von  $\sigma \in \Sigma$ .

**Erwartete Codewort-Länge**  $L_{\Sigma,p}(\mathbb{C}) := \sum_{\sigma \in \Sigma} p_{\sigma} \cdot |\mathbb{C}(\sigma)|$   
= Erwartungswert der Länge der Binärcodierung eines zufälligen Symbols

**Quellencodierungstheorem (für Symbole):** [ohne Beweis]

- 1 Für jedes  $\mathbb{C}$  gilt:  $L_{\Sigma,p}(\mathbb{C}) \geq H_{\Sigma,p}$ .
- 2 Es existiert ein  $\mathbb{C}$  mit  $L_{\Sigma,p}(\mathbb{C}) < H_{\Sigma,p} + 1$ .

**Intuitiv heißt das:**

@1: Keine Codierung kann besser als die Entropie sein.

@2: Entropie-Schranke ist auch (fast) erreichbar! Es gibt nämlich eine Codierung die im Durchschnitt pro Symbol **echt weniger als 1 bit mehr** als die Entropie braucht.

# Informationstheorie

## Huffman-Codes

# Ziel

## Quellencodierungstheorem (Shannon; für Symbole):

- 1 Für jedes  $\mathbb{C}$  gilt:  $L_{\Sigma,p}(\mathbb{C}) \geq H_{\Sigma,p}$ .
- 2 Es existiert eine  $\mathbb{C}$  mit  $L_{\Sigma,p}(\mathbb{C}) < H_{\Sigma,p} + 1$ .

Wie findet man ein  $\mathbb{C}$ , das **2** erfüllt?

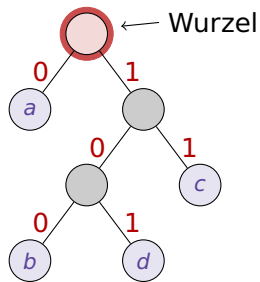
**Shannon-Fano-Codes:** Claude Shannon (1948) und Robert Fano (1949); vorgeschlagene Verfahren fast gleich.

Die Verfahren erfüllen Eigenschaft **2**, aber sie sind i.A. nicht **optimal**, also nicht so nah wie möglich an der Entropie.

Wir suchen eine Präfix-Codierung  $\mathbb{C}$  mit **minimalem**  $L_{\Sigma,p}(\mathbb{C})$ .

⇒ **Huffman-Code** (David A. Huffman, 1952)

## Binärer Präfix-Code als Binärbaum



$$C(a) = 0$$

$$C(b) = 100$$

$$C(c) = 11$$

$$C(d) = 101$$

- 1 Symbole = Blätter
- 2 Innere Knoten: **immer** 2 Kinder
- 3 **Codewort:** Wurzel  $\rightarrow$  Symbol.  
Abstieg in linkes/rechtes Kind  
fügt Codewort  $0/1$  hinzu

**Beobachtung.** Binärer Baum  $T$ .

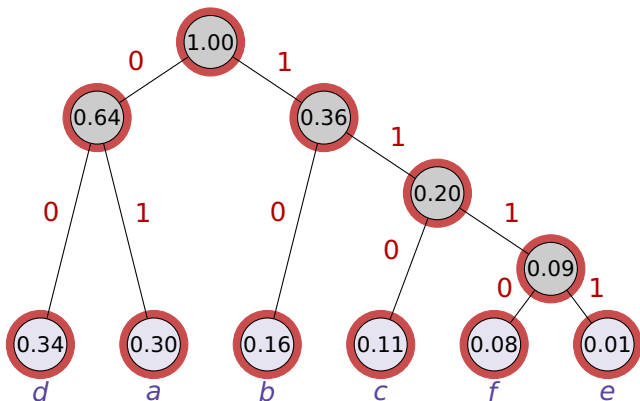
- ▶ Jeder  $T$  mit **1** und **3** liefert einen Präfix-Code.
- ▶ Jeder optimale Präfix-Code entspricht einem  $T$  mit **1** – **3**.

$|C(\sigma)|$  = Codewort-Länge von  $\sigma$  = Tiefe des Blatts von  $\sigma = d_\sigma$

$$L_{\Sigma,p}(C) = \sum_{\sigma \in \Sigma} p_\sigma \cdot |C(\sigma)| = \sum_{\sigma \in \Sigma} p_\sigma \cdot d_\sigma = L_{\Sigma,p}(T)$$

# Huffman-Code Algorithmus — Beispiel

$\sigma$	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
$p_\sigma$	0.30	0.16	0.11	0.34	0.01	0.08
$C(\sigma)$	01	10	110	00	1111	1110



# Huffman-Code Algorithmus

Menge  $\mathcal{A} = \emptyset$

**FOR ALL**  $\sigma \in \Sigma$ :

Füge einen (Blatt)knoten  $B_\sigma$  mit Gewicht  $p_\sigma$  zu  $\mathcal{A}$  hinzu

**WHILE**  $|\mathcal{A}| > 1$  :

Entferne Knoten  $K_0$  mit minimalem Gewicht  $q_0$  aus  $\mathcal{A}$

Entferne Knoten  $K_1$  mit minimalem Gewicht  $q_1$  aus  $\mathcal{A}$

neuer Knoten  $N$  mit Gewicht  $q_0 + q_1$

Setze das  $i$ -te Kind von  $N$  auf  $K_i$  ( $i \in \{0, 1\}$ )

Füge  $N$  zu  $\mathcal{A}$  hinzu

## Was zu beweisen ist...

**Theorem.** Huffman-Codes garantieren minimales  $L_{\Sigma,p}(\mathbb{C})$ .

**Beweis.** folgt...

Zusammen mit Shannons Quellencodierungstheorem ergibt das dann auch:

**Korollar.** Huffman-Codes garantieren  $L_{\Sigma,p}(\mathbb{C}) < H_{\Sigma,p} + 1$ .

## Vorbeobachtungen: Optimaler Präfix-Code

**Lemma.** Seien  $\sigma_1, \sigma_2$  zwei unwahrscheinlichste Symbole. Es gibt einen **optimalen** Präfix-Code-Baum in dem  $\sigma_1$  und  $\sigma_2$  einen gemeinsamen Elterknoten haben.

**Beweis.** Starte mit beliebigem optimalen Präfix-Code-Baum, d.h. mit minimaler erwarteter Codewortlänge  $L_{\Sigma, p}(\mathbb{C})$ .

Angenommen,  $\sigma_1$  nicht auf unterster Ebene: Tausche  $\sigma_1$  mit einem Blatt  $\tau$  auf unterster Ebene. Falls  $p_\tau > p_{\sigma_1}$  würde  $L_{\Sigma, p}(\mathbb{C})$  nun sinken (Widerspruch).  $\Rightarrow$  Also  $p_\tau = p_{\sigma_1}$  und im neuen Baum bleibt  $L_{\Sigma, p}(\mathbb{C})$  unverändert (optimal).

Sei  $\rho$  der Bruderknoten von  $\sigma_1$  (es muss ja genau einer existieren!). Tausche  $\rho$  und  $\sigma_2$ . Da der Baum optimal war, muss  $d_{\sigma_2} = d_\rho$  oder  $p_\rho = p_{\sigma_2}$  gelten (sonst würde  $L_{\Sigma, p}(\mathbb{C})$  beim Tausch sinken). So bleibt  $L_{\Sigma, p}(\mathbb{C})$  gleich und Zielzustand des Lemmas ist erreicht.  $\square$

# Optimalität von Huffman-Code

**Theorem.** Huffman-Codes garantieren minimales  $L_{\Sigma,p}(\mathbb{C})$ .

**Beweis (durch Induktion).**

$$L_{\Sigma,p}(T) = \sum_{\sigma \in \Sigma} p_{\sigma} d_{\sigma}$$

**Anfang:** Theorem gilt falls  $|\Sigma| \leq 2$ .

**Hypothese (IH):** Theorem gilt für  $|\Sigma| < n$ .

**Schritt:** Sei  $|\Sigma| = n$  mit unwahrscheinlichsten Symbolen  $\sigma_1, \sigma_2$ .

**Gemäß Lemma:** Im Huffman-Baum  $T_h$  (Algo.) **und** im optimaler Baum  $T_o$  haben  $\sigma_1, \sigma_2$  einen gemeinsamen Vater  $\tau$ .

Verschmelze  $\sigma_1, \sigma_2, \tau$  in beiden Bäumen  $\Rightarrow T'_h, T'_o \Rightarrow$   
 Präfix-Codes für Quelle  $(\Sigma', p')$  mit  $\Sigma' := \Sigma \setminus \{\sigma_1, \sigma_2\} \cup \{\tau\}$  und  
 $p' \cong p$  mit  $p'_{\tau} = p_{\sigma_1} + p_{\sigma_2}$ .  $\Rightarrow$  **Oh!** Huffman-Algo erzeugt  $T'_h$ !

**IH:**  $|\Sigma'| < n \Rightarrow T'_h$  ist optimal  $\Rightarrow L_{\Sigma',p'}(T'_h) \leq L_{\Sigma',p'}(T'_o)$

$$\begin{aligned} L_{\Sigma,p}(T_h) &= L_{\Sigma',p'}(T'_h) - p_{\tau} d_{\tau} + p_{\sigma_1} (d_{\tau} + 1) + p_{\sigma_2} (d_{\tau} + 1) \\ &\leq L_{\Sigma',p'}(T'_o) - p_{\tau} d_{\tau} + p_{\sigma_1} (d_{\tau} + 1) + p_{\sigma_2} (d_{\tau} + 1) = L_{\Sigma,p}(T_o) \square \end{aligned}$$

# Bedenkenswertes und Grenzen der Huffman-Codes

- ▶ Unterschiedliche Codewort-Längen je nach Symbol  
⇒ Blockcodes sind schneller zu Decodieren
- ▶ Kompression entfernt Redundanz  
→ anfällig für Übertragungsfehler  
⇒ dafür gibt es Error-Correcting Codes
- ▶ Huffman-Codes sind optimal wenn die Symbol-W'keiten stets **a** konstant, **b** bekannt und **c** unabhängig von den anderen Symbolen der Nachricht sind.  
⇒ Sonst: Schlauere Verfahren nötig,  
z.B. Lempel-Ziv, ZIP,...  
siehe auch: [VO Kodierungstheorie](#)

# **Formale Sprachen & Automaten**

# **Formale Sprachen & Automaten**

Sprachen & Chomsky-Hierarchie

# Optimierungs- und Entscheidungsprobleme

## Kombinatorische Optimierungsprobleme

- ▶ Kürzester Weg, Maximaler Fluss, Traveling Salesman,...

## Entscheidungsprobleme

- ▶ **Jedes** kombinatorische Optimierungsproblem lässt sich als Entscheidungsproblem simulieren!  
Gibt es einen Weg der Länge maximal  $k$ ?  
Gibt es einen Fluss der Größe mindestens  $k'$ ?  
Gibt es eine Rundtour der Länge maximal  $k''$ ? ...
- ▶ Gibt es in dem Graph überhaupt eine Rundtour die jeden Knoten genau 1x besucht?  
Ist ein gegebener Text ein gültiger C++ Code?
- ▶ Satisfiability-Probleme (SAT): Gegeben eine aussagenlogische Formel mit Variablen. Kann man eine Variablenbelegung finden, so dass die Formel erfüllt wird?

# Sprachen vs. Probleme

## Kombinatorische Optimierungsprobleme

↓ **Jedes** kombinatorische Optimierungsproblem lässt sich als Entscheidungsproblem simulieren.

## Entscheidungsprobleme

↓ **Jedes** Entscheidungsproblem lässt sich als Wortproblem über eine geeignete Sprache definieren!

## Wortproblem auf Sprachen

- ▶ Sprachen sind formal gut handhabbar
  - **das** „Hauptproblem“ der theoretischen Informatik, wenn es um Berechenbarkeit und Komplexität geht
  - alle Aussagen lassen sich zu „normalen“ Optimierungs-, Such- und Entscheidungsproblemen transferieren.

# Noam Chomsky



## **Avram Noam Chomsky**

Geboren: 7. Dezember 1928

in Philadelphia, USA

Professor für Linguistik am MIT

- ▶ Linguist mit Ausstrahlung in die Kognitionswissenschaften und Informatik
- ▶ linker Politik-Kritiker, Free-Speech-Advokat
- ▶ „Erfinder“ der generativen Transformationsgrammatik

„Jede natürliche Sprache ist rekursiv aufgebaut“

(oops... Pirahã)

# Grammatik in natürlichen Sprachen

**Grammatik** umfasst in der Linguistik

▶ **Morphologie**

Formenlehre der Worte (gehen, ging, gegangen)

▶ **Syntax**

Satzbau

(**die** Syntax, nicht **der** Syntax)

▶ **Phonologie**

Lautlehre

▶ **Semantik**, falls sie sich auf Aufbau-Regeln bezieht  
Bedeutungslehre

# Satzbau in natürlichen Sprachen

## Grammatik/Syntax (Beispiel):

⟨Satz⟩	→	⟨Subjekt⟩	⟨Prädikat⟩	⟨Objekt⟩				
⟨Subjekt⟩	→	⟨Artikel⟩	⟨Attribut⟩	⟨Substantiv⟩				
⟨Objekt⟩	→	⟨Artikel⟩	⟨Attribut⟩	⟨Substantiv⟩				
⟨Attribut⟩	→	$\epsilon$		⟨Adjektiv⟩	⟨Attribut⟩			
⟨Artikel⟩	→	der		die		das		
⟨Adjektiv⟩	→	kleine		schwierige		undurchsichtige		
⟨Substantiv⟩	→	Zauberer		Professor		Theorem		Glaskugel
⟨Prädikat⟩	→	befragt		beweist				

## Mögliche Ableitungen:

„der Zauberer befragt die kleine undurchsichtige Glaskugel“

„der undurchsichtige Professor beweist das schwierige Theorem“

„die Theorem befragt der kleine schwierige kleine Theorem“

## Wörter in formalen Sprachen

- ▶ Statt von „Sätzen“ sprechen wir von **Wörtern**.
- ▶ Ein Wort besteht aus **Symbolen**.  
Spezialfall: **Leerwort**  $\varepsilon$  besteht aus 0 Symbolen.
- ▶ Die möglichen Symbole bilden das (endliche!) **Alphabet**.

### Übliche Bezeichner

- ▶ Alphabet:  $\Sigma$
- ▶ Symbole  $\sigma \in \Sigma$ :  $a, b, c, \dots$
- ▶ Wörter:  $u, v, w, x, y, z$

### Bau von Wörtern

- ▶  $w := ab$   $\Rightarrow$  Wort aus den Symbolen  $a$  und  $b$ .
- ▶  $u := wcw$   $\Rightarrow$  Wort  $abcab$
- ▶  $v := a^3$   $\Rightarrow$  Wort  $aaa$
- ▶  $x := a^{n+2}bc^3b^n(de)^2$   $\Rightarrow$  Wort  $\underbrace{a \dots aa}_{n} abccc \underbrace{b \dots b}_{n} dede$

# Formale Sprachen

- ▶ Gegeben ein (endliches, nicht-leeres) Alphabet  $\Sigma$ .
- ▶  $\Sigma^*$  = Menge **aller möglichen Wörter** (inkl. Leerwort  $\varepsilon$ ) die aus Symbolen aus  $\Sigma$  gebildet werden können.  
→ „Kleene-Stern“
- ▶ Eine Teilmenge  $L \subseteq \Sigma^*$  ist eine **Sprache**. (Language)

## Beispiele und weitere Notation

- ▶  $L_1 := \{a^n b^n \mid n \in \mathbb{N}_+\} \Rightarrow \{ab, aabb, aaabbb, \dots\}$
- ▶  $L_2 := \Sigma^7 = \Sigma^2 \Sigma \Sigma^4 \Rightarrow$  Alle Wörter der Länge 7
- ▶  $L_3 := \Sigma^* \setminus \{\varepsilon\} = \Sigma^+ \Rightarrow$  Alle Wörter mit mind. 1 Symbol
- ▶  $L_4 := \{w \in \Sigma^* \mid |w| \bmod 2 = 0\} = \bigcup_{n \in \mathbb{N}} \Sigma^{2n} \Rightarrow$  Alle Wörter mit gerader Länge
- ▶  $L_5 := \{w \in \Sigma^* \mid w \text{ enthält mindestens 3x das Symbol } a\}$

## Operationen auf Sprachen

**Sprachen sind Mengen!** (Achtung: in unserer Notation könnte  $\Sigma$  auch eine Sprache mit ein-symboligen Wörtern sein!)

⇒ Wir definieren vorige Operationen für **Mengen**, nicht nur für Alphabete. Seien  $L$  und  $L'$  zwei Sprachen/Mengen.

- ▶ **Verkettung**  $LL' := \{xy \mid x \in L, y \in L'\}$

Wörter die entstehen, wenn man ein Wort aus  $L$  und eines aus  $L'$  aneinander schreibt.

- ▶ **Potenz**  $L^i := \{x_1x_2 \dots x_i \mid x_1, x_2, \dots, x_i \in L\}$   
 $i$  viele Wörter aus  $L$  aneinander geschrieben.

- ▶ **Kleene-Stern**  $L^* := \bigcup_{i \geq 0} L^i$

Beliebig viele (auch 0!) Wörter aus  $L$  aneinander geschrieben. Analog:  $L^+ := \bigcup_{i \geq 1} L^i$ .

**Beispiel.**  $\{la, lu\}^* \{da, dum\}^2 =$

$\{da da, dum dum, \dots, la la lu la lu da dum, \dots\}$

# Formale Sprachen

- ▶ Gegeben ein Alphabet  $\Sigma$ .
- ▶  $\Sigma^*$  = Menge **aller möglichen Wörter** (inkl. Leerwort  $\varepsilon$ ) die aus Symbolen aus  $\Sigma$  gebildet werden können.  
→ „Kleene-Stern“
- ▶ Eine Teilmenge  $L \subseteq \Sigma^*$  ist eine **Sprache**. (Language)

## Beispiele und weitere Notation

- ▶  $L_1 := \{a^n b^n \mid n \in \mathbb{N}\} \Rightarrow \{ab, aabb, aaabbb, \dots\}$
- ▶  $L_2 := \Sigma^7 = \Sigma^2 \Sigma \Sigma^4 \Rightarrow$  Alle Wörter der Länge 7
- ▶  $L_3 := \Sigma^* \setminus \{\varepsilon\} = \Sigma^+ \Rightarrow$  Alle Wörter mit mind. 1 Symbol
- ▶  $L_4 := \{w \in \Sigma^+ \mid |w| \bmod 2 = 0\} = \bigcup_{n \in \mathbb{N}_+} \Sigma^{2n} \Rightarrow$  Alle Wörter mit gerader Länge
- ▶  $L_5 := \{w \in \Sigma^* \mid w \text{ enthält mindestens 3x das Symbol } a\}$   
...sehr informell. Sprachen formaler beschreiben?

# Grammatik (in der Informatik)

Eine **Grammatik**  $G$  besteht aus:

- ▶ **Alphabet**  $\Sigma$  = Menge von **Symbolen**

üblicherweise  $a, b, c, \dots$

- ▶ **Variablen**  $\mathcal{V}$

üblicherweise  $A, B, C, \dots$

- ▶ **Startvariable**

eine der Variablen, üblicherweise  $S$ .

Falls in Beispielen kein  $S$  vorkommt, dann Variable der ersten Regel, meist  $A$

- ▶ **Regeln**

(= Produktionsregeln, Ableitungsregeln)

Jede Regel hat die Gestalt  $(\mathcal{V} \cup \Sigma)^* \mathcal{V} (\mathcal{V} \cup \Sigma)^* \rightarrow (\mathcal{V} \cup \Sigma)^*$

# Beispiel & Backus-Naur

## Ausführlich

$$A \rightarrow AB$$

$$A \rightarrow CA$$

$$A \rightarrow a$$

$$A \rightarrow \varepsilon$$

$$B \rightarrow b$$

$$B \rightarrow Bb$$

$$C \rightarrow A$$

$$C \rightarrow cA$$

$$C \rightarrow cc$$

$$AaA \rightarrow d$$

$$\left. \begin{array}{l} \\ \\ \\ \end{array} \right\}$$

$$\left. \begin{array}{l} \\ \end{array} \right\}$$

$$\left. \begin{array}{l} \\ \\ \end{array} \right\}$$

$$\left. \begin{array}{l} \end{array} \right\}$$

## Kurzschreibweise: Backus-Naur-Form (BNF)

$$A \rightarrow AB \mid CA \mid a \mid \varepsilon$$

$$B \rightarrow b \mid Bb$$

$$C \rightarrow A \mid cA \mid cc$$

$$AaA \rightarrow d$$

Egal wie man es schreibt: Es sind 10 Regeln!

# Ableitungen

## Grammatik $G$

$$\begin{array}{ccccc}
 A \xrightarrow{(A1)} AB & A \xrightarrow{(A2)} CA & A \xrightarrow{(A3)} a & A \xrightarrow{(A4)} \varepsilon & AaA \xrightarrow{(AaA)} d \\
 B \xrightarrow{(B1)} b & B \xrightarrow{(B2)} Bb & C \xrightarrow{(C1)} A & C \xrightarrow{(C2)} cA & C \xrightarrow{(C3)} cc
 \end{array}$$

## Ableitungen

- ▶ **Satzform**  $x \in (\mathcal{V} \cup \Sigma)^*$ :  
Kette von Symbolen und Variablen
- ▶ **Produktion**  $xyz \Rightarrow xy'z$ :  
 $x, y, z$  sind Satzformen und es gibt eine Regel  $y \rightarrow y'$
- ▶ **Ableitung**  $S \Rightarrow^* y$  („ $y$  ist ableitbar in  $G$ “):  
 $S$  ist die Startvariable,  $y$  eine Satzform, und es gibt eine Folge von Produktionen, sodass  $S \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \dots \Rightarrow y$ .
- ▶ **Sprache**  $\mathcal{L}(G)$ :  
Menge aller Wörter, die in  $G$  ableitbar sind.

# Ableitungen, Beispiele

## Grammatik $G$

$$\begin{array}{cccccc}
 A \xrightarrow{(A1)} AB & A \xrightarrow{(A2)} CA & A \xrightarrow{(A3)} a & A \xrightarrow{(A4)} \varepsilon & AaA \xrightarrow{(AaA)} d \\
 B \xrightarrow{(B1)} b & B \xrightarrow{(B2)} Bb & C \xrightarrow{(C1)} A & C \xrightarrow{(C2)} cA & C \xrightarrow{(C3)} cc
 \end{array}$$

## Ableitungen

$$\blacktriangleright A \xrightarrow{(A2)} CA \xrightarrow{(C3)} cCA \xrightarrow{(A3)} cca$$

Also:  $A \Rightarrow^* cca$ ,  $cca \in \mathcal{L}(G)$

- $\blacktriangleright$  Ein Wort kann i.A. unterschiedlich abgeleitet werden:

$$A \xrightarrow{(A2)} CA \xrightarrow{(A4)} C \xrightarrow{(C2)} cA \xrightarrow{(A2)} cCA \xrightarrow{(A4)} cC \xrightarrow{(C2)} cCA \xrightarrow{(A3)} cca$$

$$\blacktriangleright A \xrightarrow{(A1)} AB \xrightarrow{(A4)} B \xrightarrow{(B2)} Bb \xrightarrow{(B2)} Bbb \xrightarrow{(B2)} \dots \xrightarrow{(B2)} Bbb\dots b \xrightarrow{(B1)} bbb\dots b$$

$$\begin{array}{l}
 \blacktriangleright A \xrightarrow{(A2)} CA \xrightarrow{(A2)} CCA \xrightarrow{(C1)} CAA \xrightarrow{(C1)} AAA \xrightarrow{(A1)} AAAB \\
 \qquad \qquad \qquad \qquad \qquad \qquad \xrightarrow{(A3)} AaAB \xrightarrow{(AaA)} dB \xrightarrow{(B1)} db
 \end{array}$$

# Probleme auf Sprachen

## Grammatik $G$

$$\begin{array}{ccccc}
 A \xrightarrow{(A1)} AB & A \xrightarrow{(A2)} CA & A \xrightarrow{(A3)} a & A \xrightarrow{(A4)} \varepsilon & AaA \xrightarrow{(AaA)} d \\
 B \xrightarrow{(B1)} b & B \xrightarrow{(B2)} Bb & C \xrightarrow{(C1)} A & C \xrightarrow{(C2)} cA & C \xrightarrow{(C3)} cC
 \end{array}$$

**Wortproblem:** Gegeben ein Wort  $w$ . Gilt  $w \in \mathcal{L}(G)$ ?

- ▶ Falls „ja“: Finde Ableitung als Beweis.
- ▶ Falls „nein“: Wie zeigt man, dass gar keine Ableitung zu  $w$  existieren kann?

## Weitere Probleme

- ▶ **Leerheitsproblem.** Gilt  $\mathcal{L}(G) = \emptyset$ ? Oder kann in  $G$  mindestens ein Wort abgeleitet werden?
- ▶ **Endlichkeitsproblem.** Ist  $\mathcal{L}(G)$  endlich? Oder erlaubt  $G$  unendlich viele Wörter?

Die Schwierigkeit dieser Probleme ist abhängig davon, wie „kompliziert“ ( $\neq$  groß) die Grammatik ist!

# Chomsky-Hierarchie

Es existiert eine zugehörige Grammatik mit Einschränkungen für alle Regeln  $x \rightarrow y$ :

**Typ 0: Rekursiv aufzählbare Sprachen**

keine Einschränkungen

**Typ 1: Kontextsensitive (bzw. monotone) Sprachen**

$$|x| \leq |y|$$

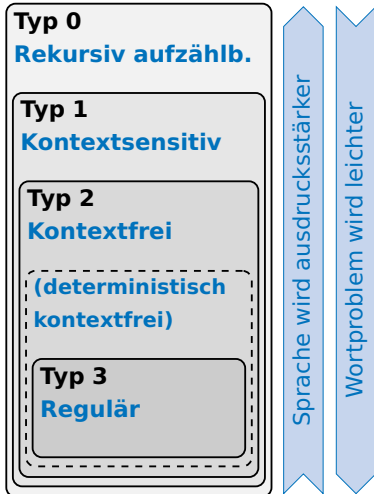
**Typ 2: Kontextfreie Sprachen**

Typ 1, und  $x \in \mathcal{V}$  (links eine **einzelne** Variable)

**Typ 3: Reguläre Sprachen**

$$x \in \mathcal{V}, y \in \Sigma \cup \Sigma \mathcal{V}$$

(rechts **ein** Symbol, ggf. gefolgt von **einer** Variable)



## Chomsky und das Leerwort

Das **Leerwort** (Wort der Länge 0) wird durch  $\varepsilon$  bezeichnet.

**Problem** (naja...)

Regeln  $x \rightarrow \varepsilon$  ab Typ 1 nicht zugelassen, da  $|x| \not\leq 0$ .

$\Rightarrow$  Typ-1,2,3 Sprachen könnten kein Leerwort erzeugen.

Daher:

**Sonderregel**

Die Regel  $S \rightarrow \varepsilon$  (für Startvariable  $S$ ) ist zugelassen, falls  $S$  nirgends auf der rechten Seite der anderen Regeln vorkommt.

**Beobachtung**

Man kann jede Grammatik  $G = (\Sigma, \mathcal{V}, S, \mathcal{R})$  (Startvariable  $S$ , Regeln  $\mathcal{R}$ ) so erweitern, dass sie mit obiger Sonderregel das Leerwort erlaubt:

$$G' = (\Sigma, \mathcal{V} \cup \{S'\}, S', \mathcal{R} \cup \{S' \rightarrow \varepsilon, S' \rightarrow S\})$$

# Probleme vs. Maschinen

Sprache d. Wortproblems	Lösbar(?) mittels	Komplexität (Laufzeit/Speicher)
<b>Rekursiv aufzählbar</b>	Turingmaschine ( $\approx$ Computer)	nicht entscheidbar
<b>Kontextsensitiv</b>	linear beschränkter Automat	exponentiell/linear
<b>Kontextfrei</b>	Kellerautomat	kubisch/quadratisch
<b>Deterministisch Kontextfrei</b>	deterministischer Kellerautomat	linear/linear
<b>Regulär</b>	Endlicher Automat	linear/konstant

# **Formale Sprachen & Automaten**

Reguläre Ausdrücke

# Reguläre Sprachen

$$A \rightarrow aA \mid bA \mid cA \mid dA \mid aB$$

$$B \rightarrow cC$$

$$C \rightarrow dD$$

$$D \rightarrow c$$

## Welche Wörter generiert diese Grammatik?

Alle Wörter über dem Alphabet  $\Sigma = \{a, b, c, d\}$  die mit dem Teilwort „*acdc*“ enden.

„Hübsch & schnell verständlich“ ist obere Schreibweise nicht...

## Regulärer Ausdruck, engl. **Regular Expression**, kurz **RegEx**

Jede reguläre Sprache lässt sich durch einen regulären Ausdruck darstellen.

- ▶ Textsuchen (in vernünftigen Texteditoren, mit `grep`,...), Filtern von Texten, Suchen von Dateien (`ls ab*`,...),...
- ▶ Compiler-Bau: Lexikalische Analyse, Scanner-Generatoren
- ▶ Software-Protokolle, Steuerungs- und Kontrollsysteme,...

Entwickelt von

**Stephen Cole Kleene** (1909–1994)

(Aussprache in etwa: Kleïni)

- ▶ Einer der Begründer der theor. Informatik,
- ▶ Begründer der **Rekursionstheorie** und (mit A. Church) des **Lambda-Kalküls** (funktionale Programmierung); Grundlagen der **Berechenbarkeitstheorie**



# Regulärer Ausdruck, Rekursive Definition

**Definition.** Sei  $E$  ein regulärer Ausdruck. Wir bezeichnen die Sprache, die von  $E$  beschrieben wird, als  $\mathcal{L}(E)$ .

## Rekursive Definition, Basisfälle

- ▶ Die leere Menge  $\emptyset$  ist ein regulärer Ausdruck.

**Achtung:**  $\emptyset \neq \varepsilon!$   $\emptyset$  ist die Sprache **ohne** Wörter;  $\varepsilon$  ist ein Wort, nur eben mit Länge 0.

$$\Rightarrow \mathcal{L}(\emptyset) := \emptyset$$

- ▶ Symbole  $\sigma \in \Sigma$  sind reguläre Ausdrücke.

$$\Rightarrow \mathcal{L}(\sigma) := \{\sigma\}$$

# Regulärer Ausdruck, Rekursive Definition

## Rekursive Definition, Rekursionen

Seien  $E_1$  und  $E_2$  zwei reguläre Ausdrücke, dann sind auch die folgenden Ausdrücke regulär:

▶ **Alternative** ( $E_1|E_2$ ):

Entweder ein Wort aus  $E_1$  oder eines aus  $E_2$ .

$$\Rightarrow \mathcal{L}(E_1|E_2) := \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$$

▶ **Verkettung** ( $E_1 E_2$ ):

Alle Wörter die durch Aneinanderhängen eines Worts aus  $E_1$  und eines aus  $E_2$  entstehen.

$$\Rightarrow \mathcal{L}(E_1 E_2) := \{uw \mid u \in \mathcal{L}(E_1), w \in \mathcal{L}(E_2)\}$$

▶ **Kleene-Stern (Kleenesche Hülle)** ( $E_1^*$ ):

beliebig-viele beliebige Wörter aus  $E_1$  aneinander gekettet, ggf. auch 0.

$$\Rightarrow \mathcal{L}(E_1^*) := \mathcal{L}(E_1)^*$$

## Regulärer Ausdruck, Aufhübschen

Regulärer Ausdruck für die vorige Beispielsprache:

$$((((((a|(b|(c|d))))^*)a)c)d)c$$

**Klammern** dienen nur zur **eindeutigen Strukturierung**:

- ▶ Für Alternative und Verkettung gilt das Assoziativgesetz:

$$(a|b)|c = a|(b|c) = a|b|c$$

$$(ab)c = a(bc) = abc$$

- ▶ Operator-Rangfolge analog zu algebraischen Ausdrücken:

$E_1^*$  („Potenzieren“) bindet stärker als

$E_1 E_2$  („Multiplikation“) bindet stärker als

$E_1|E_2$  („Addition“)

$$\Rightarrow (a|b|c|d)^*acdc$$

## Regulärer Ausdruck, Praktische Ergänzungen

- ▶ **Positive Hülle**  $(E_1)^+$  ist eine Kurzform für  $(E_1 (E_1)^*)$ , d.h.  $E_1$  kommt beliebig oft, aber mindestens einmal, vor.
- ▶ Kann ein regulärer Ausdruck das **Leerwort** (Wort der Länge 0) beschreiben? **Ja:**  $\emptyset^*$

**Begründung:**  $\emptyset$  ist die Sprache ohne Wörter. Wir schreiben jetzt beliebig oft, auch 0-mal, etwas aus dieser leeren Menge auf. Da die Menge leer ist, können wir nichts aufschreiben, also nicht „beliebig oft“, sondern nur 0-mal. Irgendetwas (auch nichts) 0-mal hinzuschreiben ergibt das Leerwort!

Daher schreibt man meist direkt  $\varepsilon$  als Kurzform für  $\emptyset^*$ .

# Regulärer Sprachen vs. Reguläre Ausdrücke

## Theorem.

Jeder reguläre Ausdruck beschreibt eine reguläre Sprache.  
Jede reguläre Sprache lässt sich als regulärer Ausdruck schreiben.

**Formal:** Sei  $E$  ein regulärer Ausdruck, dann ist  $\mathcal{L}(E)$  regulär.  
Sei  $L'$  eine reguläre Sprache, dann existiert ein regulärer Ausdruck  $E'$  mit  $L' = \mathcal{L}(E')$ .

**Beweis.** Werden wir später noch sehen!

# **Formale Sprachen & Automaten**

Endliche Automaten

## Wortproblem: Sprache $\rightarrow$ Automat

**Reguläre Grammatik  $G$ :**

$$A \rightarrow aA \mid bA \mid cA \mid dA \mid aB$$

$$B \rightarrow cC$$

$$C \rightarrow dD$$

$$D \rightarrow c$$

**Regulärer Ausdruck  $E$ :**

$$(a|b|c|d)^*acdc$$

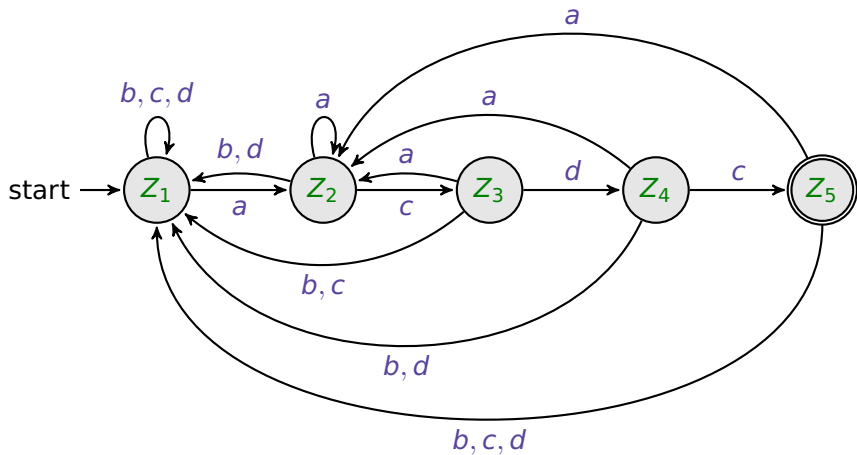
Gegeben  $G$  oder  $E$  und ein Wort  $w$  (z.B.  $w := aacddcaacdc$ ).  
Wie kann man **algorithmisch** feststellen, ob  $w \in \mathcal{L}(G)$  bzw.  
 $w \in \mathcal{L}(E)$ ?

**Knifflig...** alle Möglichkeiten der Expansion durchprobieren?

$\rightarrow$  Gibt es eine andere, zur Lösung des Wortproblems  
geschicktere, Darstellung der Sprache  $\mathcal{L}(G) = \mathcal{L}(E)$ ?

$\rightarrow$  **Endliche Automaten**

## Endlicher Automat (nur kurz)



## Aufbau eines endlichen Automaten

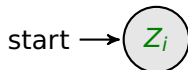
Ein (deterministischer) **endlicher Automat (EA)** (engl. **Finite State Machine, Finite Automaton**) besteht aus:

- ▶ Endliche Menge von **Zuständen**

$$\mathcal{Z} = \{Z_1, Z_2, \dots, Z_k\}$$



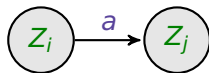
- ▶ Ein **Startzustand**  $Z_{\text{start}} \in \mathcal{Z}$



- ▶ Mehrere **Endzustände**  $\mathcal{Z}_{\text{end}} \subseteq \mathcal{Z}$



- ▶ **Übergangsfunktion**  $\delta: \mathcal{Z} \times \Sigma \rightarrow \mathcal{Z}$   
von einem Zustand in einen anderen,  
abhängig von einem Symbol.



Funktion ist **partiell**, d.h. ggf. „fehlt“ ein Übergang.

# Ablauf eines endlichen Automaten

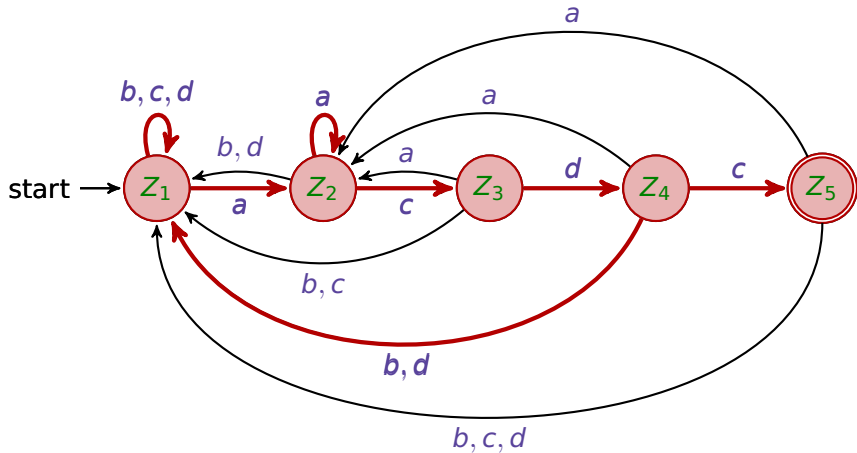
## Endlicher Automat $\mathcal{A}$ :

- ▶ **Zuständen**  $Z = \{Z_1, Z_2, \dots, Z_k\}$
- ▶ **Startzustand**  $Z_{\text{start}} \in Z$
- ▶ **Endzustände**  $Z_{\text{end}} \subseteq Z$
- ▶ (Partielle) **Übergangsfunktion**  $\delta: Z \times \Sigma \rightarrow Z$

## Ablauf

- ▶ Man befindet sich immer in einem Zustand; anfangs  $Z_{\text{start}}$ .
- ▶ Lese Eingabewort  $w$  zeichenweise von links nach rechts.
- ▶ Sei  $\sigma \in \Sigma$  der gerade gelesene Buchstabe. Man bewegt sich vom aktuellen Zustand  $Z'$  zu dem Zustand  $\delta(Z', \sigma)$ . Falls nicht definiert: Abbruch.
- ▶ Genau dann, wenn man sich (ohne Abbruch) am Ende des Wortes in einem Endzustand befindet:  $\mathcal{A}$  **akzeptiert**  $w$ , d.h.  $w \in \mathcal{L}(\mathcal{A})$ .

## Endlicher Automat (jetzt wirklich)



$w = aacddcaacdc$

# Deterministischer endlicher Automat

## Duden

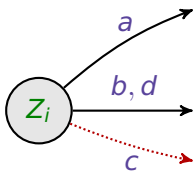
**deterministisch:** Adjektiv – 1. den Determinismus betreffend; 2. [Willens]freiheit verneinend

**Determinismus:** Substantiv, maskulin – Lehre, Auffassung von der kausalen [Vor]bestimmtheit allen Geschehens bzw. Handelns

Bisher: **deterministische endliche Automaten (DEA):**

- ▶ An jedem Zustand gibt es pro Symbol **maximal einen** möglichen Übergang.

→ Das Ergebnis (und der Weg dorthin) ist, für ein gegebenes Wort  $w$ , eindeutig bestimmt.



# Indeterministischer endlicher Automat

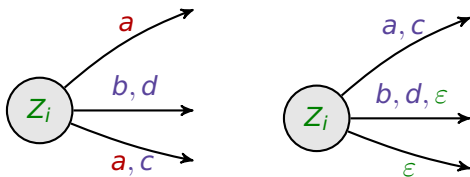
## *Duden*

**indeterminiert:** Adjektiv – unbestimmt, nicht festgelegt (abgegrenzt), frei

## Nicht-deterministischer endliche Automaten (NDEA):

- ▶ An jedem Zustand kann es pro Symbol **mehrere** mögliche Übergänge geben.
- ▶ Sogar das Leerwort  $\epsilon$  ist eine zulässiges Übergangslabel.

→ Das Ergebnis (und der Weg dort hin) ist, für ein gegebenes Wort  $w$ , **nicht** eindeutig bestimmt.



## Deterministisch vs. Indeterministisch

Wir wissen schon:

**Definition.** Sei  $\mathcal{A}_d$  ein **deterministischer** endlicher Automat. Die Sprache  $\mathcal{L}(\mathcal{A}_d)$  ist die Menge der Wörter, für die sich  $\mathcal{A}_d$  nach Lesen des Worts in einem Endzustand befindet.

Nun:

**Definition.** Sei  $\mathcal{A}_n$  ein **nicht-deterministischer** endlicher Automat. Die Sprache  $\mathcal{L}(\mathcal{A}_n)$  ist die Menge der Wörter, für die sich  $\mathcal{A}_n$  nach Lesen des Worts in einem Endzustand befinden **kann**.

**D.h.:** Es gibt eine Möglichkeit mittels der Übergangsregeln für das gegebene Wort in einen Endzustand zu gelangen. Aber: Nicht jede zulässige Abarbeitung muss in einem Endzustand enden!

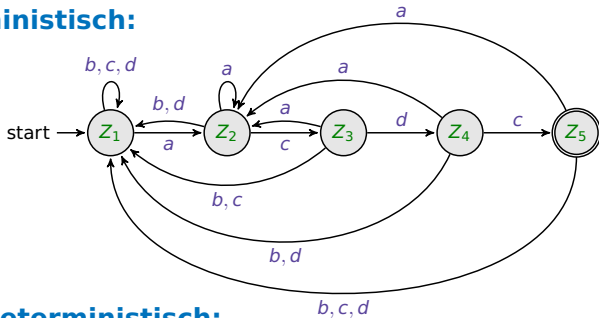
Was soll das helfen??

## Vorteil des Nicht-Determinismus

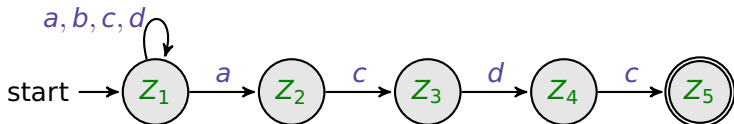
Die Automaten sind oft leichter hinzuschreiben bzw. zu lesen.

$$(a|b|c|d)^*acdc$$

### Deterministisch:



### Nicht-deterministisch:



# **Formale Sprachen & Automaten**

Beschreibungsäquivalenz regulärer  
Sprachen

## Alles das gleiche!

**Theorem.** Reguläre Grammatiken, reguläre Ausdrücke, DEAs und NDEAs erlauben alle **genau** die selben Sprachen zu beschreiben (nämlich die regulären Sprachen).

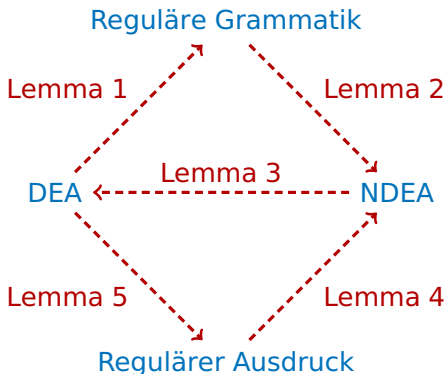
### Beweis.

Ringschluss in 5 Lemmata:

Seien  $X$  und  $Y$  jew. eine der vier Beschreibungsarten.

**Lemma  $X \rightarrow Y$ :** Nimm eine beliebige Instanz der Art  $X$ . Wir erstellen eine Instanz der Art  $Y$ , die die selbe Sprache beschreibt.

$\Rightarrow Y$  kann mind. so viele Sprachen beschreiben wie  $X$ .



## Alles das gleiche!

**Theorem.** Reguläre Grammatiken, reguläre Ausdrücke, DEAs und NDEAs erlauben alle **genau** die selben Sprachen zu beschreiben (nämlich die regulären Sprachen).

### Beweis.

Ringschluss in 5 Lemmata:

Seien  $X$  und  $Y$  jew. eine der vier Beschreibungsarten.

**Lemma**  $X \rightarrow Y$ : Nimm eine beliebige Instanz der Art  $X$ .

Wir ers

der Art

Sprach

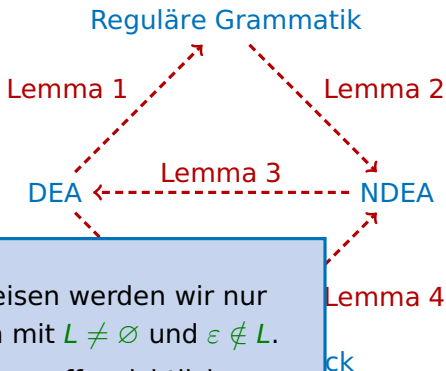
$\Rightarrow Y$  ka

Sprach

### Achtung:

In den folgenden Beweisen werden wir nur Sprachen  $L$  betrachten mit  $L \neq \emptyset$  und  $\varepsilon \notin L$ .

Diese Sonderfälle können offensichtlich von allen Beschreibungsmodellen erzielt werden.



## Lemma 1: DEA $\rightarrow$ reguläre Grammatik

### Beweis.

**Reguläre Grammatik:** Alle Regeln haben die Gestalt  $V \rightarrow \sigma V'$  oder  $V \rightarrow \sigma$ , für irgendwelche  $V, V' \in \mathcal{V}, \sigma \in \Sigma$ .

**Beobachtung:** Jede in einer Ableitung vorkommende Satzform hat die Form  $wV$ , wobei  $w \in \Sigma^*$  ein Wort ist, und nur am Ende genau eine Variable  $V \in \mathcal{V}$  vorkommt.

### Erstellen einer reg. Grammatik auf Basis eines DEA:

- ▶ Eine Variable  $V_i$  für jeden Zustand  $Z_i \in \mathcal{Z}$ .  
Die Variable des Startzustands ist die Startvariable.
- ▶ Für jeden Übergang  $\delta(Z_i, \sigma) = Z_j$  (mit  $Z_i, Z_j \in \mathcal{Z}, \sigma \in \Sigma$ ):  
Erstelle Regel  $V_i \rightarrow \sigma V_j$ .  
Falls  $Z_j$  dabei ein Endzustand ist:  
Erstelle auch noch die Regel  $V_i \rightarrow \sigma$ .



## Alles das gleiche!

**Theorem.** Reguläre Grammatiken, reguläre Ausdrücke, DEAs und NDEAs erlauben alle **genau** die selben Sprachen zu beschreiben (nämlich die regulären Sprachen).

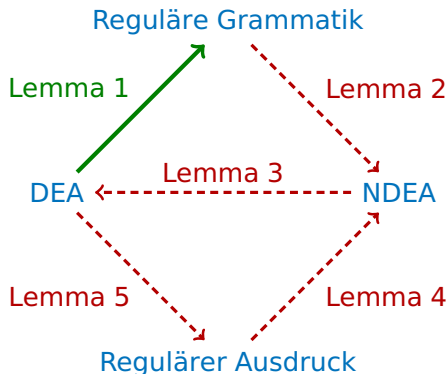
### Beweis.

Ringschluss in 5 Lemmata:

Seien  $X$  und  $Y$  jew. eine der vier Beschreibungsarten.

**Lemma  $X \rightarrow Y$ :** Nimm eine beliebige Instanz der Art  $X$ . Wir erstellen eine Instanz der Art  $Y$ , die die selbe Sprache beschreibt.

$\Rightarrow Y$  kann mind. so viele Sprachen beschreiben wie  $X$ .



## Lemma 2: Reguläre Grammatik $\rightarrow$ NDEA

### Beweis.

Im Wesentlichen die Rückwärtsrichtung des vorigen Beweises („DEA  $\rightarrow$  reguläre Grammatik“). Aber: der erstellte EA ist **nicht** deterministisch!

### Erstellen eines NDEA auf Basis einer reg. Grammatik:

- ▶ Ein Zustand  $Z_V$  für jede Variable  $V \in \mathcal{V}$ .  
Der Zustand der Startvariable ist der Startzustand.  
Ein weiterer Zustand  $Z_{\text{end}}$  als Endzustand.
- ▶ Für jede Regel der Gestalt  $V \rightarrow \sigma V'$  ( $V, V' \in \mathcal{V}, \sigma \in \Sigma$ ):  
Übergangskante von  $Z_V$  nach  $Z_{V'}$  mit Beschriftung  $\sigma$ .
- ▶ Für jede Regel der Gestalt  $V \rightarrow \sigma$  ( $V \in \mathcal{V}, \sigma \in \Sigma$ ):  
Übergangskante von  $Z_V$  nach  $Z_{\text{end}}$  mit Beschriftung  $\sigma$ .  $\square$

## Alles das gleiche!

**Theorem.** Reguläre Grammatiken, reguläre Ausdrücke, DEAs und NDEAs erlauben alle **genau** die selben Sprachen zu beschreiben (nämlich die regulären Sprachen).

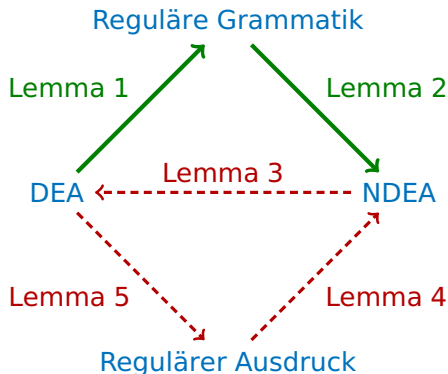
### Beweis.

Ringschluss in 5 Lemmata:

Seien  $X$  und  $Y$  jew. eine der vier Beschreibungsarten.

**Lemma  $X \rightarrow Y$ :** Nimm eine beliebige Instanz der Art  $X$ . Wir erstellen eine Instanz der Art  $Y$ , die die selbe Sprache beschreibt.

$\Rightarrow Y$  kann mind. so viele Sprachen beschreiben wie  $X$ .



## Lemma 3: NDEA $\rightarrow$ DEA

### Beweis.

#### Vorgedanken:

- ▶ NDEA: Zustände  $Z$ , Startzustand  $Z_{\text{start}}$ , Endzustände  $Z_{\text{end}}$
- ▶ Nach jedem Lesen eines Symbols kann der NDEA in einem von mehreren verschiedenen Zuständen sein.  
 $\rightarrow$  „aktive“ Zustandsmenge  $\subseteq Z$ .
- ▶ Potenzmenge  $2^Z =$  alle möglichen Teilmengen von  $Z$   
 $=$  alle möglichen aktiven Zustandsmengen.
- ▶  $Z$  endlich  $\Rightarrow 2^Z$  endlich

## Lemma 3: NDEA $\rightarrow$ DEA

### Beweis.

- ▶ NDEA: Zustände  $Z$ , Startzustand  $Z_{\text{start}}$ , Endzustände  $Z_{\text{end}}$

### Erstelle einen DEA auf Basis des NDEA:

- ▶ Ein Zustand  $Y_S$  für jede Teilmenge  $\emptyset \neq S \in 2^Z$ .  
(Erstelle nur Zustände  $Y_S$ , die im Folgenden auftauchen!)
- ▶ Sei  $S_{\text{start}} \in 2^Z$  die Menge aller Zustände, die man von  $Z_{\text{start}}$  mittels beliebig vieler (auch 0)  $\varepsilon$ -Übergänge erreichen kann.  $\Rightarrow Y_{S_{\text{start}}}$  ist Startzustand des DEAs.
- ▶  $\sigma$ -Übergang ( $\forall \sigma \in \Sigma$ ) von  $Y_S$  ( $\forall$  auftauchenden Zustände  $Y_S$ ) nach  $Y_{S'}$  **genau dann wenn**:  $S'$  enthält **genau alle** Zustände, die ausgehend von irgendeinem Zustand aus  $S$  mittels einem  $\sigma$ -Übergang, gefolgt von beliebig vielen (auch 0)  $\varepsilon$ -Übergängen, erreicht werden können.
- ▶ Ein Zustand  $Y_S$  ist ein Endzustand **genau dann wenn**  $S \cap Z_{\text{end}} \neq \emptyset$ .



## Alles das gleiche!

**Theorem.** Reguläre Grammatiken, reguläre Ausdrücke, DEAs und NDEAs erlauben alle **genau** die selben Sprachen zu beschreiben (nämlich die regulären Sprachen).

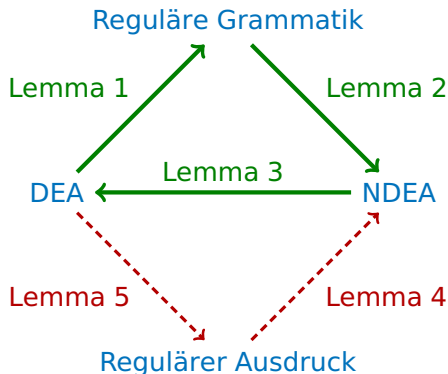
### Beweis.

Ringschluss in 5 Lemmata:

Seien  $X$  und  $Y$  jew. eine der vier Beschreibungsarten.

**Lemma  $X \rightarrow Y$ :** Nimm eine beliebige Instanz der Art  $X$ . Wir erstellen eine Instanz der Art  $Y$ , die die selbe Sprache beschreibt.

$\Rightarrow Y$  kann mind. so viele Sprachen beschreiben wie  $X$ .



## Lemma 4: Regulärer Ausdruck $\rightarrow$ NDEA

**Beweis. Erstelle NDEA auf Basis einer RegEx.**

Induktion über den rekursiven Aufbau.

**Induktionsanfang:**

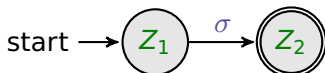
- ▶ **Die leere Menge  $\emptyset$  ist ein regulärer Ausdruck.**

NDEA mit einem Start- und einem Endzustand, aber ohne Übergang dazwischen.



- ▶ **Symbole  $\sigma \in \Sigma$  sind reguläre Ausdrücke.**

NDEA mit einem Start- und einem Endzustand, und einem  $\sigma$ -Übergang dazwischen.



## Lemma 4: Regulärer Ausdruck $\rightarrow$ NDEA

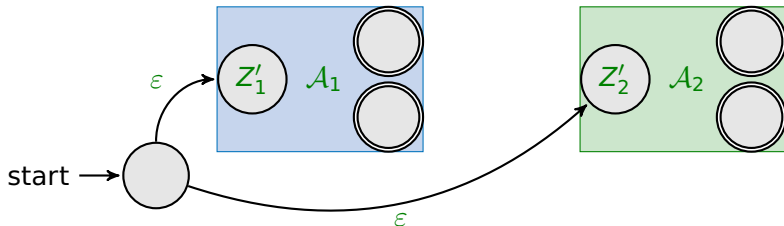
**Beweis. Erstelle NDEA auf Basis einer RegEx.**

Induktion über den rekursiven Aufbau.

**Induktionshypothese:** Sei  $E_i$  ( $i = 1, 2$ ) ein regulärer Ausdruck. Es existiert ein  $\mathcal{L}(E_i)$ -akzeptierender NDEA  $\mathcal{A}_i$  mit Startzustand  $Z'_i$  und Endzuständen  $Z''_i$ .

**Induktionsschritt (1/3):**

- ▶ **Alternative ( $E_1|E_2$ ):** NDEA  $\mathcal{A}_1 \cup \mathcal{A}_2$  inklusive neuem Startzustand, der  $\varepsilon$ -Übergänge zu  $Z'_1$  und  $Z'_2$  hat.



## Lemma 4: Regulärer Ausdruck $\rightarrow$ NDEA

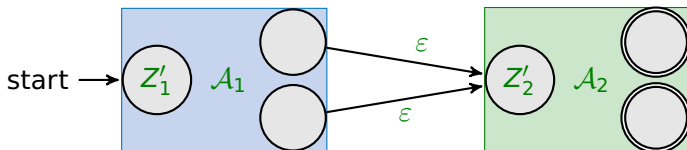
**Beweis. Erstelle NDEA auf Basis einer RegEx.**

Induktion über den rekursiven Aufbau.

**Induktionshypothese:** Sei  $E_i$  ( $i = 1, 2$ ) ein regulärer Ausdruck. Es existiert ein  $\mathcal{L}(E_i)$ -akzeptierender NDEA  $\mathcal{A}_i$  mit Startzustand  $Z'_i$  und Endzuständen  $Z''_i$ .

**Induktionsschritt (2/3):**

- **Verkettung ( $E_1 E_2$ ):** NDEA  $\mathcal{A}_1 \cup \mathcal{A}_2$  mit Startzustand  $Z'_1$ , Endzuständen  $Z''_2$ , und  $\varepsilon$ -Übergängen von jedem Zustand aus  $Z'_1$  zu  $Z'_2$ .



## Lemma 4: Regulärer Ausdruck $\rightarrow$ NDEA

**Beweis. Erstelle NDEA auf Basis einer RegEx.**

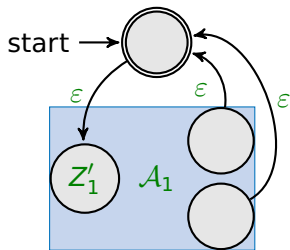
Induktion über den rekursiven Aufbau.

**Induktionshypothese:** Sei  $E_i$  ( $i = 1, 2$ ) ein regulärer Ausdruck. Es existiert ein  $\mathcal{L}(E_i)$ -akzeptierender NDEA  $\mathcal{A}_i$  mit Startzustand  $Z'_i$  und Endzuständen  $Z''_i$ .

**Induktionsschritt (3/3):**

► **Kleene-Stern ( $E_1$ )<sup>\*</sup>:**

NDEA, basierend auf  $\mathcal{A}_1$ , mit neuem Startzustand (der gleichzeitig der einzige Endzustand ist), einem  $\varepsilon$ -Übergang von diesem zu  $Z'_1$ , und  $\varepsilon$ -Übergängen von jedem Zustand aus  $Z'_1$  zum neuen Startzustand.  $\square$



## Alles das gleiche!

**Theorem.** Reguläre Grammatiken, reguläre Ausdrücke, DEAs und NDEAs erlauben alle **genau** die selben Sprachen zu beschreiben (nämlich die regulären Sprachen).

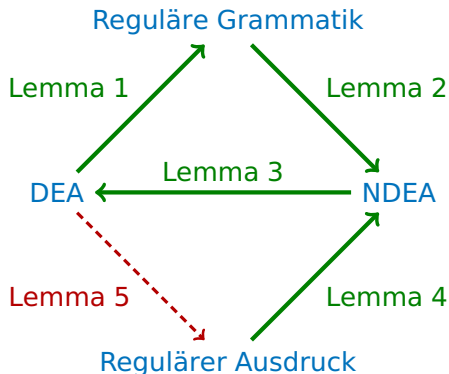
### Beweis.

Ringschluss in 5 Lemmata:

Seien  $X$  und  $Y$  jew. eine der vier Beschreibungsarten.

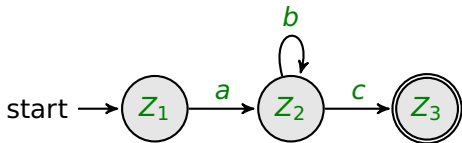
**Lemma  $X \rightarrow Y$ :** Nimm eine beliebige Instanz der Art  $X$ . Wir erstellen eine Instanz der Art  $Y$ , die die selbe Sprache beschreibt.

$\Rightarrow Y$  kann mind. so viele Sprachen beschreiben wie  $X$ .



## Lemma 5: DEA $\rightarrow$ Regulärer Ausdruck

### Beispiele für einfache Automaten:



Regex:  $a b^* c$



Regex:  $a^* b c^* (d a^* b c^*)^*$

## Lemma 5: DEA $\rightarrow$ Regulärer Ausdruck

**Beweis.** Verwandle den DEA schrittweise in einen Automaten, von dem man die RegEx direkt ablesen kann.

**1** Neuer Zustand  $Z_e$ ;  $\varepsilon$ -Übergänge von jedem  $Z \in Z_{\text{end}}$  nach  $Z_e$ ; neues  $Z_{\text{end}} := \{Z_e\}$ .  $\Rightarrow$  **ein** Anfangszustand ( $Z_s$ ) und **ein** Endzustand; lese Übergangsbeschriftung als RegEx.

**2** Solange  $\exists Z \in Z \setminus \{Z_s, Z_e\}$ :

Seien  $Z_{\text{in}}/Z_{\text{out}}$  die Zustände ( $\neq Z$ ) mit Überg. nach/von  $Z$ .

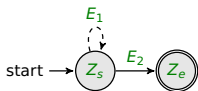
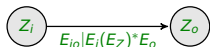
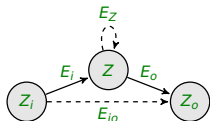
Für **jedes Paar**  $(Z_i, Z_o) \in Z_{\text{in}} \times Z_{\text{out}}$ :

► Seien  $E_i, E_Z, E_o, E_{io}$  die RegEx an den Übergängen  $Z_i \rightarrow Z$ ,  $Z \rightarrow Z$ ,  $Z \rightarrow Z_o$ ,  $Z_i \rightarrow Z_o$  (so sie  $\exists$ ).

► Setze  $E_{io} := E_{io} | E_i (E_Z)^* E_o$ .

Entferne  $Z$ .

**3**  $Z_e$  bekommt nie eine ausgehende Kante!  
 $\rightarrow$  Es bleibt EA mit RegEx  $(E_1)^* E_2$



## Alles das gleiche!

**Theorem.** Reguläre Grammatiken, reguläre Ausdrücke, DEAs und NDEAs erlauben alle **genau** die selben Sprachen zu beschreiben (nämlich die regulären Sprachen).

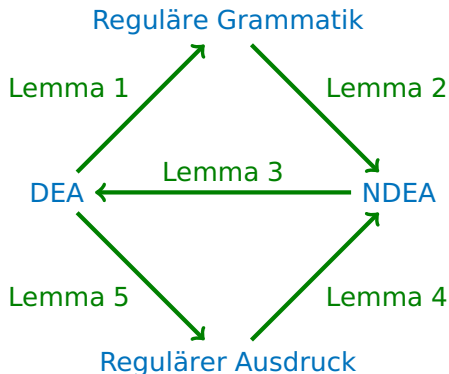
### Beweis.

Ringschluss in 5 **Lemmata**:

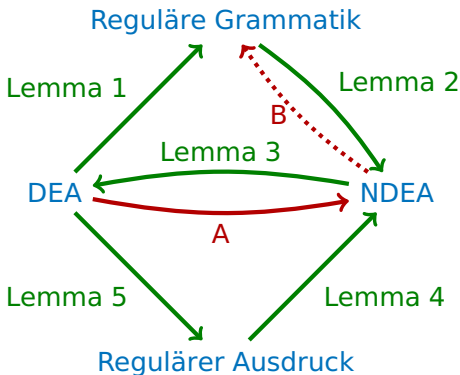
Seien  $X$  und  $Y$  jew. eine der vier Beschreibungsarten.

**Lemma  $X \rightarrow Y$ :** Nimm eine beliebige Instanz der Art  $X$ . Wir erstellen eine Instanz der Art  $Y$ , die die selbe Sprache beschreibt.

$\Rightarrow Y$  kann mind. so viele Sprachen beschreiben wie  $X$ .



## Apropos...



**A** Ist trivial, da  $DEA \subset NDEA$ .

**B** Kann man für NDEAen **die keine  $\epsilon$ -Übergänge enthalten** analog zu Lemma 1 machen.

## Beispiel – Sprache

► **Reg. Ausdruck:**

$$c (ab \mid aba)^*$$

► **Reg. Grammatik:**

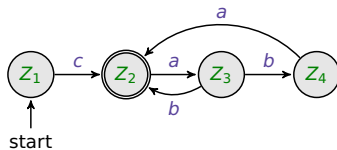
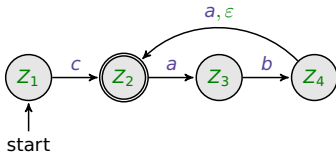
$$S \rightarrow c \mid cA$$

$$A \rightarrow aB$$

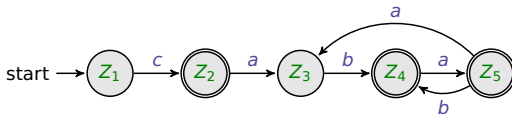
$$B \rightarrow b \mid bA \mid bC$$

$$C \rightarrow a \mid aA$$

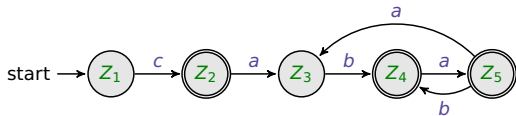
► **NDEA** (mit und ohne  $\varepsilon$ )



► **DEA**



# Beispiel – Lemma 1: DEA $\rightarrow$ Reg. Grammatik



Durch den Beweis generiert:

$$A_1 \rightarrow cA_2 \mid c$$

$$A_2 \rightarrow aA_3$$

$$A_3 \rightarrow bA_4 \mid b$$

$$A_4 \rightarrow aA_5 \mid a$$

$$A_5 \rightarrow bA_4 \mid b \mid aA_3$$

Handgestrickt:

$$S \rightarrow c \mid cA$$

$$A \rightarrow aB$$

$$B \rightarrow b \mid bA \mid bC$$

$$C \rightarrow a \mid aA$$

## Beispiel – Lemma 2: Reg. Grammatik $\rightarrow$ NDEA

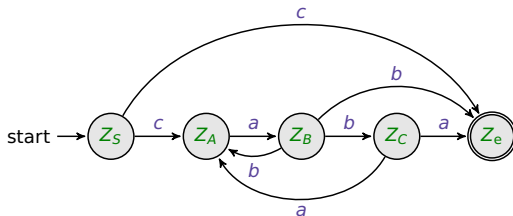
$$S \rightarrow c \mid cA$$

$$A \rightarrow aB$$

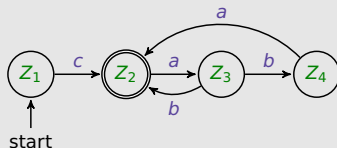
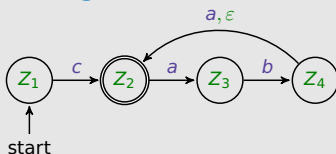
$$B \rightarrow b \mid bA \mid bC$$

$$C \rightarrow a \mid aA$$

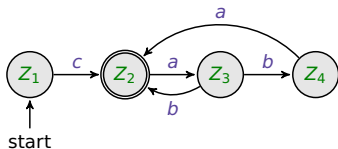
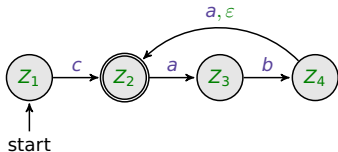
Durch den Beweis generiert:



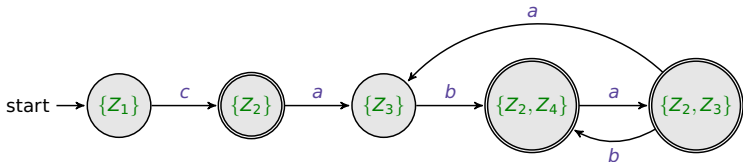
Handgestrickt:



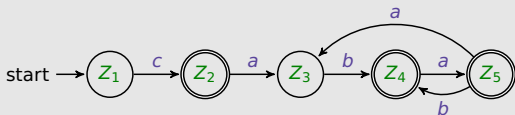
## Beispiel – Lemma 3: NDEA $\rightarrow$ DEA



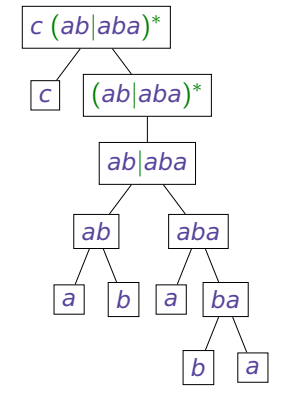
Durch den Beweis generiert:



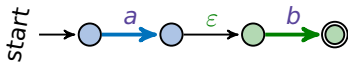
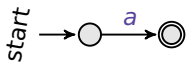
Handgestrickt:



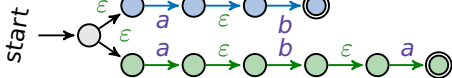
# Beispiel – Lemma 4: RegEx $\rightarrow$ NDEA



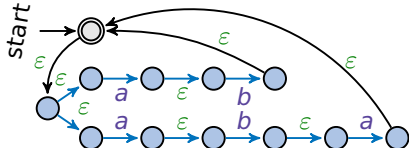
$a$ : ( $b, c$  analog)     $ab$ : ( $aba$  analog)



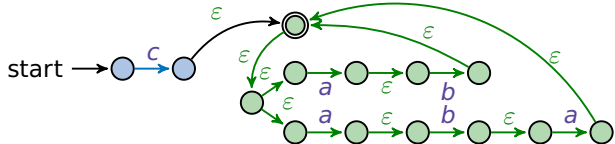
$ab|aba$ :



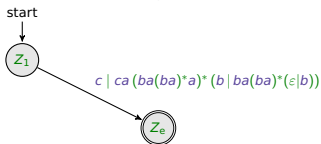
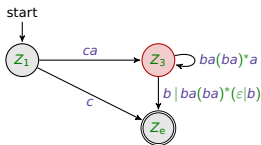
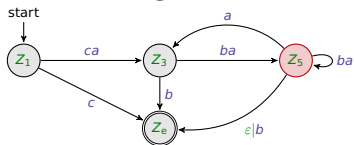
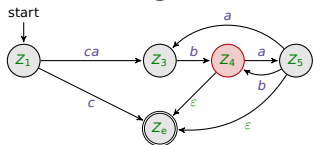
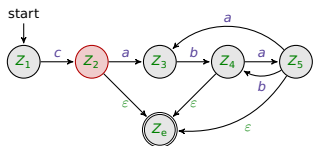
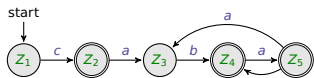
$(ab|aba)^*$ :



$c(ab|aba)^*$ :



# Beispiel – Lemma 5: DEA $\rightarrow$ RegEx



RegEx, durch den Beweis:

$c \mid ca (ba(ba)^*a)^* (b \mid ba(ba)^*(\epsilon \mid b))$

RegEx, handgestrickt:

$c(ab|aba)^*$

# **Formale Sprachen & Automaten**

Reguläres Pumping Lemma und  
Abschlusseigenschaften

# Nicht-Regularität

**Gegeben:** Eine Sprache  $L$ .

**Frage:** Ist  $L$  regulär?

**Falls ja:**

Beweis durch reg. Grammatik, reg. Ausdruck, DEA oder NDEA, die/der  $L$  beschreibt.

**Falls nein:**

Wie beweist man, dass eine Sprache **nicht** regulär ist?

- ▶ **Pumping Lemma** → jetzt
- ▶ Abschlusseigenschaften
- ▶ (...)

## Pumping Lemma

### Pumping Lemma (für reguläre Sprachen).

Sei  $L$  eine reguläre Sprache. Es gibt eine Zahl  $n := n(L)$  (d.h. in Abhängigkeit von  $L$ ), so dass alle Wörter  $z \in L$  mit  $|z| \geq n$  sich zerlegen lassen als  $z = uvw$  mit den Eigenschaften:

- 1  $|v| \geq 1$ ,     
 2  $|uv| \leq n$ ,     
 3  $uv^*w \in L$ .

**Beweis.**  $L \rightarrow \exists$  DEA  $\mathcal{A}$  mit Zuständen  $\mathcal{Z}$ . Wähle  $n := |\mathcal{Z}|$ .

Bei  $\mathcal{A}$ -Abarbeitung eines Wortes  $z$  werden  $|z| + 1$  Zustände abgelaufen (inkl. Startzustand). Da  $|z| \geq n$ : mindestens ein Zustand wird öfters (mind. 2x) besucht.

Wähle Zerlegung  $z = uvw$  so, dass man nach Lesen des letzten Symbols von  $u$  und von  $uv$  im gleichen Zustand ( $Z_i$ ) ist. Dabei ist es trivial, 1 und 2 zu erfüllen.

Nun könnte man von  $Z_i$  aus auch mehrmals (inkl. 0-mal)  $v$  ablaufen, bevor man  $w$  ablauft  $\rightarrow uv^*w \in L \rightarrow$  3. □

## Anwendung des Pumping Lemmas

$\forall$  reg. Spr.  $L$ :  $\exists n$ :  $\forall z \in L$  mit  $|z| \geq n$ :  $\exists uvw = z$  mit:

**1**  $|v| \geq 1$ ,

**2**  $|uv| \leq n$ ,

**3**  $uv^*w \in L$ .

**Aufgabe:** Sei  $L = \{a^i b^i \mid i \geq 0\}$  die Sprache der Wörter, deren vordere Hälfte lauter  $a$  und deren hintere Hälfte lauter  $b$  sind. Zeige, dass  $L$  **nicht** regulär ist.

**Lösung: Beweis durch Widerspruch.**

Nimm an,  $L$  wäre regulär, dann würde für  $L$  das Pumping Lemma gelten. Sei  $n$  die entsprechende Wortmindestgröße.

Betrachte  $z = a^n b^n \in L$ . Wähle **ein** Wort als Gegenbeispiel!

- ▶  $\exists$  Zerlegung  $z = uvw$  mit **1** – **3**.

Argumentation über **alle** möglichen Zerlegungen!

**2**:  $uv$  besteht nur aus  $a$ -Symbolen.

**1**:  $|v| = \ell \geq 1$ .

- ▶ Da **3**:  $uw = a^{n-\ell} b^n \in L \rightarrow$  Widerspruch. □

## Anwendung des Pumping Lemmas

$\forall$  reg. Spr.  $L$ :  $\exists n$ :  $\forall z \in L$  mit  $|z| \geq n$ :  $\exists uvw = z$  mit:

**1**  $|v| \geq 1$ ,

**2**  $|uv| \leq n$ ,

**3**  $uv^*w \in L$ .

**Aufgabe:** Sei  $L = \{a^{2^i} \mid i \geq 0\}$ , d.h. Wörter die aus „2er-Potenz“ vielen „a“s bestehen. Zeige, dass  $L$  **nicht** regulär ist.

**Lösung: Beweis durch Widerspruch.**

Nimm an,  $L$  wäre regulär, dann würde für  $L$  das Pumping Lemma gelten. Sei  $n$  die entsprechende Wortmindestgröße.

Wähle ein  $k$ , so dass  $2^k > n$ . Betrachte das Wort  $z = a^{2^k} \in L$ .

Wähle **ein** Wort  $z$  als Gegenbeispiel!

▶  $\exists$  Zerlegung  $z = uvw$  mit **1** – **3**. Bedenke **alle** mögl. Zerl.!

**1**:  $|v| \geq 1$       **2**:  $|uv| \leq n \rightarrow |w| \geq 1$ .

▶ Da **3**:  $y = uv^2w \in L$ :  $|uv^2w| = |uvw| + |v| < 2|uvw|$   
 $\rightarrow y$  ist länger als  $z$  (wg. **1**), aber kürzer als nächste 2er-Potenz (wg. **2**).  $\rightarrow$  Widerspruch. □

## Grenzen des Pumping Lemmas

### Beobachtung.

Es gibt nicht-reguläre Sprachen, für die das Pumping Lemma **nicht stark genug** ist, um die Nicht-Regularität zu beweisen.

**Beispiel.** Sei  $L = \{c^j a^i b^i \mid i, j \geq 0\} \cup \{a^j b^i \mid i, j \geq 0\}$ .

→  $L$  kann nicht regulär sein, da (in erster Teilmenge) nur ein paar  $c$  vor einer nicht-regulären Sprache stehen (siehe vorhin)

### Pumping Lemma.

Betrachte bel. Wort  $z = c^j a^i b^i$  oder  $z = a^j b^i$  mit  $|z| \geq n$  aus  $L$ .

Zerlegung:  $u = \varepsilon$ ,  $v = z[1]$ ,  $w = [2 \dots]$  ( $v$  ist das erste Symbol von  $z$ ,  $w$  ist der Rest). → Jedes  $u v^* w$  liegt in  $L$ !

→ Kein Widerspruch → **Beweis funktioniert nicht.**

⇒ Es gibt jedoch andere Methoden (statt des Pumping Lemmas), um den Beweis der Nicht-Regularität zu führen (siehe einschlägige Literatur und nächste Folie).

# Abgeschlossenheit

## Theorem.

Reguläre Sprachen sind abgeschlossen bezüglich Verkettung, Vereinigung, Komplementbildung, und Schnitt.

*D.h.: Gegeben reguläre Sprachen  $L_1, L_2$ . Die folgenden Sprachen sind auch regulär:  $L_1 L_2$ ,  $L_1 \cup L_2$ ,  $\overline{L_1} := \Sigma^* \setminus L_1$ ,  $L_1 \cap L_2$ .*

## Beweis.

**Verkettung und Vereinigung.** → Definition reg. Ausdrücke!

## Komplementbildung.

Sei  $\mathcal{A}_1$  der deterministische EA zu  $L_1$ . Alle Nicht-Endzustände von  $\mathcal{A}_1$  werden Endzustände, und umgekehrt. Zusätzlicher Endzustand, zu dem alle „fehlenden“ Übergänge führen.

**Schnitt.**  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

# **Formale Sprachen & Automaten**

Kontextfreie Sprachen &  
Kellerautomaten

# Jenseits der Regularität – Grammatik

Einschränkungen für die Regeln  $x \rightarrow y$ :

**Typ 0: Rekursiv aufzählbare Sprachen**

keine Einschränkungen

**Typ 1: Kontextsensitive (bzw. monotone) Sprachen**

$|x| \leq |y|$

**Typ 2: Kontextfreie Sprachen**

Typ 1, und  $x \in V$  (links eine **einzelne** Variable)

**Typ 3: Reguläre Sprachen**

$x \in V, y \in \Sigma \cup \Sigma V$  (rechts **ein** Symbol, ggf. gefolgt von **einer** Variable)

**Typ 0**

**Rekursiv aufzählb.**

**Typ 1**

**Kontextsensitiv**

**Typ 2**

**Kontextfrei**

(deterministisch  
kontextfrei)

**Typ 3**

**Regulär**

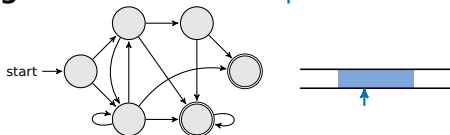
Sprache wird ausdrucksstärker

Wortproblem wird leichter

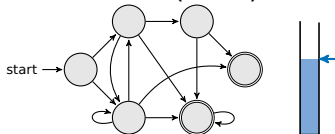
# Jenseits der Regularität – Automaten



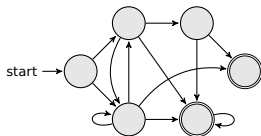
← **Turingmaschine** = EA + Speicherband



← **Kellerautomat** = (det.?) EA + Stack

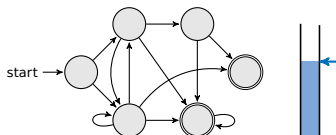


← **EA**



# Kellerautomaten

Kellerautomat = Endlicher Automat + Stack (Stapel)



- ▶ Der nächste Schritt wird durch das aktuell gelesene Symbol **und** dem obersten Symbol auf dem Stack bestimmt.
- ▶ Bei jedem Schritt kann man ein (oder mehrere) Symbole auf den Stack legen oder davon entfernen.

**Nicht-Deterministischer Kellerautomat** (also mehrere Möglichkeiten bei Schritten)  $\longleftrightarrow$  **kontextfreie** Sprachen.

**Deterministischer Kellerautomat** (also immer eindeutige Schritte)  $\longleftrightarrow$  **deterministisch kontextfreie** Sprachen.

# Geklammerte Ausdrücke

Kontextfreie Sprachen  $\approx$  „korrekt geklammerte Ausdrücke“

- ▶ Klammerfolgen  $((()((())))$   
aber nicht  $()()((()$

- ▶ HTML, XML, etc.:

```
<html>
  <head>...</head>
  <body>
    <h1>Text</h1>
    <ul>
      <li> auch ohne
      <li> end-tags
    </ul>
  </body>
</html>
```

- ▶ Palindrome: otto, kayak
- ▶ Arithmetische Ausdrücke  
u.ä.:  $3 + (2 - 1) * 7$
- ▶ Programmiersprachen (?):  
IF clown != white THEN  
 REPEAT  
 laugh()  
 UNTIL act.done()  
ELSE  
 leaveCircus()  
ENDIF
- ▶ u.v.m.

# Kontextfrei vs. Programmiersprachen?

## Wofür benötigt man kontextfreie Sprachen?

### Struktur von **Programmiersprachen**

– die Antwort ist nicht richtig und nicht falsch

### Tatsachen:

- ▶ „Reine“ Grammatik von Programmiersprachen ist **deterministisch** kontextfrei!  
Für Menschen leichter (eindeutig) lesbar.  
Für Computer schneller zu parsen (linear statt kubisch).
- ▶ Programmiersprachen sind **eigentlich kontextsensitiv!**  
(z.B. Variablen vor Benutzung deklarieren, etc.)  
Kontextsensitive Sprachen viel schwerer zu parsen  
(exponentielle Laufzeit!) → Parsen der „reinen“ determ.  
KF Grammatik und Prüfen der weiteren Bedingungen  
danach mit Hilfe des Syntaxbaums.  
Ausnahmen bestätigen Regel: Prolog ist tatsächlich DKF.

# **Formale Sprachen & Automaten**

Rekursiv Aufzählbare Sprachen &  
Turingmaschinen

# Alan Turing



## Alan Mathison Turing

\* 23. Juni 1912 in London

† 7. Juni 1954 in Wilmslow (England)

Der wohl wichtigste Informatiker aller Zeiten.

- ▶ **1936:** Veröffentlichung: **On Computable Numbers, with an Application to the “Entscheidungsproblem”**:  
Erfindung der Turing-Maschine, Beweise der Mächtigkeit der Maschine
- ▶ **1938:** Promotion in Princeton unter **Alonzo Church**.
- ▶ Grundlagen der theoretischen Informatik, Berechenbarkeit (“Algorithmische Variante von Gödels Unvollständigkeitssatz”)

# Alan Turing



## Alan Mathison Turing

\* 23. Juni 1912 in London

† 7. Juni 1954 in Wilmslow (England)

Der wohl wichtigste Informatiker aller Zeiten.

- ▶ **2. Weltkrieg:** Knacken deutscher Geheimcodes (z.B. der **Enigma**) im Bletchley Park, **Turing-Bombe** (Maschine zum Code-Knacken),... — Anteil bis in die 70er Jahre geheim!
- ▶ **1948:** **LU Faktorisierung** von Matrizen
- ▶ **1950:** **Turing-Test**, Erkennen von Intelligenz
- ▶ **1952–54:** Mathematik in Biologie (**Turing-Mechanismus**)

# Alan Turing



## Alan Mathison Turing

\* 23. Juni 1912 in London

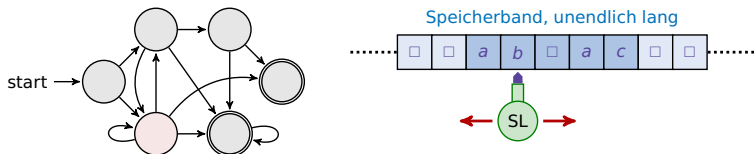
† 7. Juni 1954 in Wilmslow (England)

[Selbstmord mit vergiftetem Apfel?]

Der wohl wichtigste Informatiker aller Zeiten.

- ▶ **1952:** Verurteilung wg. Homosexualität, Ausschluss aus allen Geheimprojekten der Regierung, zwangsweise Hormonbehandlung → Depressionen
- ▶ **1954:** Tod durch Cyanidvergiftung
- ▶ Ihm zu Ehren seit **1966:** [Turing Award](#)  
≈ „Nobelpreis“ der Informatik.
- ▶ **2009:** Britische Regierung entschuldigt sich offiziell,  
**2013:** Königliche Begnadigung durch Queen Elisabeth II

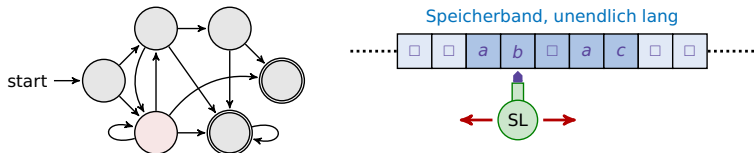
## Turingmaschine (TM), 1/3



Eine **Turingmaschine** ist ein endlicher Automat, der mit einem **Speicherband** und einem **Schreib-Lese-Kopf (SL-Kopf)** am Band gekoppelt ist.

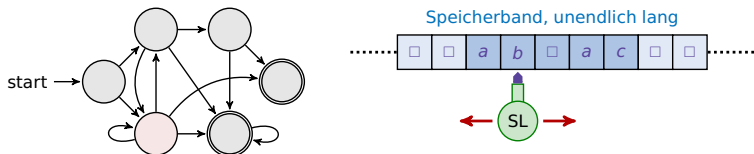
- ▶ Das Band ist unendlich lang und in Zellen unterteilt.
- ▶ In jeder Zelle ist immer ein Symbol aus dem **Bandalphabet**  $\Gamma (\supseteq \Sigma)$  gespeichert. Das Symbol  $\square \in \Gamma$  kennzeichnet eine „leere“ Zelle.
- ▶ Der SL-Kopf zeigt stets auf eine Zelle des Bands. Nur dieser **aktuelle Bandeintrag** kann gelesen bzw. geändert werden.

## Turingmaschine (TM), 2/3



- ▶ **Konfiguration** = „Zustand der Maschine“  
= Aktueller Zustand des EA + Inhalt des Bands + Position des SL-Kopfs
- ▶ Zustandsübergänge im EA sind abhängig vom aktuellen Bandeintrag
- ▶ Beim Zustandswechsel im EA wird...
  - ▶ ...der aktuelle Bandeintrag durch einen neuen Wert überschrieben (ggf. durch den alten selbst).
  - ▶ ...der SL-Kopf wird ggf. um **eine** Zelle nach links oder rechts verschoben.

## Turingmaschine (TM), 3/3



### Initialisierung:

- ▶ Das Eingabewort wird vom EA **nicht** explizit gelesen.
- ▶ Das Eingabewort steht (von links nach rechts) am Band.
- ▶ Der SL-Kopf steht auf dem ersten Zeichen der Eingabe.

### Terminierung:

- ▶ Die Übergangsfunktion ist **partiell!** → Die TM terminiert („hält an“) wenn kein weiterer Übergang möglich ist.
- ▶ Die TM **akzeptiert** das Eingabewort genau dann, wenn sie **in einem Endzustand terminiert**.  
→ Die TM akzeptiert **nicht**, genau dann wenn sie in einem Nicht-Endzustand **oder gar nicht** terminiert.

# Determinismus vs. Nicht-Determinismus

Wir unterscheiden wieder zwischen zwei Arten von TMn:

- ▶ **Deterministische TM (DTM):** Zu jedem Zeitpunkt ist eindeutig, welcher Übergang ausgeführt wird.
- ▶ **Nicht-deterministische TM (NDTM):** Es kann mehrdeutig sein, welchen Übergang man benutzt.

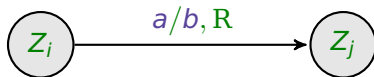
## Faszinierend!

- ▶ **Sowohl** DTM als auch NDTM beschreiben **genau** die Typ-0 Sprachen.
- ▶ Aus Komplexitätstheoretischer Sicht ist der Unterschied zwischen DTM und NDTM die **größte offene Frage der Informatik!** (siehe später in der VO)

## Graphische Notation

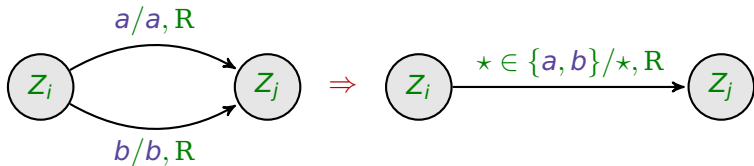
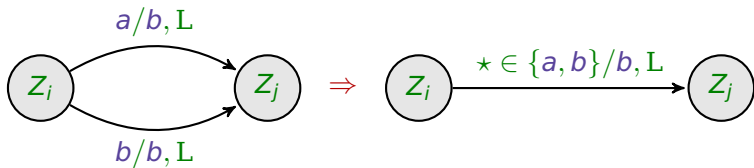
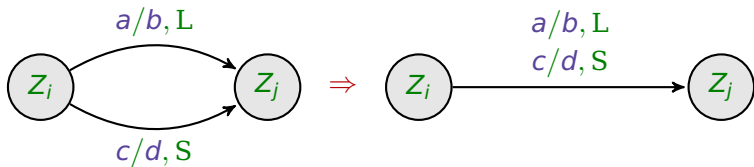
An einer Übergangskante annotieren wir **drei** Einträge:

- ▶ zu lesendes Symbol ( $\in \Gamma$ )
- ▶ zu schreibendes Symbol ( $\in \Gamma$ )
- ▶ Bewegungsrichtung:  
L=links, R=rechts, S=stehen bleiben



„Falls der aktuelle Bandeintrag  $a$  ist, wechsle von Zustand  $Z_i$  nach  $Z_j$ , überschreibe den Bandeintrag mit  $b$  und bewege den SL-Kopf eine Position nach rechts (=erhöhe den Positionsindex um 1).“

## Graphische Notation: Kurzschreibweisen

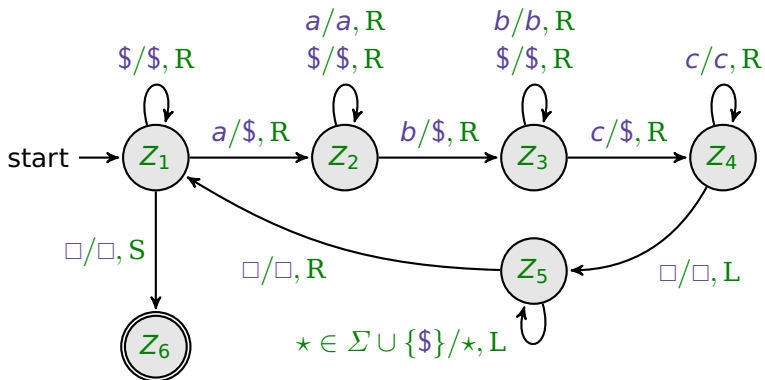


## Turingmaschine — Beispiel

$$L = \{a^i b^i c^i \mid i \geq 0\} \Rightarrow \Sigma = \{a, b, c\}, \Gamma = \{\square, a, b, c, \$\}$$

(Wir dürfen davon ausgehen, dass am Anfang nur Symbole aus  $\Sigma \cup \{\square\}$  am Band stehen.)

**DTM:**



# Turingmaschine — Beispiel 1, Auflösung

Z<sub>1</sub> □□aabbcc□□

Z<sub>2</sub> □□\$abbcc□□

Z<sub>2</sub> □□\$abbbcc□□

Z<sub>3</sub> □□\$a\$bcc□□

Z<sub>3</sub> □□\$a\$bccc□□

Z<sub>4</sub> □□\$a\$b\$c□□

Z<sub>4</sub> □□\$a\$b\$c□□□

Z<sub>5</sub> □□\$a\$b\$c□□□

Z<sub>5</sub> □□\$a\$b\$□c□□

... ..

Z<sub>5</sub> □□\$a\$b\$c□□

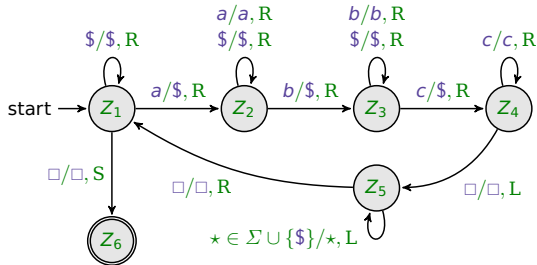
Z<sub>5</sub> □□□\$a\$b\$c□□

Z<sub>1</sub> □□\$a\$b\$c□□

Z<sub>1</sub> □□a\$b\$c□□

Z<sub>2</sub> □□\$\$\$b\$c□□

Z<sub>2</sub> □□\$\$\$b\$c□□



Z<sub>3</sub> □□\$\$\$\$\$c□□

Z<sub>3</sub> □□\$\$\$\$\$□c□□

Z<sub>4</sub> □□\$\$\$\$\$\$\$□□

Z<sub>5</sub> □□\$\$\$\$\$\$\$□□

Z<sub>5</sub> □□\$\$\$\$\$\$\$□□

... ..

Z<sub>5</sub> □□\$\$\$\$\$\$\$□□

Z<sub>5</sub> □□\$\$\$\$\$\$□□□

Z<sub>1</sub> □□\$\$\$\$\$\$\$□□

Z<sub>1</sub> □□\$\$\$\$\$\$\$□□

... ..

Z<sub>1</sub> □□\$\$\$\$\$\$\$□□

Z<sub>1</sub> □□\$\$\$\$\$\$\$□□

Z<sub>6</sub> □□\$\$\$\$\$\$□□□

Endzustand ⇒ *abbcc* ∈ L

# Turingmaschine — Beispiel 2, Auflösung

Z<sub>1</sub> □□aabcc□□

Z<sub>2</sub> □□\$abcc□□

Z<sub>2</sub> □□\$ab<sub>c</sub>cc□□

Z<sub>3</sub> □□\$a\$cc□□

Z<sub>4</sub> □□\$a\$\$\$c□□

Z<sub>4</sub> □□\$a\$\$\$c□□

Z<sub>5</sub> □□\$a\$\$\$c□□

Z<sub>5</sub> □□\$a\$\$\$c□□

... ..

Z<sub>5</sub> □□\$a\$c□□

Z<sub>5</sub> □□\$a\$c□□

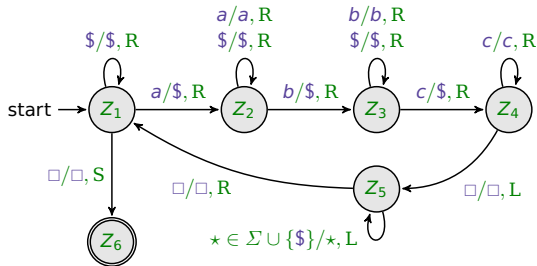
Z<sub>1</sub> □□\$a\$c□□

Z<sub>1</sub> □□\$a\$c□□

Z<sub>2</sub> □□\$\$\$c□□

Z<sub>2</sub> □□\$\$\$c□□

Z<sub>2</sub> □□\$\$\$c□□



kein weiterer Übergang möglich

→ TM terminiert in einem Nicht-Endzustand

## TM vs. Sprachen

**Definition.** Eine TM die nur die durch die Eingabe belegten Zellen benutzt (+ Grenzüberprüfung um nicht hinauszulaufen), bezeichnet man als **Linear Beschränkter Automat (Linear Bounded Automaton, LBA)**.

Determinismus? → Es gibt wieder DLBA und NDLBA.

**Beobachtung.** Unser voriger Automat war ein (D)LBA.

**Theorem.** (ohne Beweis)

Typ-0 Sprachen = Sprachen, für die es eine DTM gibt  
= Sprachen, für die es eine NDTM gibt.

Typ-1 Sprachen = Sprachen, für die es einen NDLBA gibt.

**Offenes Problem.** Typ-1 Sprachen = DLBA?

## Probleme vs. Maschinen

<b>Sprache d. Wortproblems</b>	<b>Lösbar(?) mittels</b>	<b>Komplexität (Laufzeit/Speicher)</b>
<b>Rekursiv aufzählbar</b>	Turingmaschine ( $\approx$ Computer)	nicht entscheidbar
<b>Kontextsensitiv</b>	linear beschränkter Automat	exponentiell/linear
<b>Kontextfrei</b>	Kellerautomat	kubisch/quadratisch
<b>Deterministisch Kontextfrei</b>	deterministischer Kellerautomat	linear/linear
<b>Regulär</b>	Endlicher Automat	linear/konstant

# **Berechenbarkeitstheorie**

# **Berechenbarkeitstheorie**

Rechnende Turingmaschinen

## Turingmaschinen können mehr!

Man kann mit einer (in diesem Kapitel: deterministischen) TM nicht nur Wortprobleme lösen, sondern auch direkt Berechnungen durchführen!

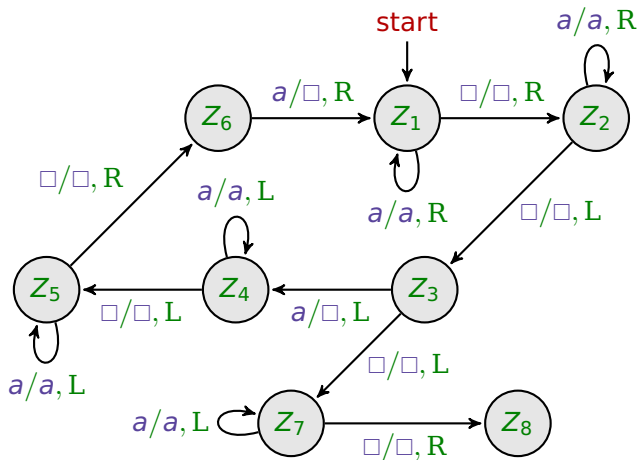
- ▶ **Vor der Berechnung:** Eingabe am Band, SL-Kopf am ersten (linkesten) Zeichen der Eingabe
- ▶ **Bei Terminierung:** Ergebnis am Band, SL-Kopf am ersten (linkesten) Zeichen des Ergebnisses

### Kodierung von Zahlen.

- ▶ Am einfachsten (aber ineffizientesten) ist es, Zahlen **unär** zu kodieren: Eine Zahl wird durch die entsprechende Anzahl von *a*s dargestellt:  $3 = aaa, 7 = aaaaaaa, \dots$
- ▶ Natürlich kann eine TM Zahlen auch binär (bzw. zu belieb. anderer Basis) kodieren. (**keine führenden Nullen!**)
- ▶ Einzelne Zahlen werden durch ein Trennsymbol (z.B.  $\square, \diamond, \#, \dots$ ) von einander getrennt.

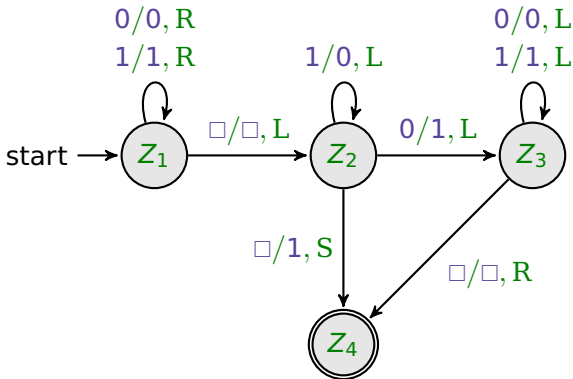
## Rechenbeispiel (Unäre Kodierung)

Gegeben zwei unär kodierte Zahlen  $\alpha$  und  $\beta$  mit  $\alpha \geq \beta$ ,  
getrennt durch  $\square$ . Berechne  $\alpha - \beta$ .



## Rechenbeispiel (Binäre Kodierung)

Gegeben eine binär kodierte Zahl  $\alpha$  ( $\Sigma = \{0, 1\}$ ).  
 Berechne  $\alpha + 1$ .



Endzustände müssen **nicht** markiert werden! Sie haben **keine** Bedeutung!  
 Wir werden die Markierung jedoch im Weiteren gerne nutzen, um diese Knoten beim „Zusammenkleben“ zu referenzieren.

## Eigenschaften der rechnenden TM

- ▶ (In diesem Kapitel reichen **DTM**n. In Komplexitätstheorie werden wir sie auch für **NDTM**n definieren.)
- ▶ Man kann TMn (z.B. um Berechnungen einfacher beschreiben zu können), auch mit mehreren SL-Köpfen, mehreren Speicherbändern, mehr-dimensionalen Speicherfeldern, zusätzlichen Stacks, etc. ausstatten.
- ▶ **Aber:** Alles was diese erweiterten Turingmaschinen berechnen können, kann man auch mit einer „normalen“ Turingmaschine berechnen!  
(Interessant: die Laufzeitkomplexität nimmt dabei i.A. gar nicht besonders stark zu, sondern z.B. nur quadratisch)
- ▶ **Insbesondere gilt sogar:**  
Eine Turingmaschine kann **alles** berechnen, was ein „normaler“ Computer berechnen kann!

→ Wir kommen später noch detaillierter darauf zurück.

# Universelle Turingmaschine

**Beobachtung.** In einer TM gilt die Aufteilung:

- ▶ Der EA kodiert das **Programm**
- ▶ Das Band kodiert den **(Arbeits)speicher**

**Nachteil.** Für jede Aufgabenstellung benötigt man eine „neue“ TM, da man einen speziellen EA benötigt.

## Universelle Turingmaschine ( $\approx$ CPU).

- ▶ Man kann das Programm (den EA) als Zeichenfolge kodieren und als Eingabe auf das Band schreiben!
- ▶ Initial stehen am Band also das **Programm**, ein **Trennsymbol** und die eigentliche **Eingabe**.
- ▶ Der (immer gleiche) EA der **universellen TM** ist programmunabhängig ( $\rightarrow$  CPU), und arbeitet die Eingabe auf Basis der im Programm kodierten Steuerbefehle ab.

# Universelle Turingmaschine – EA Kodierung

## Kodiere (D)TM für eine (D)UTM:

- ▶ Nummeriere alle  $\alpha$  Zustände  $Z_1, \dots, Z_\alpha$ , so dass  $Z_1 = Z_{\text{start}}$  und die letzten  $\beta$  Zustände Endzustände sind.
- ▶ Nummeriere alle  $\gamma$  Symbole  $\rightarrow a_1, \dots, a_\gamma$
- ▶ Mehrere Übergänge  $U_1, U_2, \dots, U_\delta$ . Ein Übergang  $Z_i \rightarrow Z_j$  mit Beschriftung „ $a_k/a_\ell, \langle d \rangle$ “ wird als 5-Tupel  $(i, j, k, \ell, d)$  kodiert. Dabei kodiert  $d \in \{0, 1, 2\}$  die Bewegungsrichtung)
- ▶ Sei  $\mathbb{B}(n)$  die Binärdarstellung einer Zahl  $n$ , also z.B.  $n = 5 \rightarrow \mathbb{B}(n) = 101$ . Sei  $\diamond$  ein Trennsymbol.
- ▶ Repräsentiere einen Übergang  $U = (i, j, k, \ell, d)$  durch das Wort  $\mathbb{W}(U) := \mathbb{B}(i) \diamond \mathbb{B}(j) \diamond \mathbb{B}(k) \diamond \mathbb{B}(\ell) \diamond \mathbb{B}(d)$ .
- ▶ Repräsentiere Turingmaschine  $\mathcal{M}$  durch das Wort  $\mathbb{W}(\mathcal{M}) := \mathbb{B}(\alpha) \diamond \mathbb{B}(\beta) \diamond \mathbb{B}(\gamma) \diamond \mathbb{B}(\delta) \diamond \mathbb{W}(U_1) \diamond \dots \diamond \mathbb{W}(U_\delta)$
- ▶ Reine Binärbeschreibung durch  $0 \rightarrow 10, 1 \rightarrow 11, \diamond \rightarrow 00$ .  
Jede TM wird durch eine **Zahl eindeutig** beschrieben

$\rightarrow$  „Gödelisierung“

# **Berechenbarkeitstheorie**

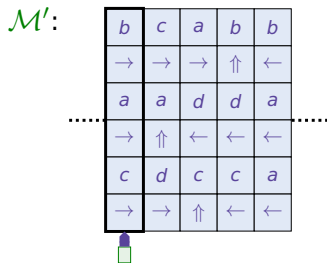
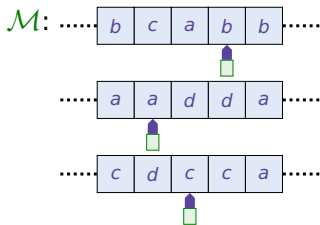
Von der TM zum Computer

## Mehrband-Turingmaschine

Eine **Mehrband-Turingmaschine**  $\mathcal{M}$  ist eine TM die  $k$  viele Bänder besitzt ( $k > 1$  und endlich). Jedes Band hat einen eigenen, unabhängigen SL-Kopf. Bei jedem Zustandsübergang im EA kann jeder SL-Kopf lesen/schreiben/sich-bewegen.

### Theorem.

Für jede Mehrband-TM  $\mathcal{M}$  existiert auch eine normale (1-Band) TM  $\mathcal{M}'$  die die selbe Funktion berechnet.



$$\Gamma_{\mathcal{M}'} = (\Gamma_{\mathcal{M}} \times \{\rightarrow, \uparrow, \leftarrow\})^k$$

## Mehrband-TM → Programmiersprache, 1/3

Wir betrachten nun immer eine Mehrband-TM  $\mathcal{M}$  mit „ausreichend vielen“ Bändern.

- ▶ Wir können jede 1-Band TM auf einem der Bänder von  $\mathcal{M}$  ablaufen lassen, ohne die anderen Bänder zu ändern.
- ▶ Wir benutzen jedes Band von  $\mathcal{M}$  als eine Variable, die genau einen Wert gespeichert hat.  
→  $V(i)$  bezeichnet den am  $i$ -ten Band gespeicherten Wert.
- ▶ Wir haben schon eine TM gesehen, die  $\mathbb{B}(\alpha) + \mathbb{B}(1)$  berechnet. Eine derartige TM können wir auf jedem beliebigen Band von  $\mathcal{M}$  laufen lassen.  
→ Wir können also die Anweisung  $V(i) := V(i) + 1$  auszuführen.
- ▶ Analog können wir TMn für  $V(i) := V(i) - 1$ ,  $V(i) := 0$ ,  $V(i) := V(j)$  erstellen.

## Mehrband-TM → Programmiersprache, 2/3

### ► Verkettung

Wir können zwei TMn  $\mathcal{M}_1, \mathcal{M}_2$  hintereinander schalten:

**Blindübergang** = Zustandwechsel für jedes bisher an dieser Stelle nicht akzeptierte Symbol. Ausgangszustand ist kein Endzustand mehr. Speicherband und SL-Kopf bleiben unverändert.

Vereinige die EAn und füge Blindübergänge von jedem Endzustand von  $\mathcal{M}_1$  zum Startzustand von  $\mathcal{M}_2$  hinzu.

→ Wir schreiben  $A_1; A_2$ , wobei  $A_1$  und  $A_2$  die durch die TMn kodierte Anweisungen sind.

### ► Addition (etc.) von Konstanten

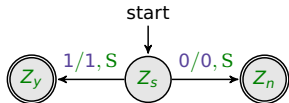
Wir können Additionen  $V(i) := V(i) + c$  erzeugen, durch

$$\underbrace{V(i) := V(i) + 1; V(i) := V(i) + 1; \dots; V(i) := V(i) + 1,}_{c \text{ mal}}$$

und analog Subtraktionen und Zuweisungen  $V(i) := c$ .

## Mehrband-TM → Programmiersprache, 3/3

- ▶ Betrachte die spezielle TM  $\mathcal{M}_{V(i) \neq 0}$  die nur am  $i$ -ten Band arbeitet. Falls  $V(i) \neq 0$  terminiert die TM in  $Z_y$ , sonst in  $Z_n$ .



Analog kann man weitere Test-TMs konstruieren.

- ▶ **If-Then-Else** → `if( $V(i) \neq 0$ ) {  $A_{\text{then}}$  } else {  $A_{\text{else}}$  }`  
 Starte mit  $\mathcal{M}_{V(i) \neq 0}$ . Von dem entspr. Endzustand dann Blindübergang zum Start der TMn  $\mathcal{M}_{\text{then}}$  bzw.  $\mathcal{M}_{\text{else}}$ .
- ▶ **While** → `while( $V(i) \neq 0$ ) {  $A_{\text{body}}$  }`  
 Starte mit  $\mathcal{M}_{V(i) \neq 0}$ . Blindübergang von  $Z_y$  zum Start einer TM  $\mathcal{M}_{\text{body}}$ , und von  $\mathcal{M}_{\text{body}}$ s Endzuständen nach  $Z_s$ .
- ▶ **Beliebige Addition, etc.** →  $V(i) := V(j) + V(k)$

```

V(i) := V(j); V(l) := V(k); // neues Band l
while(V(l) ≠ 0) { V(l) := V(l) - 1; V(i) := V(i) + 1 }
  
```

# WHILE-Programme

## WHILE-Programmiersprache:

Variablen $\in \mathbb{N}$	$x_1, x_2, \dots$
Konstanten $c$ aus $\mathbb{N}$	$0, 1, 2, 3, \dots$
Zuweisung, Addition, Subtraktion	$x_i := c, x_i := x_j \pm c$
Verkettung	;
While-Schleife	<b>while</b> ( $x_i \neq 0$ ) { ... }
Bedingtes Ausführen <sup>a</sup>	<b>if</b> (...) { ... } <b>else</b> { ... }

<sup>a</sup>Lässt sich durch Hilfsvariablen und **while** simulieren.

**Beispiel:** Berechne  $x_2 := 5 \cdot x_1$ .

$x_2 := 0; x_3 := x_1; \text{while}(x_3 \neq 0) \{ x_2 := x_2 + 5; x_3 := x_3 - 1 \}$

Wir haben gerade gesehen:

### Theorem.

Eine TM kann jedes beliebige WHILE-Programm simulieren.

# GOTO-Programme

**GOTO-Programmiersprache:** Endliche Folge

$L_1:A_1; L_2:A_2; L_3:A_3; \dots$

von Paaren aus Labeln  $L_j$  und Anweisungen  $A_j$ .

Wir verzichten auf Label, die nie angesprungen werden.

**Anweisungen:** (mit Variablen  $x_i \in \mathbb{N}$  & Konstanten  $c \in \mathbb{N}$ )

Zuweisung	$x_i := c, x_i := x_j \pm c$
Unbedingter Sprung <sup>a</sup>	<b>goto</b> $L_j$
Bedingter Sprung	<b>if</b> (Vergleich) <b>goto</b> $L_j$
Stopanweisung	<b>halt</b>

<sup>a</sup>Ließe sich auch durch bedingten Sprung mit  $x_1 = x_1$  simulieren.

**Beispiel:** Setze  $x_2 := x_1$  falls  $x_1 < 7$ , und  $x_2 := 2 \cdot x_1$  sonst.

$x_2 := x_1; \text{if}(x_1 < 7) \text{goto } L_2;$

$x_3 := x_1; L_1: x_2 := x_2 + 1; x_3 := x_3 - 1; \text{if}(x_3 \neq 0) \text{goto } L_1;$

$L_2: \text{halt}$

## WHILE vs. GOTO

**Lemma.** WHILE-Programme können durch GOTO-Programme simuliert werden.

$\text{while}(x_i \neq 0) \{ A \} \Rightarrow \text{goto } L_2; L_1: A; L_2: \text{if}(x_i \neq 0) \text{ goto } L_1;$

**Lemma.** GOTO-Programme können durch WHILE-Programme (mit einer einzigen While-Schleife!) simuliert werden.

$L_1: A_1;$	$A_i$	$A'_i$	
$L_2: A_2;$	$x_i := \dots$	$x_i := \dots;$ $x_{\text{pos}} := x_{\text{pos}} + 1$	
$L_3: A_3;$	$\text{goto } L_j$	$x_{\text{pos}} := i$	
$L_4: A_4; \Rightarrow$	$\text{if}(B) \text{ goto } L_j$	$\text{if}(B) \{ x_{\text{pos}} := i \}$ $\text{else } \{$ $x_{\text{pos}} := x_{\text{pos}} + 1 \}$	$\Rightarrow$
$L_5: A_5;$			
$\dots$			
$L_\ell: A_\ell;$	halt	$x_{\text{pos}} := 0$	

$x_{\text{pos}} := 1;$   
 $\text{while}(x_{\text{pos}} \neq 0) \{$   
 $\quad \text{if}(x_{\text{pos}} = 1) \{ A'_1 \}$   
 $\quad \text{if}(x_{\text{pos}} = 2) \{ A'_2 \}$   
 $\quad \dots$   
 $\quad \text{if}(x_{\text{pos}} = \ell) \{ A'_\ell \}$   
 $\}$

# Gleich mächtig

## Wir haben gesehen:

- ▶ Eine Turingmaschine kann WHILE-Programme simulieren.
- ▶ Ein GOTO-Programm kann WHILE-Programme simulieren.
- ▶ Ein WHILE-Programm kann GOTO-Programme simulieren.

**Daher bis jetzt:** Turingmaschinen **könnten** mächtiger sein, als WHILE/GOTO-Programme.

## Es gilt aber auch (ohne Beweis):

- ▶ GOTO-Programme können Turingmaschinen simulieren.

Daher gilt:

**Theorem.** Mit Turingmaschinen, GOTO-Programmen und WHILE-Programmen kann man **genau die selben Dinge** berechnen.

## Höhere Programmiersprachen

- ▶ Sowohl die WHILE- als auch GOTO-Programmiersprache sind (syntaktisch!) sehr reduziert.
  - ▶ **Jedes** weitere Konstrukt aus höheren algorithmischen (=imperativen bzw. funktionalen) Programmiersprachen lässt sich jedoch (aus Sicht der Berechenbarkeit) mit diesen wenigen Mitteln ausdrücken!
  - ▶ Variablennamen, Objekte, etc. sind nur „angenehmere“ Namen für Variablen(menge)  $x_1, x_2, \dots$
  - ▶ **for**-schleifen, **do-while**-Schleifen, Funktionsaufrufe, etc. lassen sich mit **goto** simulieren.
  - ▶ Negative Zahlen, Fließkomma-Zahlen, etc. lassen sich simulieren,...
- ⇒ Statt immer mit einer Turingmaschine zu arbeiten, können wir auch eine beliebige (algorithmische) Programmiersprache benutzen: C, Java, Pascal, Python, „Pseudocode“,...

# **Berechenbarkeitstheorie**

## Berechenbarkeit

# Berechenbarkeit

**Was kann ein Computer überhaupt prinzipiell berechnen, und wo liegen seine Grenzen?**

## Definition.

Eine (**totale**) Funktion  $f: \mathbb{N}^k \rightarrow \mathbb{N}$  heißt **berechenbar** falls irgendeine **endliche algorithmische Beschreibung** existiert, die für jedes  $X \in \mathbb{N}^k$  den Wert  $f(X)$  berechnet.

Eine **partielle** Funktion  $g: \text{Def} \rightarrow \mathbb{N}$ , mit  $\text{Def} \subseteq \mathbb{N}^k$ , heißt **berechenbar** falls irgendeine **endliche algorithmische Beschreibung** existiert, die für jedes  $X \in \text{Def}$  den Wert  $g(X)$  berechnet und für jedes  $X \in \mathbb{N}^k \setminus \text{Def}$  **nicht terminiert**.

(„Gleichbedeutend“:  $g: \mathbb{N}^k \rightarrow \mathbb{N} \cup \{\text{undef}\}$ )

Warum sollte eine Funktion **nicht** berechenbar sein?

# Natürliche Zahlen

Die Beschränkung auf Zahlen aus  $\mathbb{N}$  stellt **keine fachliche Einschränkung** dar!

- ▶ Ein digitaler Computer kann nur in diskreten Quantitäten rechnen  $\rightarrow \mathbb{N}, \mathbb{Z}$ .
- ▶ Rationale (fraktionale) Zahlen  $\mathbb{Q}$  können immer als Bruch zweier ganzer Zahlen dargestellt werden
- ▶ Irrationale Zahlen  $\mathbb{R}$  (z.B.  $\pi, e$ ) werden in Computern immer nur angenähert oder implizit gespeichert.

## Berechenbarkeit, Beispiele (1)

Sind die folgenden Funktionen berechenbar?

▶  $f(a, b) = \begin{cases} a^2 & \text{falls } a > 5 \\ a + b & \text{sonst} \end{cases}$

**Berechenbar!**

```
if(a>5) { return a*a; } else { return a+b; }
```

▶  $f(a) = \text{undef}$  für alle  $a$

**Berechenbar!** `while(true) {};`

▶  $f(a) = \begin{cases} 1 & \text{falls } a \text{ **Anfang** der} \\ & \text{Dezimalbruchentwicklung von } \pi \text{ ist.} \\ 0 & \text{sonst} \end{cases}$

(Beispiele:  $f(3) = 1, f(314) = 1, f(314159) = 1, f(4) = 0$ )

**Berechenbar!** *Es gibt einen Algorithmus der die ersten (beliebig vielen) Stellen von  $\pi$  berechnen kann.*

## Berechenbarkeit, Beispiele (2)

Sind die folgenden Funktionen berechenbar?

$$\blacktriangleright f(a) = \begin{cases} 1 & \text{falls } a \text{ Teil der} \\ & \text{Dezimalbruchentwicklung von } \pi \text{ ist.} \\ 0 & \text{sonst} \end{cases}$$

*Man weiß derzeit zu wenig über  $\pi$  um zu entscheiden, ob berechenbar oder nicht!*

$$\blacktriangleright f(a) = \begin{cases} 1 & \text{falls Ziffer "0" } a\text{-mal hintereinander in der} \\ & \text{Dezimalbruchentwicklung von } \pi \text{ vorkommt.} \\ 0 & \text{sonst} \end{cases}$$

**Berechenbar!** *Man weiß **derzeit** nicht, ob beliebig lange 0-er Folgen vorkommen.*

*Angenommen 0-er Folgen können beliebig lang sein, dann gilt  $f(a) = 1 \forall a$ , berechenbar.*

*Angenommen die längste 0-er Folge hat Länge  $n_0$ , dann gilt  $f(a) = 1 (\forall a \leq n_0)$ ,  $f(a) = 0 (\forall a > n_0)$ , berechenbar.*

## Church'sche These

**Informelle** Definition („endliche algorithm. Beschreibung“) von Berechenbarkeit ist formal nicht zu fassen!

**Berechenbarkeiten.** Wir kennen implizit schon verschiedene **stets äquivalente** formale Berechenbarkeiten:

<b>Berechenbarkeit</b>	<b>= berechenbar durch</b>
Turing-berechenbar	Turingmaschine
WHILE-berechenbar	WHILE-Programm
GOTO-berechenbar	GOTO-Programm
$\mu$ -rekursiv-berechenbar	$\mu$ -rekursive Funktionen

### Church'sche These

(Alonzo Church, Doktorvater von Alan Turing)

Die Turing-berechenbaren, WHILE-berechenbaren, GOTO-berechenbaren und  $\mu$ -rekursiv-berechenbar Funktionen sind genau die intuitiv berechenbaren Funktionen.

# Turing-berechenbar

## Definition.

Eine eventuell partielle Funktion  $f: \text{Def} \rightarrow \mathbb{N}$ , mit  $\text{Def} \subseteq \mathbb{N}^k$ , heißt **Turing-berechenbar**, wenn eine (D)TM  $\mathcal{M}$  existiert, so dass für jedes  $(x_1, x_2, \dots, x_k) \in \mathbb{N}^k$ :

- ▶  $\mathcal{M}$  mit Eingabe  $\mathbb{B}(x_1) \diamond \mathbb{B}(x_2) \diamond \dots \diamond \mathbb{B}(x_k)$  terminiert mit Ausgabe  $\mathbb{B}(y)$ ,  
**genau dann wenn**  
 $(x_1, x_2, \dots, x_k) \in \text{Def}$  und  $f(x_1, x_2, \dots, x_k) = y$ .
- ▶  $\mathcal{M}$  mit Eingabe  $\mathbb{B}(x_1) \diamond \mathbb{B}(x_2) \diamond \dots \diamond \mathbb{B}(x_k)$  terminiert **nicht**,  
**genau dann wenn**  
 $(x_1, x_2, \dots, x_k) \notin \text{Def}$  (also  $f(x_1, x_2, \dots, x_k) = \text{undef}$ ).

# Turing-vollständig

**Definition.** Eine Maschine oder Programmiersprache ist **Turing-vollständig**, wenn sie eine TM simulieren kann.

- ▶ Eine Turing-vollständige Maschine/Programmiersprache ist also (mindestens) so mächtig wie eine TM.
- ▶ Nach Church'scher These ist Turing-Berechenbarkeit das selbe wie allgemeine/intuitive Berechenbarkeit.
- ▶ Alle bekannten realen Turing-vollständigen Systeme sind **Turing-äquivalent**, d.h. genauso mächtig wie TMn.

**Reale Computer** sind formal **nicht Turing-vollständig**, da sie nur beschränkten Speicher haben!

→ **Dennoch:** Wir bezeichnen eine **reale** Maschine bzw. Programmiersprache als Turing-vollständig, wenn sie TMn simulieren **könnte** wenn sie unbeschränkten Speicher **hätte**.

## LOOP und Ackermann, 1/5

**1920er Jahre:** Heutige Definition von Berechenbarkeit noch unbekannt. Man betrachtete Berechenbarkeit von **totalen** Funktionen (also kein **undef** = Endlosschleife).

**LOOP-Programme:** Wie WHILE-Programme, aber statt while-Schleifen nur for-Schleifen: „**loop**( $x_i$ ) {  $A$  }“  $\Rightarrow A$  wird  $x_i$ -mal ausgeführt;  $x_i$  wird dazu beim ersten Ablaufen ausgelesen; Änderungen von  $x_i$  innerhalb von  $A$  ändern nichts an der Iterationszahl.

**(Falsche) Vermutung:** (u.A. von David Hilbert) Alle berechenbaren Funktionen sind „primitiv-rekursiv“, d.h. lassen sich als LOOP-Programm schreiben.

**Widerlegung:** (Wilhelm Ackermann, 1928) **(1)** Zeige, dass Zahlenwerte in LOOP-Programmen nicht beliebig schnell wachsen können. **(2)** Definiere eine (totale) berechenbare Funktion (**Ackermann-Funktion**) die schneller wächst.

## LOOP und Ackermann, 2/5

Vereinfachte (einstellige) Ackermannfunktion:

$$\begin{aligned}
 A(0) &:= 0 && = 0 \\
 A(1) &:= 1 \circ_1 1 = 1 + 1 && = 2 \\
 A(2) &:= 2 \circ_2 2 = 2 \times 2 &= 2 + 2 && = 4 \\
 A(3) &:= 3 \circ_3 3 = 3 \uparrow 3 = 3^3 &= 3 \times 3 \times 3 && = 27 \\
 A(4) &:= 4 \circ_4 4 = 4 \uparrow\uparrow 4 &= 4^{4^4} && \geq 10^{10^{153}} \\
 A(5) &:= 5 \circ_5 5 = 5 \uparrow\uparrow\uparrow 5 &= 5 \uparrow\uparrow 5 \uparrow\uparrow 5 \uparrow\uparrow 5 \uparrow\uparrow 5 && \geq \text{urgh...} \\
 &\vdots \\
 A(n) &:= n \circ_n n = n \uparrow^{n-2} n &= \underbrace{n \uparrow^{n-3} n \uparrow^{n-3} \dots \uparrow^{n-3} n}_{n \text{ mal}} = \\
 &&&= \underbrace{n \circ_{n-1} n \circ_{n-1} \dots \circ_{n-1} n}_{n \text{ mal}}
 \end{aligned}$$

## LOOP und Ackermann, 3/5

Warum kann man  $A(n)$  nicht mit **LOOP** beschreiben?

**Beweisidee.**

- ▶ LOOP unterstützt nur Addition ( $\circ_1$ ).
- ▶ Multiplikation ( $\circ_2$ ) = Iteriertes Addieren  
⇒ Schleife über  $\circ_1$
- ▶ Potenzieren ( $\circ_3$ ) = Iterierte Multiplikation  
⇒ Schleife über Schleife über  $\circ_1$
- ▶  $n$ -te Operation ( $\circ_n$ ) = Iteriertes  $\circ_{n-1}$   
⇒  $n - 1$  verschachtelte Schleifen über  $\circ_1$

Angenommen  $\exists$  LOOP-Programm  $\mathcal{P}$  für die Ackermann:

Sei  $\ell$  die Anzahl der Befehle in  $\mathcal{P}$  (**endlich**)

→  $\mathcal{P}$  enthält maximal  $\ell - 1$  verschachtelte Schleifen.

→  $\mathcal{P}$  **kann nicht**  $A(n)$  für  $n > \ell$  berechnen können!

## LOOP und Ackermann, 4/5

Ackermannfunktion als Pseudocode:

```

int ackermann(int n) {
    if (n ≤ 2) return 2n;           // 0, 1 + 1, 2 × 2
    return ack(n,n,n);           // n ≥ 3: n ◦n n
}

int ack(int a, int b, int i) {
    if (i == 2) return a × b;     // a ◦i b ⇒
    if (b == 2) return ack(a, a, i−1); // a ◦i 2 ⇒ a ◦i−1 a
    return ack(a, ack(a,b−1,i), i−1); // ⇒ a ◦i−1 (a ◦i (b − 1))
}

```

∃ algorithmische Beschreibung ⇒ **berechenbar** !

Der Code lässt sich auch mittels WHILE bzw. GOTO schreiben.  
(Aber eben nicht mit LOOP.)

## LOOP und Ackermann, 5/5

Wozu sind Ackermann- und ähnlich schnellwachsende Funktionen gut?

- ▶ Ursprüngl.: LOOP- schwächer als WHILE-Berechenbarkeit.
- ▶ Laufzeit der **Union-Find Datenstruktur** beim Minimalen Spannbaum nach Kruskal:  $O(|E| \cdot \alpha(|V|))$ , wobei  $\alpha$  die *inverse* Ackermannfunktion ist.
- ▶ Ramsey-Theory: Was ist die kleinste Dimension  $n$ , so dass ein  $n$ -dimensionalen Würfel eine bestimmte Eigenschaft (Details nun irrelevant) hat.

**Graham's number** (Guinness Book of World Records):

$n \leq F^{64}(4) = F(F(F(\dots F(F(4)) \dots)))$ , mit  $F(x) := 3 \uparrow^x 3$

$\Rightarrow$  1971:  $6 \leq n \leq f^7(12)$ , mit  $f(x) := 2 \uparrow^x 3$

$\Rightarrow$  derzeit:  $13 \leq n \leq 2 \uparrow\uparrow\uparrow 6$  (Nov. 2014)

# **Berechenbarkeitstheorie**

Entscheidbarkeit

# Entscheidbarkeit

**Berechenbarkeit** war für Funktionen definiert.

Uns interessieren i.A. allerdings „Probleme“ (z.B. Gilt eine Formel? Wie verläuft ein kürzester Weg?..).

**Definition.** Sei  $L \subseteq \Sigma^*$  eine Sprache und  $\chi_L, \chi'_L: \Sigma^* \rightarrow \{0, 1\}$  die **charakteristische Funktion**, bzw. ihre „positive Hälfte“:

$$\chi_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{falls } w \notin L. \end{cases} \quad \chi'_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0 \text{ oder undef,} & \text{falls } w \notin L. \end{cases}$$

$L$  ist **entscheidbar**, falls  $\chi_L$  berechenbar ist.

$L$  ist **semi-entscheidbar**, falls  $\chi'_L$  berechenbar ist.

## Entscheidungsproblem.

(engl. *decision problem*, *Entscheidungsproblem*)

Gegeben  $L \subseteq \Sigma^*$ . Ist  $L$  entscheidbar?

## Intermezzo: Sprachen vs. Probleme, 1/3

**Berechenbarkeit** war für Funktionen definiert.

Uns interessieren i.A. allerdings „Probleme“ (z.B. Gilt eine Formel? Wie verläuft ein kürzester Weg? . . .).

„**Probleme**“? Jetzt haben wir schon wieder nur Sprachen benutzt...

Sprachen sind (Entscheidungs)probleme und umgekehrt!

### Beispiel: Formelerfüllbarkeit (engl.: Satisfiability)

**Gegeben:** aussagenlogische Formel

$$f := (x_1 \vee x_2) \wedge x_3 \dots \wedge (x_4 \oplus x_3).$$

**Frage:** Gibt es eine Variablenbelegung, sodass  $f = \text{true}$ ?



Sei  $\Sigma := \{\wedge, \vee, \oplus, \rightarrow, \leftrightarrow, (, ), x, 0, 1, \dots, 9\}$ .

Sei  $L \subseteq \Sigma^*$  die Sprache aller erfüllbaren Formeln.

Gilt  $f \in L$ ?

## Intermezzo: Sprachen vs. Probleme, 2/3

### Beispiel: Zusammenhang von Graphen

**Gegeben:** Graph  $G$ . **Frage:** Ist  $G$  zusammenhängend?



Sei  $\Sigma$  ein Alphabet mit dem wir Graphen kodieren können.

Sei  $L' \subseteq \Sigma^*$  eine Sprache (Methode) Graphen zu kodieren.

Sei  $L \subseteq L' \subseteq \Sigma^*$  eine Sprache, in der die kodierten Graphen zusammenhängend sind.

Sei  $\mathbb{W}(G) \in L'$  der gemäß  $L'$  kodierte Graph.

Gilt  $\mathbb{W}(G) \in L$ ?

### Beobachtungen

- ▶ Die erste Schreibweise („Gegeben-Frage“) ist einfacher.
- ▶ Die genaue Kodierung des Graphs (z.B. als Adjazenzmatrix, etc.) ist **irrelevant!**

## Intermezzo: Sprachen vs. Probleme, 3/3

- ▶ „**Das** Entscheidungsproblem“ ist, ob das Wortproblem für eine gegebene Sprache entscheidbar ist.
- ▶ Das Wortproblem für eine gegebene Sprache ist **ein** Entscheidungsproblem.
- ▶ Entscheidungsprobleme sind Fragestellungen die nur mit **JA** oder **NEIN** beantwortet werden können.
- ▶ Jedes beliebige Entscheidungsproblem lässt sich in ein Wortproblem über eine geeignete Sprache umwandeln.
- ▶ **Sprachen = Probleme**
- ▶ Sei  $L$  eine Sprache/ein Problem.  
Das Komplement  $\bar{L}$  von  $L$  ist das **Co-Problem**.  
Eine JA-Instanz für  $L$  ist eine NEIN-Instanz für  $\bar{L}$  und umgekehrt, denn:  $w \in L \iff w \notin \bar{L}$

## Entscheidbarkeit (nochmals)

**Berechenbarkeit** war für Funktionen definiert.

Uns interessieren i.A. allerdings „Probleme“ (z.B. Gilt eine Formel? Wie verläuft ein kürzester Weg?..).

**Definition.** Sei  $L \subseteq \Sigma^*$  eine Sprache und  $\chi_L, \chi'_L: \Sigma^* \rightarrow \{0, 1\}$  die **charakteristische Funktion**, bzw. ihre „positive Hälfte“:

$$\chi_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{falls } w \notin L. \end{cases} \quad \chi'_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0 \text{ oder undef,} & \text{falls } w \notin L. \end{cases}$$

$L$  ist **entscheidbar**, falls  $\chi_L$  berechenbar ist.

$L$  ist **semi-entscheidbar**, falls  $\chi'_L$  berechenbar ist.

### Entscheidungsproblem.

(engl. *decision problem*, *Entscheidungsproblem*)

Gegeben  $L \subseteq \Sigma^*$ . Ist  $L$  entscheidbar?

## Entscheidbarkeit (nochmals)

**Berechenbarkeit:** Eine Sprache, die nicht entscheidbar ist, ist **unentscheidbar**. Sie kann dennoch gleichzeitig **auch** **semi-entscheidbar** sein!

**Definition.** Sei  $L \subseteq \Sigma^*$  eine Sprache und  $\chi_L, \chi'_L: \Sigma^* \rightarrow \{0, 1\}$  die **charakteristische Funktion**, bzw. ihre „positive Hälfte“:

$$\chi_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{falls } w \notin L. \end{cases} \quad \chi'_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0 \text{ oder undef,} & \text{falls } w \notin L. \end{cases}$$

$L$  ist **entscheidbar**, falls  $\chi_L$  berechenbar ist.

$L$  ist **semi-entscheidbar**, falls  $\chi'_L$  berechenbar ist.

### Entscheidungsproblem.

(engl. *decision problem*, *Entscheidungsproblem*)

Gegeben  $L \subseteq \Sigma^*$ . Ist  $L$  entscheidbar?

## Entscheidbarkeit vs. Semi-Entscheidbarkeit

**Theorem.** Eine Sprache  $L$  ist entscheidbar  $\iff$   
 $L$  und  $\bar{L}$  (das Komplement von  $L$ ) sind semi-entscheidbar.

### Beweis.

$L$  entscheidbar  $\Rightarrow$   $L$  und  $\bar{L}$  semi-entscheidbar: trivial,  $\checkmark$ .

$L$  und  $\bar{L}$  semi-entscheidbar  $\Rightarrow$   $L$  entscheidbar:

- ▶ Seien  $\mathcal{M}$ ,  $\bar{\mathcal{M}}$  Semi-Entscheidungsalgorithmen für  $L$ ,  $\bar{L}$  (gegeben als Turingmaschinen).
- ▶  $\mathcal{M}_i(x)$  bzw.  $\bar{\mathcal{M}}_i(x)$  führt die ersten  $i$  Schritte des jeweiligen Algorithmus' für Eingabe  $x$  aus.
- ▶ Der folgende Algorithmus entscheidet  $L$  für Eingabe  $x$ :

```
for( $i := 1, 2, 3, \dots$ ) {  
    if( $\mathcal{M}_i(x)$  stoppt mit 1) return 1;  
    if( $\bar{\mathcal{M}}_i(x)$  stoppt mit 1) return 0;  
}
```

# Co-semi-entscheidbar und unentscheidbar

	$w \in L$	$w \notin L$
Charakt. Funktion $\chi_L(w)$	1	0
„Positive Hälfte“ $\chi'_L(w)$	1	0 oder undef
„Negative Hälfte“ $\chi''_L(w)$	1 oder undef	0

$L$  ist **entscheidbar**  $\iff \chi_L(w)$  berechenbar

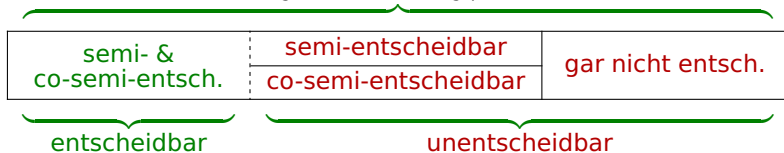
$L$  ist **semi-entscheidbar**  $\iff \chi'_L(w)$  berechenbar

$L$  ist **co-semi-entscheidbar**  $\iff \chi''_L(w)$  berechenbar

$L$  ist **unentscheidbar**  $\iff \chi_L(w)$  **nicht** berechenbar

$L$  ist **gar nicht entscheidbar**  $\iff$  weder  $\chi'_L(w)$  noch  $\chi''_L(w)$  berechenbar

Menge aller Entscheidungsprobleme



## Rekursiv aufzählbar, 1/5

**Definition.** Eine Sprache  $L$  ist **rekursiv aufzählbar**, falls ( $L = \emptyset$  oder) eine **berechenbare aufzählende** Funktion  $f: \mathbb{N} \rightarrow \Sigma^*$  existiert, so dass  $L = \{f(0), f(1), f(2), \dots\}$  und  $\min f^{-1}(w)$  für jedes  $w \in L$  endlich ist. ( $f(i) = f(j)$  für  $i \neq j$  ist OK!)

**Achtung:** **rekursiv** hat hier **nichts** mit Rekursionen von Programmiersprachen zu tun. . .

### Anmerkung.

Alle Sprache die durch Grammatikregeln beschrieben werden sind gemäß obiger Definition rekursiv aufzählbar!

⇒ Daher der Name in der Chomsky-Hierarchie!

### Beweisidee.

Die aufzählende Funktion versucht „der Reihe nach“ (schlaue Reihenfolge nötig!) alle Möglichkeiten durch, alle Regeln an allen Positionen der aktuellen Satzform anzuwenden. □

## Rekursiv aufzählbar, 2/5

**Definition.** Eine Sprache  $L$  ist **rekursiv aufzählbar**, falls ( $L = \emptyset$  oder) eine **berechenbare aufzählende** Funktion  $f: \mathbb{N} \rightarrow \Sigma^*$  existiert, so dass  $L = \{f(0), f(1), f(2), \dots\}$  und  $\min f^{-1}(w)$  für jedes  $w \in L$  endlich ist. ( $f(i)=f(j)$  für  $i \neq j$  ist OK!)

### Theorem.

$L$  ist rekursiv aufzählbar  $\iff L$  ist semi-entscheidbar.

**Beweis.**  $\Rightarrow$ : Semi-Entscheidungsalgorithmus für „ $w \in L$ ?“:

```
for  $i := 0, 1, 2, \dots$ :  
    if  $f(i) = w$ :  
        return 1
```

$\Leftarrow$ : nächste Folien...

## Rekursiv aufzählbar, 3/5

**Beweis (Fortsetzung).**  $\Leftarrow$ : Aufzählende Funktion  $f$ ?

**Idee:** **1** Gehe alle Worte  $w$  der Reihe nach durch,  
und **2** prüfe ob  $w \in L$ .

So finden wir der Reihe nach  $f(0), f(1), f(2), \dots$

**Problem 1:** Jedes mögliche Wort muss mal dran kommen!

$\Sigma = \{a, b\}$ . Alle Worte  $\Sigma^*$ ?

**Schlechte Reihenfolge:**  $\varepsilon, \underbrace{a, aa, aaa, \dots}_{\infty \text{ viele}}, \underbrace{b, bb, \dots, \dots}_{\text{wird nie erreicht}}$

**Gute Reihenfolge:**  $\varepsilon, \underbrace{a, b}_{\text{Länge 1}}, \underbrace{aa, ab, ba, bb}_{\text{Länge 2}}, aaa, \dots$

**Problem 2:** Das Prüfen dauert ggf.  $\infty$ -lang  $\rightarrow$  nächstes Wort wird nie angeschaut...

## Rekursiv aufzählbar, 4/5

### Diagonalisierung

(Sie erinnern sich an Cantor?)

$\mathcal{A}(w, i)$  = Gibt  $w$  aus, falls der Semi-Entsch.alg. für das Wort  $w$  nach maximal  $i$  Rechenschritten eine  $\mathbf{1}$  zurückliefert.

	1	2	3	4	5	...
$\varepsilon$	$\mathcal{A}(\varepsilon, 1)$	$\mathcal{A}(\varepsilon, 2)$	$\mathcal{A}(\varepsilon, 3)$	$\mathcal{A}(\varepsilon, 4)$	$\mathcal{A}(\varepsilon, 5)$	...
$a$	$\mathcal{A}(a, 1)$	$\mathcal{A}(a, 2)$	$\mathcal{A}(a, 3)$	$\mathcal{A}(a, 4)$	$\mathcal{A}(a, 5)$	...
$b$	$\mathcal{A}(b, 1)$	$\mathcal{A}(b, 2)$	$\mathcal{A}(b, 3)$	$\mathcal{A}(b, 4)$	$\mathcal{A}(b, 5)$	...
$aa$	$\mathcal{A}(aa, 1)$	$\mathcal{A}(aa, 2)$	$\mathcal{A}(aa, 3)$	$\mathcal{A}(aa, 4)$	$\mathcal{A}(aa, 5)$	...
$ab$	$\mathcal{A}(ab, 1)$	$\mathcal{A}(ab, 2)$	$\mathcal{A}(ab, 3)$	$\mathcal{A}(ab, 4)$	$\mathcal{A}(ab, 5)$	...
$ba$	$\mathcal{A}(ba, 1)$	$\mathcal{A}(ba, 2)$	$\mathcal{A}(ba, 3)$	$\mathcal{A}(ba, 4)$	$\mathcal{A}(ba, 5)$	...
$bb$	$\mathcal{A}(bb, 1)$	$\mathcal{A}(bb, 2)$	$\mathcal{A}(bb, 3)$	$\mathcal{A}(bb, 4)$	$\mathcal{A}(bb, 5)$	...
$aaa$	$\mathcal{A}(aaa, 1)$	$\mathcal{A}(aaa, 2)$	$\mathcal{A}(aaa, 3)$	$\mathcal{A}(aaa, 4)$	$\mathcal{A}(aaa, 5)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

In obiger Reihenfolge durchlaufen, jedes Wort aus  $L$  wird irgendwann ausgegeben. □

## Rekursiv aufzählbar, 5/5

### Einfacher:

$\forall w \in \Sigma^*$ mit	1	2	3	4	5	...
$ w  \leq 1$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$ w  = 2$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$ w  = 3$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$ w  = 4$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$ w  = 5$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$ w  = 6$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

In jeder Zeile werden nur endlich viele ( $|\Sigma|^{|w|}$ ) Worte betrachtet.

→ Jede Zelle benötigt nur endliche Zeit.

→ Jedes Wort aus  $L$  wird irgendwann ausgegeben.

# Rekursiv aufzählbar, 5/5

## Einfacher:

$\forall w \in \Sigma^*$ mit	1	2	3	4	5	...
$ w  \leq 1$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$ w  = 2$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$ w  = 3$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$ w  = 4$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$ w  = 5$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$ w  = 6$	$\mathcal{A}(w, 1)$	$\mathcal{A}(w, 2)$	$\mathcal{A}(w, 3)$	$\mathcal{A}(w, 4)$	$\mathcal{A}(w, 5)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Gegeben ein Semi-Entscheidungsalgorithmus  $\mathcal{A}$ .

### Aufzählende Funktion:

for  $i = 1, 2, 3, \dots$ :

  for all  $w \in \Sigma^*$  mit  $|w| \leq i$ :

    if  $\mathcal{A}(w)$  liefert nach  $\leq i$  Schritten eine 1:

      output  $w$

# Berechenbarkeitstheorie

## Halteproblem

# Halteproblem

## Halteproblem.

**Gegeben:** Ein Programm  $\mathcal{P}$  (WHILE, GOTO, TM, Pseudocode, etc.) und eine Eingabe  $x$ .

**Frage:** Hält (=terminiert)  $\mathcal{P}(x)$ ?

Oder gerät das Programm in eine Endlosschleife?

**Spezielles Halteproblem.** (Spezialfall des Halteproblems)

**Gegeben:** Ein Programm  $\mathcal{P}$  und eine Eingabe  $x = \mathcal{P}$ .

**Frage:** Hält  $\mathcal{P}(\mathcal{P})$ ?

## Unentscheidbarkeit des Halteproblems, 1/2

### Theorem.

Das spezielle Halteproblem (und damit auch das Halteproblem) ist **unentscheidbar**.

**Beweis.** Angenommen, das spezielle Halteproblem wäre entscheidbar durch einen Algorithmus  $\mathcal{A}$ , d.h.

$$\mathcal{A}(\mathcal{P}) = \begin{cases} 1, & \text{falls } \mathcal{P}(\mathcal{P}) \text{ h\"alt} \\ 0, & \text{falls } \mathcal{P}(\mathcal{P}) \text{ h\"alt nicht.} \end{cases}$$

Erzeuge neuen Algorithmus  $\mathcal{B}$ :

```
 $\mathcal{B}(\mathcal{P}) :=$  if( $\mathcal{A}(\mathcal{P}) = 0$ ) return 1;  
else while(true) {};
```

### Das heißt:

Falls  $\mathcal{A}(\mathcal{P})$  sagt, dass  $\mathcal{P}(\mathcal{P})$  **nicht** h\"alt, dann h\"alt  $\mathcal{B}(\mathcal{P})$ .

Falls  $\mathcal{A}(\mathcal{P})$  sagt, dass  $\mathcal{P}(\mathcal{P})$  h\"alt, dann h\"alt  $\mathcal{B}(\mathcal{P})$  **nicht**.

## Unentscheidbarkeit des Halteproblems, 2/2

$$A(\mathcal{P}) = \begin{cases} 1, & \text{falls } \mathcal{P}(\mathcal{P}) \text{ h\"alt,} \\ 0, & \text{falls } \mathcal{P}(\mathcal{P}) \text{ h\"alt nicht.} \end{cases}$$

$$B(\mathcal{P}) := \begin{cases} \text{if}(A(\mathcal{P}) = 0) \text{ return } 1; \\ \text{else while}(\text{true}) \{ \}; \end{cases} = \begin{cases} 1, & \text{falls } \mathcal{P}(\mathcal{P}) \text{ h\"alt nicht,} \\ \text{undef,} & \text{falls } \mathcal{P}(\mathcal{P}) \text{ h\"alt.} \end{cases}$$

Was passiert, wenn man  $B$  mit Eingabe  $\mathcal{P} = B$  aufruft, also  $B(B)$  berechnet?

- ▶ Angenommen  $B(B) = 1$  (also  $B$  h\"alt):  
Dann h\"alt (nach Def. von  $B$ )  $B(B)$  nicht.  $\rightarrow$  Widerspruch.
- ▶ Angenommen  $B(B) = \text{undef}$  (also  $B$  h\"alt nicht):  
Dann h\"alt (nach Def. von  $B$ )  $B(B)$ .  $\rightarrow$  Widerspruch.

**Also:** Der Entscheidungsalgorithmus  $A$  f\"ur das Halteproblem **kann nicht existieren**, da seine Existenz zu einem Widerspruch f\"uhren w\"urde. □

**Jedes** Turing-vollst\"andige Rechenmodell hat dieses Limit, **unabh\"angig** von der Church'schen These!

## Unentscheidbarkeit des Wortproblems

**(TM-)Wortproblem.** (Wiederholung) Gegeben eine akzeptierende TM  $M$  und ein Wort  $w$ . Wird  $w$  von  $M$  akzeptiert?

**Theorem.** Das (TM-)Wortproblem ist **unentscheidbar**.

**Beweis.** Angenommen, das Wortproblem wäre entscheidbar durch einen Algorithmus  $\mathcal{A}$ , d. h.  $\mathcal{A}(M, w) = 1$ , falls  $M$  das Wort  $w$  akzeptiert, und  $\mathcal{A}(M, w) = 0$  sonst.

Erzeuge neuen Algorithmus  $\mathcal{B}$ : (Eingabe: TM  $M$  „als Wort“)

$\mathcal{B}(M) :=$

```

definiere TM  $N :=$ 
    führe  $M(M)$  aus;           // ignoriere  $w$ 
    akzeptiere ( $\rightarrow$  vorher keine  $\infty$ -Schleife)
    //  $N$  wird nie wirklich ausgeführt!
return  $\mathcal{A}(N, \varepsilon)$       // wähle irgendein  $w$ , hier  $w = \varepsilon$ 
  
```

Also:  $\mathcal{A}(N, \varepsilon) = 1 \iff N$  akzeptiert  $\varepsilon \iff M(M)$  terminiert.

$\mathcal{B}(M)$  entscheidet das spezielle Halteproblem. Widerspruch!  $\square$

## Satz von Rice

**Jedes** nicht-triviale funktionale Verhalten ist unentscheidbar!  
Wie oft wird eine Schleife durchlaufen? Wird ein Codestück jemals ausgeführt? Wird eine Variable größer als 17?...

**Satz von Rice.** (1953; Henry Gordon Rice, 1920–2003)

Sei  $\mathcal{F}$  die Menge **aller** berechenbaren Funktionen, und  $\emptyset \neq \mathcal{S} \subsetneq \mathcal{F}$ . Es ist unentscheidbar, ob ein gegebenes Programm eine Funktion aus  $\mathcal{S}$  berechnet.

**Beweisidee.** Betrachte beliebige berechenbare Funktion  $f$  (o.B.d.A:  $\exists n: f(n) \neq \text{undef}$ ). Sei  $\mathcal{S} := \{f\}$ .

$\mathcal{B}(n) :=$

- 1 lasse ein Programm laufen, von dem es unentscheidbar ist, ob es anhält.
- 2 return  $f(n)$

**Frage:** Berechnet  $\mathcal{B}$  eine Funktion aus  $\mathcal{S}$ , also  $f$ ?



## Aber nicht vergessen...

### Theorem.

Das Wortproblem auf rekursiv aufzählbaren Sprachen (= durch TM beschreibbare Sprachen) und das Halteproblem sind **semi-entscheidbar**.

**Beweis (trivial!).** Lasse die Turingmaschine  $\mathcal{M}$  auf  $w$  ablaufen.  $\mathcal{M}$  terminiert (bzw. akzeptiert) gdw. die Frage des Halteproblems (bzw. Wortproblems) mit „Ja“ beantwortet werden kann. □

**Also nochmals:** Semi-Entscheidbarkeit und Unentscheidbarkeit sind **kein Widerspruch!**

# **Berechenbarkeitstheorie**

Weitere unentscheidbare Probleme und  
unberechenbare Funktionen

# Post'sches Korrespondenzproblem (PCP)

(Emil Leon Post, 1897–1954; 1947)

**Gegeben:** Endliche Menge  $\{(x_i, y_i)\}_{i=1, \dots, k}$  von 2-Tupeln aus  $\Sigma^+ \times \Sigma^+$  (mit  $|\Sigma| \geq 2$ ).

**Frage:** Gibt es eine endliche Folge von Indizes  $i_1, i_2, \dots, i_n$  mit  $n \geq 1$  und

$$x_{i_1} x_{i_2} x_{i_3} \dots x_{i_n} = y_{i_1} y_{i_2} y_{i_3} \dots y_{i_n}?$$

**Achtung:** Es darf  $n \gg k$  sein. Man kann das selbe Tupel also mehrmals benutzen!

**Beispiel.** 2-Tupel als „Domino-Stein“:

$x_i$
$y_i$

Gegeben ( $\Sigma = \{0, 1\}$ )

0	00	01	1	10
1	11	0	110	0

Mögliche Lösung

01	1	0	10	=	011010
0	110	1	0		011010

0	001	01
1	11	10

keine Lösung möglich

## PCP, Böses Beispiel

Gegebene Menge:

001	01	01	10
0	011	101	001

Kürzeste Lösung enthält **66** Dominosteine!

01	10	01	10	10	01	001	01	10	01	10
011	001	101	001	001	011	0	011	001	101	001
01	10	10	01	10	10	01	001	10	10	01
101	001	001	101	001	001	011	0	001	001	011
001	01	10	001	001	01	10	10	10	01	001
0	101	001	0	0	101	001	001	001	011	0
01	001	001	001	01	10	01	10	001	01	001
011	0	0	0	101	001	101	001	0	011	0
10	10	01	001	10	001	001	01	10	001	001
001	001	011	0	001	0	0	101	001	0	0
01	001	001	01	001	01	001	10	001	001	01
101	0	0	101	0	011	0	001	0	0	101

## Semi-entscheidbarkeit von PCP

**Theorem.** PCP ist semi-entscheidbar.

### Beweis.

Für immer größer werdende Werte von  $n$ , teste jede Auswahl und Reihenfolge von  $n$  Indizes.

```
for  $n := 1, 2, 3, \dots$ :  
  for all  $(i_1, \dots, i_n) \in \{1, \dots, k\}^n$ :  
    if  $x_{i_1} \dots x_{i_n} = y_{i_1} \dots y_{i_n}$ : return 1
```

□

Wir wollen nun aber zeigen:

**Theorem.** PCP ist unentscheidbar.

Dazu müssen wir ein bisschen ausholen...

## Modifiziertes PCP

**Definition.** Das **modifizierte PCP (=MPCP)** fragt, ob es eine PCP-Lösung mit  $i_1 = 1$  gibt, d.h. man muss mit dem ersten Tupel anfangen.

**Theorem.** MPCP ist ein Spezialfall von PCP.

**Beweis** – durch **Reduktion** von MPCP auf PCP.

Wir transformieren jede beliebige MPCP-Instanz  $J$  mittels einer **berechenbaren** Funktion  $f$  in eine äquivalente PCP-Instanz  $J' = f(J)$ , so dass  $J$  erfüllbar ist genau dann wenn  $J'$  erfüllbar ist.

Sei  $w = \sigma_1\sigma_2\dots\sigma_s \in \Sigma^+$  und  $\# \notin \Sigma$ . Wir definieren

$$\acute{w} := \sigma_1\#\sigma_2\#\dots\#\sigma_s\# \quad \text{und} \quad \grave{w} := \#\sigma_1\#\sigma_2\#\dots\#\sigma_s.$$

$$J = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_k, y_k)\}$$

$$\Rightarrow J' = \{(\#\acute{x}_1, \grave{y}_1), (\acute{x}_1, \grave{y}_1), (\acute{x}_2, \grave{y}_2), (\acute{x}_3, \grave{y}_3), \dots, (\acute{x}_k, \grave{y}_k), (\#, \#\#)\}$$

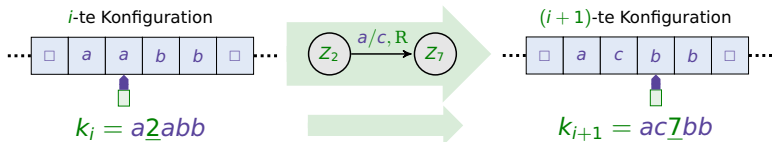
□

# Modifiziertes PCP ist unentscheidbar

**Theorem.** Das Halteproblem ist ein Spezialfall von MPCP.

**Beweisidee** – durch **Reduktion** vom Halteproblem auf MPCP.

Wir simulieren Ablauf einer Turingmaschine mittels MPCP. Kodiere „Konfiguration“ (Zustand im EA, Bandinhalt, SL-Kopf-Position):



Startkonfiguration ist  $k_0$ . Ablauf festlegbar über Einteilung:



**Problem:** Es gibt unendlich viele mögliche Konfigurationen, darf aber nur endlich viele Domino-Steine  $(x_i, y_i)$  geben.

**Lösungsidee:** Dominos, die Bandinhalt symbolweise kopieren, SL-Kopf bewegen, Terminierung beschreiben  $\rightarrow$  fummelige Details!  $\square$

# Unentscheidbarkeit von PCP

**Theorem.** PCP ist unentscheidbar.

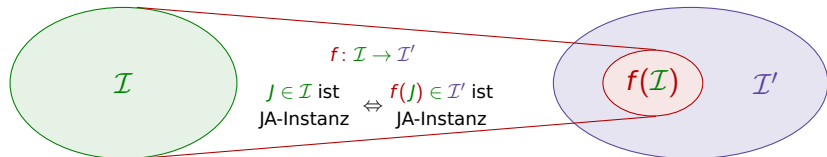
**Beweis** – durch **Reduktion** des **Halteproblems** auf **PCP**:

- ▶ Das Halteproblem ist ein Spezialfall von MPCP.
- ▶ MPCP ist ein Spezialfall vom PCP.
- ▶ PCP ist also mindestens so schwer wie MPCP, und MPCP ist mindestens so schwer wie das Halteproblem.
- ▶ Da das Halteproblem unentscheidbar ist, sind also auch MPCP und PCP unentscheidbar. □

Da das Halteproblem ein Spezialfall vom semi-entscheidbaren PCP ist, sehen wir noch einmal:

**Korollar.** Das Halteproblem ist semi-entscheidbar.

## Reduktion: ein tolles, stets hilfreiches Konzept



Was wir gesehen haben, ist eine **Reduktion** von einem Entscheidungsproblem  $\mathcal{P}$  auf ein anderes Entsch.problem  $\mathcal{P}'$ .

**Ziel:** Zeige, dass  $\mathcal{P}'$  **mindestens so schwer** ist wie  $\mathcal{P}$ .

- ▶ Finde berechenbare Funktion  $f: \mathcal{I} \rightarrow \mathcal{I}'$  die  $\mathcal{P}$ -Instanzen in  $\mathcal{P}'$ -Instanzen mit gleicher Ja/Nein-Antwort verwandelt.
- ▶ Sei  $\mathcal{A}'$  ein Algorithmus, der  $\mathcal{P}'$  löst.  
 Löse  $\mathcal{P}$ -Instanz  $J$  mit  $\mathcal{A}(J) := J' := f(J); \text{ return } \mathcal{A}'(J');$   
 → Instanz  $J$  für  $\mathcal{P}$  wird auf eine Instanz  $J'$  für  $\mathcal{P}'$  **reduziert**.
- ▶  $\mathcal{P}'$  kann alle (mittels  $f$  umgewandelten) Instanzen von  $\mathcal{P}$  lösen, aber ggf. noch mehr.
- ▶  $\mathcal{P}$  ist eingeschränkteres Problem als (= Spezialfall von)  $\mathcal{P}'$ .

## Sprachen

Seien  $G_1, G_2$  KF Grammatiken und  $L_1, L_2$  DKF Sprachen.  
Auf Basis der Tatsache, dass PCP unentscheidbar ist, kann man durch Reduktionen<sup>1</sup> zeigen, dass (z.B.) die folgenden Fragen unentscheidbar sind:

- ▶  $\mathcal{L}(G_1) = \mathcal{L}(G_2)$ ?
- ▶  $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$ ?       $L_1 \subseteq L_2$ ?
- ▶  $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$ ?       $L_1 \cap L_2 = \emptyset$ ?
- ▶ Ist  $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$  KF?      Ist  $L_1 \cap L_2$  KF?
- ▶ Ist  $\overline{\mathcal{L}(G_1)}$  KF?      Ist  $\mathcal{L}(G_1)$  DKF?      Ist  $\mathcal{L}(G_1)$  regulär?
- ▶ ...

---

<sup>1</sup>Man verwandelt also eine beliebige PCP-Instanz in entsprechende Grammatiken/Sprachen, und zeigt, dass die PCP-Instanz lösbar ist, genau dann wenn die Antwort auf die Grammatik-Fragestellung **ja** lautet.

## Busy Beaver (Tibor Radó 1962)

**$n$ -Busy Beaver** = Eine TM  $M$  mit  $n$  Zuständen und  $|I| = 2$  die, bei einem anfangs leeren Band, möglichst viele Schritte (=Zustandsübergänge) ausführt **und hält**.

Wie groß ist  $S(n)$ , die Anzahl der Schritte eines  $n$ -Busy Beaver?

**Theorem.**  $S(n)$  ist **nicht-berechenbar**.  $S(n)$  wächst **schneller** als **jede mögliche berechenbare** Funktion.

**Beweis.** Annahme,  $S(n)$  berechenbar oder  $\exists$  berechenbare Funktion  $S'(n) \geq S(n)$ . Wir könnten das Halteproblem einer TM mit  $n$  Zuständen lösen: Berechne die ersten  $S(n)$  bzw.  $S'(n)$  Schritte – wenn noch nicht terminiert, wird die TM nie halten.  $\square$

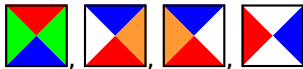
Man weiß:  $S(1) = 1$ ,  $S(2) = 6$ ,  $S(3) = 21$ ,  $S(4) = 107$ ,

**2024:**  $S(5) = 47\,176\,870$  (fast 64 Billionen TMen)

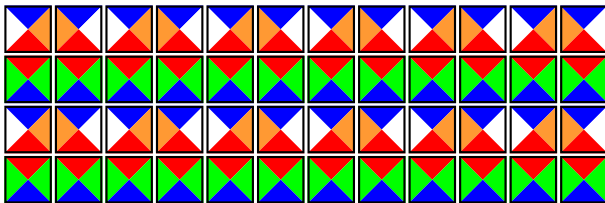
Optimalen Lösungen für  $n \geq 6$  **unbekannt!**  $S(6) \geq 7 \cdot 10^{36534}$

## Wang Parkett, 1/2

**Gegeben:** Eine Menge von möglichen (nicht-rotierbaren) Kachel-Typen:



**Frage:** Kann man mittels dieser Kachel-Typen (beliebig viele Kacheln pro Typ) die komplette unendliche Ebene ausfüllen, so dass sich-berührende Kanten gleich gefärbt sind?



~~**Wang 1961:** Das Problem ist entscheidbar.~~

**Fehlerhafter Beweis!**

Er nahm an: Wenn es eine Lösung gibt, so ist sie **periodisch**.

## Wang Parkett, 2/2 (1966–2019)

**Berger 1966:** Es gibt Instanzen, die nur **aperiodische(!)** Lösungen erlauben. Dies macht das Problem **unentscheidbar**. Sein Beispiel hatte **20 426** Kachel-Typen.

**Jeandel&Rao 2015/19:** Minimale Instanz mit 11 Kacheln, 4 Farben:



## Kolmogorov Komplexität (Андрей Николаевич Колмогоров)

- ▶ Auch genannt **Beschreibungskomplexität** oder **Algorithmischer Informationsgehalt**.
- ▶ Maß für die Strukturiertheit bzw. Zufälligkeit einer Zeichenkette.
- ▶ ca. 1963, unabhängig jeweils auch entdeckt von Solomonov und Chaitin.
- ▶ Verwandt zur Informationstheorie von Shannon.
- ▶ Anwendung in Kompression, etc.

**Gegeben:** Beschreibungssprache (Programmiersprache)  $P$  (meist Turing-vollständig) und eine Zeichenkette  $w$ .

**Gesucht:** Kolmogorov Komplexität  $K_P(w)$

= Länge der kürzestmöglichen Beschreibung von  $w$  mittels  $P$

= Länge des kürzestmöglichen  $P$ -Programms, das  $w$  ausgibt.

## Kolmogorov Komplexität, 2/3

**Beispiel.**  $w = \underbrace{abcabc \dots abc}_{900 \text{ Zeichen}}$

- ▶ `print 'abcabcabcabcabcabc...abc'` → 908 Zeichen
- ▶ `for i=1,..,300: print 'abc'` → 27 Zeichen

**Beobachtung:**  $K_P(w)$  und  $K_Q(w)$  für zwei beliebige Turing-äquivalente Programmiersprachen  $P, Q$  unterscheiden sich maximal um eine **additive** Konstante  $c_{P,Q}$ .

**Beweis.** Da  $P$  und  $Q$  Turing-äquivalent sind, kann man ein (konstant großes)  $P$ -Programm schreiben, das als Eingabe ein  $Q$ -Programm erhält und simuliert. → Eine gute Beschreibung in  $Q$  ergibt also zusammen mit diesem Simulator eine nur konstant größere Beschreibung in  $P$ . □

## Kolmogorov Komplexität, 3/3

**Theorem.** Sei  $P$  eine beliebige Turing-vollständige Programmiersprache.  $K_P(w)$  ist nicht berechenbar.

**Beweis.** Angenommen,  $K_P(w)$  wird von einem Programm  $\mathcal{K}(w)$  berechnet. Betrachte das Programm:

```
 $\mathcal{B}(n) :=$    for  $i = 1, 2, 3, \dots$ :  
              for all strings  $w$  with  $|w| = i$ :  
                if  $\mathcal{K}(w) \geq n$ : return  $w$ 
```

- ▶  $\mathcal{B}(n)$  liefert ein kürzestes Wort mit Komplexität  $\geq n$ .
- ▶  $\mathcal{B}(n)$  (inkl.  $\mathcal{K}(w)$ ) hat konstante Größe (Programmgröße, Größe der Codedatei,...), sagen wir  $n_0$ .
- ▶ Starte  $\mathcal{B}(n_0 + 1)$  und erhalte  $w' \Rightarrow$   
 $w'$  hat Kolmogorov-Komplexität  $\geq n_0 + 1$ , wird aber durch Programm der Länge  $n_0$  erzeugt  $\rightarrow$  Widerspruch.  $\square$

# Game of Life, 1/2

## Zelluläre Automaten

Beliebt für Simulationen (Physik, Straßenverkehr, ...)

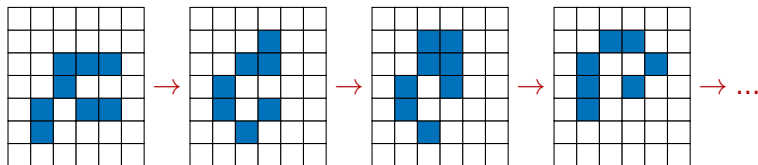
**1940er:** Stanisław Ulam, John von Neumann

**1970:** John Conway (1937–2020; Corona) → **Game of Life:**





Unendliches 2D-Gitter. Jede Zelle ist **tot** □ oder **lebendig** ■.






Zelle  $z$  in Generation  $i + 1$  ist lebendig g.d.w. in Generation  $i$ :

- ▶  $z$  war tot aber hatte 3 (von 8) lebende Nachbarn
- ▶  $z$  war lebendig und hatte 2–3 lebende Nachbarn



## Game of Life, 2/2

Statisches:  ,  , ... Oszillierendes:   $\leftrightarrow$   , ...

Gleiter, z.B.:   $\rightarrow$    $\rightarrow$    $\rightarrow$    $\rightarrow$  

Gleiterkanonen, ...

**Aufgabe:** Gegeben zwei Gitter.

Entsteht das zweite irgendwann aus dem ersten?

**Unentscheidbar!** Denn...

Mit Objekten wie den oberen (und weiteren) lässt sich eine Turingmaschine bauen! Game of Life ist **Turing-vollständig!**

RIP John Conway <https://xkcd.com/2293/>

## Diophantische Gleichungen, 1/2

Diophantos von Alexandria, 250 nChr(?), Bücher „Arithmetica“

**Gegeben:** Polynomgleichung (Koeff.  $\in \mathbb{Z}$ , mehrere Var.).

**Frage:** **ganzahlige** (oder  $\mathbb{N}$ ) Lösung, oder zeige, dass  $\nexists$

Beispiele:

- ▶ Variablen  $x, y$ , Grad 2,  $a_i \in \mathbb{Z}$ :

$$a_1x^2 + a_2xy + a_3y^2 + a_4x + a_5y + a_6 = 0$$

- ▶  $x^2 = 3 \Rightarrow$  keine **ganzahlige** Lsg

- ▶  $x^2 = y \Rightarrow \{(1, 1), (-1, 1), (2, 4), (-2, 4), (3, 9), (-3, 9), \dots\}$

- ▶  $x^2 + y^2 = z^2 \Rightarrow$  Pythagoras!  $\{(3, 4, 5), (5, 12, 13), \dots\}$

- ▶  $x^k + y^k = z^k$  für fixes  $k > 2$ , suche  $x, y, z \in \mathbb{N}_+$

$\Rightarrow$  Großer Fermatscher Satz; Andrew Wiles 1994:  $\nexists$

**Algorithmus?**  $\exists$  für linear, oder bivariat&quadratisch... aber ab Grad 3 unbekannt!

## Diophantische Gleichungen, 2/2

Ist eine gegebene diophantische Polynomgleichung  $P$  lösbar?

**Hilberts zehntes Problem** (1900): Gib Verfahren dafür an.

Juri W. Matijassewitsch, 1970 (22jährig): **Existiert nicht!**

Idee aus den 1950ern, div. Vorarbeiten in den 60ern.

**Kernidee:** Rekursiv aufzählbare Mengen  $\Leftrightarrow$  Diophantische Gleichungen

▶ **„Einfache“ Richtung**  $\Leftarrow$ :

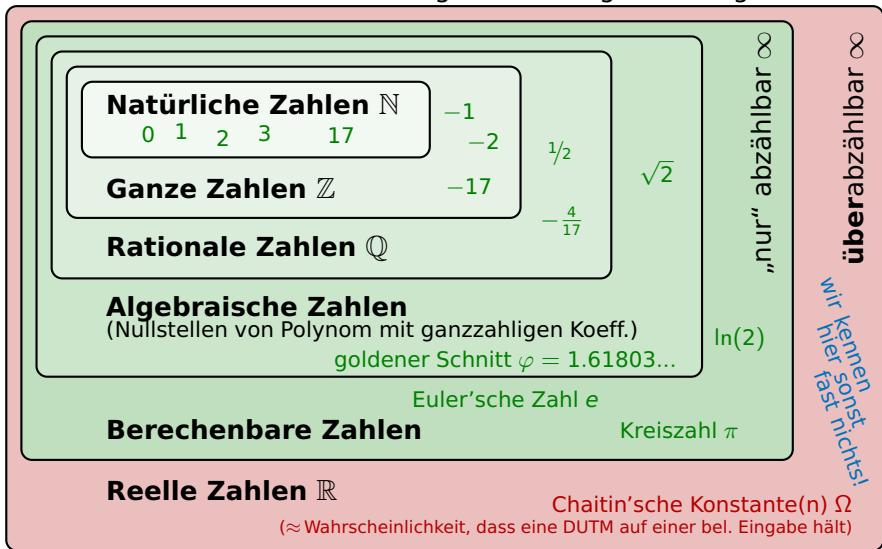
Gegeben eine diophantische Gleichung  $P \rightarrow \exists$  rekursiv aufzählende Funktion  $f$  die die Lösungen (nur Wert der ersten Variable) von  $P$  aufzählt.  $\rightarrow$  Falls  $P$  unlösbar: Algorithmus für  $f$  hält nicht.

▶ **Kompliziertere Richtung**  $\Rightarrow$ :

Gegeben  $f$ , finde eine diophantische Gleichung  $P$  mit genau den passenden Lösungen...

# Die wenigsten Zahlen lassen sich berechnen?!

Zahl ist berechenbar  $\iff \exists$  Alg. für beliebige Genauigkeit



# Komplexitätstheorie

# Komplexitätstheorie

P vs. NP

## Komplexität / Effizienz

Angenommen, wir betrachten ein berechenbares Problem (Kürzester Weg, kürzeste Rundtour, etc.).

### Wie schnell können wir eine Lösung finden?

(vergleiche „Einf. in die Algorithmik“)

- ▶ Obere Schranken durch **Algorithmen**.  
„Ich schaffe es auf jeden Fall in der Zeit  $\mathcal{O}(n^2)$ “
- ▶ Untere Schranken (meist schwerer zu beweisen).  
„Es dauert mindestens  $\Omega(n)$ “

### Was bedeuten eigentlich diese Laufzeitschranken?

Größenordnung der Anzahl der Rechenschritte.

$\mathcal{O}(n^2)$  = Es existiert eine Konstante  $c > 0$ , so dass man bei einer beliebigen Instanz der Größe  $n$  ( $n$  ausreichend groß) maximal  $c \cdot n^2$  Rechenschritte benötigt.

# Maschinenmodell

Laufzeitschranken sind abhängig vom **Maschinenmodell**:

- ▶ Betrachte  $\mathcal{O}(f(n))$ -Algorithmus auf einer Mehrband-TM. Eine TM kann eine Mehrband-TM simulieren, benötigt dann aber  $\mathcal{O}(f(n)^2)$  Zeit.
- ▶ Wie teuer ist es, zwei Zahlen  $a_1, a_2$  zu addieren?
  - ▶ **Logarithmisches Kostenmaß.**  
 $a_i$  benötigt  $\log_2 a_i$  viele Bits  $\rightarrow \mathcal{O}(\log a_1 + \log a_2)$ .
  - ▶ **Uniformes Kostenmaß.**  
 $a_i$  wird in **Speicherzelle** gespeichert (z.B. 64 Bit). Speicherzellen können in konstanter Zeit addiert werden  $\rightarrow \mathcal{O}(1)$ .  
(Problem: Was bei Zahlen die mehr Bits benötigen?)

Formal ist  $\log$  „richtiger“, allerdings aufwendig und „unrealistisch“!  $\rightarrow$  i.d.R. **uniformes Kostenmaß**.

## Komplexitätsklasse $P$

**Definition.** Die Komplexitätsklasse  $P$  umfasst alle **Entscheidungsprobleme**, die sich von einer **deterministischen Turingmaschine** in **polynomieller Zeit** entscheiden lassen.

**Anmerkung.** Die Einschränkung auf Turingmaschinen ist **nicht** notwendig (die auf **Determinismus** schon!)

- ▶ Laufzeiten von TM und Mehrband-TM unterscheiden sich nur durch Quadrieren. → Polynomialität bleibt bestehen.
- ▶ WHILE-/GOTO-Programme haben bis auf Kostenmaß-log-Faktoren gleiche Laufzeiten wie Mehrband-TM.

**Definition.** Die Komplexitätsklasse  $P$  umfasst alle **Entscheidungsprobleme**, die sich mithilfe einer **deterministischen Programmiersprache** in **polynomieller Zeit** entscheiden lassen.

## Beispiele für $P$ ?

Liegen die folgenden Probleme in  $P$ ?

- ▶ Zusammenhangstest von einem Graph?  
→ **Ja!** Mit Tiefensuche sogar in Linearzeit.
- ▶ Halteproblem?  
→ **Nein!** Problem nicht entscheidbar.
- ▶ DTM mit anfangs leerem Band hält nach max.  $k$  Schritten?  
→ **Nein!** Problem nicht in poly. Zeit lösbar.  
(ohne Bew.; Eingabelänge  $n = \lceil \log k \rceil \Rightarrow k \approx 2^n$  Schritte)
- ▶ Kürzester Weg in einem Graph?  
→ **Nein!** Polynomiell, aber kein Entscheidungsproblem.
- ▶ Existiert in einem Graph ein Weg mit Maximallänge  $k$ ?  
→ **Ja!** Berechne in poly-Zeit den kürzesten Weg und vergleiche mit  $k$ .
- ▶ Existiert in einem Graph eine Rundtour (Traveling Salesman) mit Maximallänge  $k$ ?  
→ **Unbekannt!** Nur exponentielle Algorithmen bekannt.

## Komplexitätsklasse $NP$

**Definition.** Die Komplexitätsklasse  $P$  umfasst alle **Entscheidungsprobleme**, die sich von einer **deterministischen Turing-Maschine** in **polynomieller Zeit** entscheiden lassen.

**Definition.** Die Komplexitätsklasse  $NP$  umfasst alle **Entscheidungsprobleme**, die sich von einer **nicht-deterministischen Turing-Maschine** in **polynomieller Zeit** entscheiden lassen.

**Beobachtung.**  $P \subseteq NP$ .

## Beispiel: SAT

### Erfüllbarkeitsproblem – (engl. **Satisfiability**, **SAT**)

**Gegeben:** Formel  $F$  der Aussagenlogik (in KNF).

**Gefragt:** Gibt es eine Variablenbelegung, so dass  $F = \text{true}$ ?

### KNF = Konjunktive Normalform

Jede aussagenlogische Formel der Größe  $n$  lässt sich (ggf. durch Einführen neuer Variablen) in eine  $\mathcal{O}(n)$  große KNF umschreiben.

- ▶ Eine Formel in KNF besteht aus einer Menge von **Klauseln** die durch  $\wedge$  (UND) verbunden sind.
- ▶ Jede Klausel besteht aus einer Menge von **Literalen** (Variablen  $x_j$  bzw. ihre Negationen  $\bar{x}_j := \neg x_j$ ) die durch  $\vee$  (OR) verbunden sind.

Beispiel:  $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4)$

## Beispiel: SAT — Deterministisch

### Deterministischer Exponentialzeit-Algorithmus.

- Am Anfang sind alle Variablen frei (=nicht auf *true* oder *false* fixiert). Wenn  $K$  eine Klausel ist, so definieren wir
- $$K' = \begin{cases} \textit{true} & \text{falls mindestens ein fixiertes Literal } \textit{true} \text{ ist,} \\ \textit{false} & \text{falls alle Literale fixiert und } \textit{false} \text{ sind,} \\ ? & \text{sonst.} \end{cases}$$
- 1 Solange es eine Klausel  $K$  mit  $K' = ?$  und einer einzigen freien Variable  $x_i$  gibt: Fixiere  $x_i$  so, dass  $K' = \textit{true}$ .
  - 2 Falls  $\exists$  Klausel  $K$  mit  $K' = \textit{false}$ : **return 'false'**.
  - 3 Falls  $K' = \textit{true}$  für alle Klauseln  $K$ : **return 'true'**
  - 4 Wähle freie Variable  $x_i$ , starte Algorithmus inklusive derzeitigen Fixierungen **zweimal neu** (mit  $x_i = \textit{true}$  bzw.  $x_i = \textit{false}$ ), und erhalte davon  $\text{ret}_{x_i=\textit{true}}$  und  $\text{ret}_{x_i=\textit{false}}$ .  
**return**  $\text{ret}_{x_i=\textit{true}} \vee \text{ret}_{x_i=\textit{false}}$

## Beispiel: SAT — Nicht-deterministisch

### Nicht-deterministischer Polynomialzeit-Algorithmus.

- ▶ Am Anfang sind alle Variablen frei (=nicht auf *true* oder *false* fixiert). Wenn  $K$  eine Klausel ist, so definieren wir
 
$$K' = \begin{cases} \textit{true} & \text{falls mindestens ein fixiertes Literal } \textit{true} \text{ ist,} \\ \textit{false} & \text{falls alle Literale fixiert und } \textit{false} \text{ sind,} \\ ? & \text{sonst.} \end{cases}$$
- 1** Solange es eine Klausel  $K$  mit  $K' = ?$  und einer einzigen freien Variable  $x_i$  gibt: Fixiere  $x_i$  so, dass  $K' = \textit{true}$ .
- 2** Falls  $\exists$  Klausel  $K$  mit  $K' = \textit{false}$ : **return 'false'**.
- 3** Falls  $K' = \textit{true}$  für alle Klauseln  $K$ : **return 'true'**
- 4** Wähle freie Variable  $x_i$ , und **rate** den richtigen Wert für  $x_i$ . Fixiere  $x_i$  entsprechend und goto **1**.

## Beispiel: SAT — Nicht-deterministisch

### Nicht-deterministischer Polynomialzeit-Algorithmus.

- ▶ Am Anfang sind alle Variablen frei (=nicht auf *true* oder *false*)
- Ein nicht-deterministischer Algorithmus berechnet **alle** möglichen Konsequenzen des Raten **gleichzeitig**.
- Er berechnet also **auch** die Konsequenzen des **Richtig-Ratens**.
- 1** S Bei Nicht-Determinismus reicht es für eine positive Antwort aus, wenn **einer** aller Rechenwege erfolgreich ist.
- 2** Fä
- 3** Fä **Aber Achtung:** Es darf durch Falsch-Raten kein inkorrektes Ergebnis entstehen! – Nicht-Determinismus versucht ja **alle** möglichen Wege.
- 4** W
- Fi ⇒ Wir raten nur Dinge der **Lösung** (oder allgemeiner: des **Ja-Zeugens**, siehe nächste Folie)

## Beispiel: SAT — Nicht-deterministisch

### Ja-Zeugen validieren.

Sei  $\mathcal{X}$  ein beliebiges Entscheidungsproblem und  $J$  eine Ja-Instanz für  $\mathcal{X}$ . Ein **(Ja-)Zeuge** für  $J$  ist ein Datenkonstrukt (in der Regel eine **Lösung** von  $J$ ) als Beweis, dass es sich um eine Ja-Instanz handelt.

**Theorem.** Ein Problem ist in  $NP$ , genau dann wenn das Problem, einen gegebenen Ja-Zeugen auf „Echtheit“ zu überprüfen, in  $P$  ist.

**Beispiel.** Sei  $\mathcal{X} = \text{SAT}$ , und  $J = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$ .

Die Belegung  $x'_1 = \text{true}$ ,  $x'_2 = \text{false}$  ist ein Zeuge dafür, dass die Formel erfüllbar ist.

Wir setzen  $x'_1, x'_2$  in die Formel ein, und erhalten in polynomieller (sogar linearer) Zeit

$(\text{true} \vee \text{false}) \wedge (\text{false} \vee \text{true}) \vdash \text{true} \wedge \text{true} \vdash \text{true}.$

## Beispiel: SAT — Nicht-deterministisch

### Ja-Zeugen validieren.

Sei  $\mathcal{X}$  ein beliebiges Entscheidungsproblem und  $J$  eine Ja-Instanz für  $\mathcal{X}$ . Ein **(Ja-)Zeuge** für  $J$  ist ein Datenkonstrukt (in der Regel eine **Lösung** von  $J$ ) als Beweis, dass es sich um eine Ja-Instanz handelt.

**Theorem.** Ein Problem ist in  $NP$ , genau dann wenn das Problem, einen gegebenen Ja-Zeugen auf „Echtheit“ zu überprüfen, in  $P$  ist.

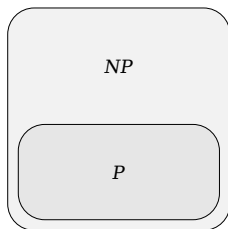
**Beweis.** Spiele den Prüfalgorithmus „ohne Lösung“ durch. Immer wenn der Algorithmus einen Teil der Lösung benötigt (den wir vorher noch nicht geraten haben) „rate“ (Nicht-Determinismus!) den richtigerweise notwendigen Teil der Lösung. □

**Beobachtung:** Ein Zeuge kann/darf nur polynomiell groß sein!

## P vs. NP

Wir wissen derzeit nur:  $P \subseteq NP$ .

Man kennt **viele** Probleme aus  $NP$ , für die man (trotz jahrzehntelanger Forschung!) **keine polynomiellen Algorithmen gefunden** hat  
⇒ Starke **Vermutung**:  $P \subsetneq NP$ .



Die größte offene Frage der Informatik:

Gilt  $P = NP$  oder  $P \neq NP$ ?

Millennium Prize des Clay Mathematics Institutes:

**1.000.000 US-Dollar**

# NP und Polynomiell vs. Exponentiell

## NP $\neq$ nicht-polynomiell:

- ▶ NP steht für **Nicht-deterministisch Polynomiell!**
- ▶ Es bedeutet **nicht** nicht-polynomiell!
- ▶ Es wäre möglich ( $P = NP$ ), dass NP-Probleme auch deterministisch polynomiell lösbar sind!
- ▶ Wir **vermuten**, dass NP-Probleme nicht in polynomieller Zeit gelöst werden können (also, gemäß unterer Sprachregelung, exponentielle Zeit benötigen)

## Sprachregelung:

- ▶ Formal korrekt hat eine exponentielle Funktion die Gestalt  $c^n$ , für eine Konstante  $c > 1$ .
- ▶ Es gibt nicht-polynomielle Funktionen, die schwächer wachsen als exponentielle, z.B.  $c^{\text{poly } \log n}$
- ▶ Im allgemeinen Sprachgebrauch bedeutet **exponentiell** ab nun immer **nicht polynomiell beschränkt**.

## P vs. NP: Sinn

Macht **polynomiell** vs. **exponentiell** als Unterscheidung eigentlich Sinn?

- ▶ Schließlich ist z.B.  $n^{1000} \geq 1.0001^n$  für alle realistisch vorkommenden Werte von  $n$ !

### Begründungen für die Unterscheidung:

- ▶  $P/NP$  ist **unabhängig** vom Maschinenmodell.
- ▶  $P$  ist abgeschlossen bzgl. polynomiell beschränkter Funktionen.
- ▶ Für viele Praxisprobleme ist „polynomiell“ meist ein Polynom mit kleinem Grad, also auch „effizient in der Praxis“. Dagegen haben exponentielle Algorithmen **tendenziell** tatsächlich sehr böse Praxislaufzeiten.
- ▶ Man kennt keine bessere formal saubere Unterscheidung zwischen „in der Praxis gut lösbaren“ Problemen, und solchen die das nicht sind.

# Komplexitätstheorie

## NP-Vollständigkeit

# NP-vollständig

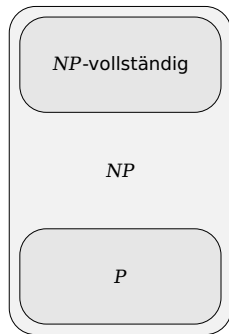
## Definition.

Ein Problem  $\mathcal{X}$  ist **NP-vollständig** (engl. **NP-complete, NPC**) wenn

- 1  $\mathcal{X} \in NP$ , und
- 2 Jedes andere Problem  $\mathcal{Y} \in NP$  lässt sich **deterministisch** und **in polynomieller Zeit** auf  $\mathcal{X}$  **reduzieren**<sup>1</sup>.

schwer

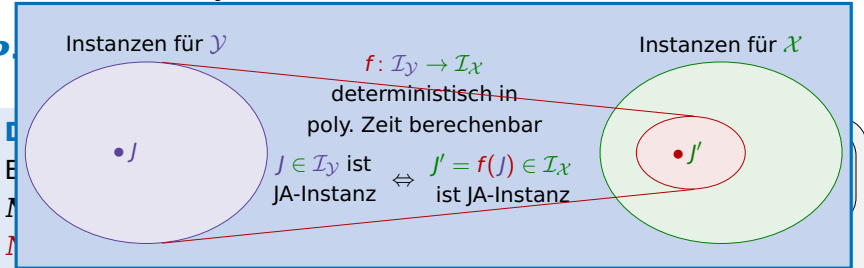
leicht



**Reduzieren?** Man kann aus jeder Problem Instanz  $J$  für  $\mathcal{Y}$  eine Problem Instanz  $J'$  für  $\mathcal{X}$  gewinnen, die **ja** liefert, **genau dann wenn**  $J$  eine **ja**-Instanz ist (und daher auch analog für **nein**).

<sup>1</sup>Im Folgenden benutzen wir einfach nur den Begriff „**reduzieren**“ für „**deterministisch und in polynomieller Zeit reduzieren**“.

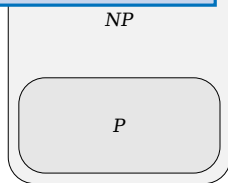
NP



- $\mathcal{X} \in NP$ , und
- Jedes andere Problem  $\mathcal{Y} \in NP$  lässt sich **deterministisch** und **in polynomieller Zeit** auf  $\mathcal{X}$  **reduzieren**<sup>1</sup>.

Komplex

leicht



**Reduzieren?** Man kann aus jeder Probleminstance  $J$  für  $\mathcal{Y}$  eine Probleminstance  $J'$  für  $\mathcal{X}$  gewinnen, die **ja** liefert, **genau dann wenn**  $J$  eine **ja**-Instanz ist (und daher auch analog für **nein**).

<sup>1</sup>Im Folgenden benutzen wir einfach nur den Begriff „**reduzieren**“ für „**deterministisch und in polynomieller Zeit reduzieren**“.

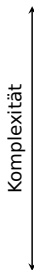
# NP-vollständig

## Definition.

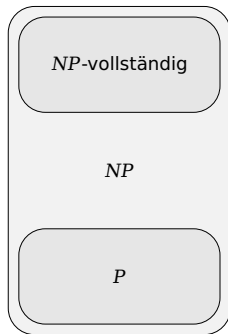
Ein Problem  $\mathcal{X}$  ist **NP-vollständig** (engl. **NP-complete, NPC**) wenn

- 1  $\mathcal{X} \in NP$ , und
- 2 Jedes andere Problem in  $NP$  lässt sich auf  $\mathcal{X}$  **reduzieren**.

schwer



leicht



- ▶  $NP$ -vollständige Probleme sind die **schwersten** Probleme in  $NP$ .
- ▶ Angenommen man könnte für **ein**  $NP$ -vollständiges Problem  $\mathcal{X}$  zeigen, dass  $\mathcal{X} \in P$ . Dann wäre  $P = NP$ .

## NP-intermediate

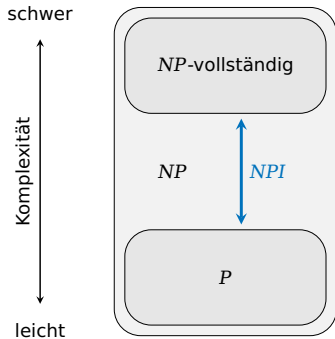
### Faszinierend!

**Fast alle** Probleme  $\in NP$  für die wir nicht  $\in P$  wissen, sind tatsächlich *NP-vollständig*!

### Theorem (Richard Ladner, 1975).

Falls  $P \neq NP$ , dann enthält  $NP \setminus P$  auch nicht-*NP-vollständige* Probleme. (ohne Beweis.)

$\Rightarrow$  **NP-intermediate (NPI)**



Im Beweis werden sehr künstliche Probleme für *NPI* generiert.

Es gibt nur sehr wenige **natürliche** Probleme, die Kandidaten für *NPI* sind:

Graph-Isomorphie, Faktorisierung,  
Diskreter Logarithmus

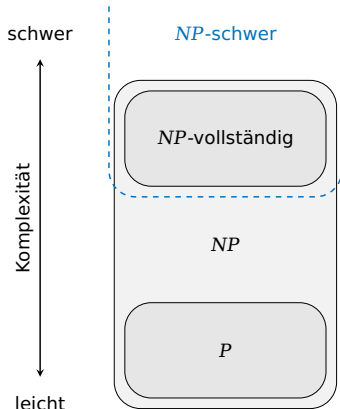
# NP-schwer

**Definition.** Ein Entscheidungsproblem  $\mathcal{X}$  ist **NP-schwer** (*NP-hart*, engl. *NP-hard*) wenn sich jedes Problem aus *NP* auf  $\mathcal{X}$  reduzieren lässt.

→  $\mathcal{X}$  ist **mindestens** so schwer wie die schwersten Probleme in *NP*.

## Beispiele:

- ▶ Wir werden noch sehen, dass *SAT* *NP*-vollständig ist.  
→ *SAT* ist *NP*-schwer.
- ▶ Das Halteproblem  $\notin$  *NP*, sondern sogar unentscheidbar  
→ Halteproblem ist *NP*-schwer.
- ▶ Graph-Zusammenhangstest  $\in$  *P*  
→ Graph-Zusammenhangstest ist nicht *NP*-schwer.



# Suchprobleme, Optimierungsprobleme

**Bisher:** Immer **Entscheidungsprobleme**, da sich diese **formal einfach präzise** als Wortprobleme über Sprachen interpretieren lassen.

**Aber:** Viele Probleme, die wir in der Praxis betrachten, sind **keine** Entscheidungsprobleme:

- ▶ Algorithmen, die eine **Lösung** für ein Problem suchen (nicht nur eine ja/nein-Antwort) heißen **Suchprobleme**.  
z.B. Sortieren, Kürzester Weg, Traveling Salesman, ...
- ▶ Die vielleicht wichtigsten Suchprobleme sind **(kombinatorische) Optimierungsprobleme**:<sup>1</sup>  
Man sucht eine Lösung mit optimalem Zielfunktionswert.  
z.B. Kürzester Weg, Traveling Salesman, ...

---

<sup>1</sup>Formal kann man definieren, dass Suchprobleme **Lösungen** suchen, und Optimierungsprobleme nur den **optimalen Zielfunktionswert**. Für uns ist diese Unterscheidung nun nicht hilfreich.

# Optimierungsprobleme und zugehörige Entscheidungsprobleme

Ein **(kombinatorisches) Optimierungsproblem** (Min- oder Maximierungsproblem)  $\mathcal{X}$  sucht für eine **Instanz**  $J$  nach einer **Lösung**  $\mathcal{L}$ , die gewisse **Eigenschaften** (spezifiziert durch  $\mathcal{X}$ ; z.B. Pfad, Rundtour, . . .) erfüllt, und dabei unter allen **zulässigen** Lösungen diejenige mit **optimalem** (minimum/maximum) **Zielfunktionswert**  $z(\mathcal{L})$  ist.

**(Kombinatorisch  $\approx$  nur endlich viele zulässige Lösungen)**

Annahme,  $\mathcal{X}$  sei ein Minimierungsproblem.

Das zu  $\mathcal{X}$  **zugehörige Entscheidungsproblem**  $(\mathcal{X}, k)$  fragt, ob für die Instanz  $J$  eine  $\mathcal{X}$ -zulässige Lösung  $\mathcal{L}$  **existiert**, so dass  $z(\mathcal{L}) \leq k$ .

(Bei Maximierungsproblem:  $z(\mathcal{L}) \geq k$ )

## Optimierung vs. Entscheidung

Sei  $\mathcal{X}$  ein Minimierungsproblem und  $(\mathcal{X}, k)$  das zugehörige Entscheidungsproblem.

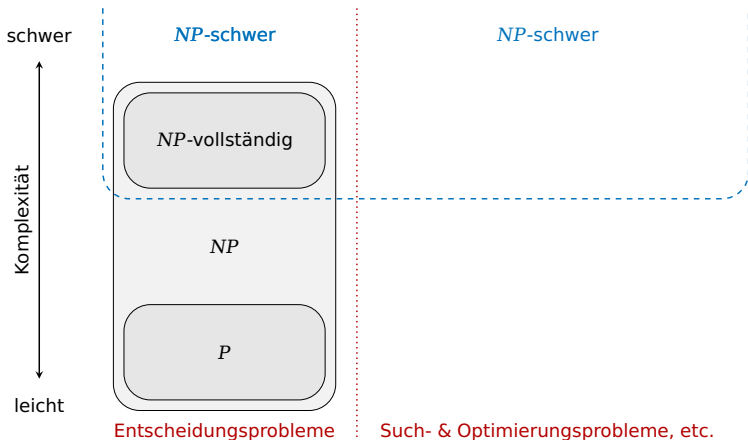
- ▶ **Falls**  $\mathcal{X}$  in poly-Zeit berechenbar:  
Prüfen von  $z(\mathcal{L}) \leq k$  liefert direkt Antwort für  $(\mathcal{X}, k)$ .
- ▶ **Falls**  $(\mathcal{X}, k) \in P$ :  
Berechne  $(\mathcal{X}, k)$  für verschiedene Werte von  $k$   
→ Finde optimales  $k$  mittels binärer Suche  
→ Lösen von  $\mathcal{X}$  ist nur polynomiell (logarithmisch) mehr Aufwand als  $(\mathcal{X}, k) \rightarrow$  noch immer polynomiell.

⇒ Für Such- und Optimierungsprobleme kann man **äquivalente Klassen** zu  $P$ ,  $NP$ , etc. definieren. Dies ist jedoch formal fummelig, daher benutzt man zur Komplexitätsbestimmung meist einfach das zugehörige Entscheidungsproblem.

# NP-schwer, nochmal

## Erweiterte Definition von NP-schwer.

Ein allgemeines Problem  $\mathcal{X}$  ist **NP-schwer**, wenn sich jedes NP-vollständige Entscheidungsproblem auf  $\mathcal{X}$  reduzieren lässt. ( $\rightarrow \mathcal{X}$  mindestens so schwer wie NP-vollständig)



# **Komplexitätstheorie**

Erstes NP-vollständiges Problem

## Beweisbau

Um zu beweisen, dass ein Entscheidungsproblem  $\mathcal{X}$  NP-vollständig ist, muss man

- 1 zeigen, dass es in NP liegt (NP-membership), und
  - 2 zeigen, dass sich jedes andere Problem  $\mathcal{Y} \in NP$  auf  $\mathcal{X}$  reduzieren lässt (NP-hardness).
- 1 ist meist trivial („Validieren eines Ja-Zeugen“).
- 2 ist ad hoc nicht klar, wie man das tut...

**Annahme:** Man kennt ein NP-vollständiges Problem  $\mathcal{Y}^*$

- Jedes andere Problem  $\mathcal{Y}$  aus NP lässt sich auf  $\mathcal{Y}^*$  reduzieren.
- **Es reicht aus,  $\mathcal{Y}^*$  auf  $\mathcal{X}$  zu reduzieren!**
- alle anderen Probleme lassen sich dann über  $\mathcal{Y}^*$  bis nach  $\mathcal{X}$  reduzieren.

## Cook-Levin: SAT ist NP-vollständig

- ▶ Wir müssen also nur für **ein einziges** Problem  $\mathcal{Y}^*$  zeigen, dass sich **alle** Probleme aus  $NP$  auf  $\mathcal{Y}^*$  reduzieren lassen.
- ▶ Ab dann, für ein Problem  $\mathcal{X}$ , nur mehr **eine** Reduktion von **irgendeinem** bekannten  $NP$ -vollständigen Problem (am Anfang nur  $\mathcal{Y}^*$ ) auf  $\mathcal{X}$ .

Dieses erste wichtige  $NP$ -vollständige Problem ist **SAT**:

**Cook-Levin Theorem.** SAT ist  $NP$ -vollständig.

**Beweis.** Folgt auf den nächsten Folien.

Das Theorem wurde 2x **unabhängig** von einander entdeckt:

- ▶ **Stephen Cook** (USA), 1971
- ▶ **Leonid Levin** (UdSSR), 1973 (veröffentlicht, Ergebnis schon ein paar Jahre vorher erwähnt)
- ▶ **Levin betrachtete Such- statt Entscheidungsprobleme.**

## SAT ist NP-vollständig: Beweis, 1/7

**Cook-Levin Theorem.** SAT ist NP-vollständig.

**Beweisteil 1:** SAT ist in NP.

Gegeben eine SAT-Instanz der Größe  $n$ , mit  $k \leq n$  Booleschen Variablen.

Rate (Nicht-Determinismus!) die richtige Belegung der Variablen, und überprüfe ob die Formel durch diese Belegung erfüllt ist.

(Alle  $2^k$  Möglichkeiten werden „gleichzeitig“ überprüft, also auch die erfüllende Belegung, falls sie existiert)

Dieser (nicht-deterministische) Algorithmus benötigt nur polynomielle (lineare!) Zeit.

$\Rightarrow \text{SAT} \in \text{NP}$ .



## SAT ist NP-vollständig: Beweis, 2/7

**Cook-Levin Theorem.** SAT ist NP-vollständig.

**Beweisteil 2: SAT ist NP-schwer.**

- ▶  $\mathcal{X}$  = beliebiges Entscheidungsproblem (Sprache) aus NP.
- ▶ Es existiert eine polynomiell-beschränkte nicht-determ. Turingmaschine  $M := M_{\mathcal{X}}$  die  $\mathcal{X}$  entscheidet.
- ▶ Sei  $w = \sigma_0\sigma_1\sigma_2 \dots \sigma_{n-1}$  eine Problem Instanz für  $\mathcal{X}$  (=initiale Bandbelegung für  $M$ ).
- ▶ Sei  $\pi(n)$  das Polynom, dass die Laufzeit von  $M$  bei Eingabelänge  $n$  beschränkt.
- ▶ **Wir können annehmen:** Falls  $M$  akzeptiert, dann nach **genau**  $\pi(n)$  Schritten (vorher nichts-ändernde Endlosschleife, die nicht-deterministisch verlassen wird).

→ baue SAT-Instanz, die **erfüllbar** ist **genau dann wenn**  $w \in \mathcal{X}$

## SAT ist NP-vollständig: Beweis, 3/7

Die SAT-Instanz benutzt die folgenden Booleschen Variablen für alle mögl. TM-Zustände  $Z \in \mathcal{Z}$ , Zeitpunkte  $0 \leq t \leq \pi(n)$ , Bandpositionen  $-\pi(n) \leq p \leq \pi(n)$  und Symbole  $\sigma \in \Gamma$ .

Zum Zeitpunkt  $t...$

---

$Z_{t,Z} = \text{true} \iff$  ...befindet sich  $M$  im Zustand  $Z$

$SL_{t,p} = \text{true} \iff$  ...steht der SL-Kopf an Position  $p$

$B_{t,p,\sigma} = \text{true} \iff$  ...steht an Bandposition  $p$  das Symbol  $\sigma$

Unsere SAT-Formel  $F$  hat die Gestalt  $R \wedge S \wedge E \wedge U_1 \wedge U_2$ :

- ▶ **R**andbedingungen
- ▶ **S**tartsituation
- ▶ **E**ndsituation
- ▶ **Ü**bergänge ( $U_1, U_2$ )

## SAT ist NP-vollständig: Beweis, 4/7

### Randbedingungen $R$

Zu jedem Zeitpunkt ist der Zustand, die Position des SL-Kopfs und der Bandinhalt eindeutig:

$$R := \bigwedge_t \bigvee_z (Z_{t,z}) \wedge \bigwedge_t \bigvee_p (SL_{t,p}) \wedge \bigwedge_t \bigwedge_p \bigvee_\sigma (B_{t,p,\sigma})$$

wobei

$$\bigvee_i (x_i) := \bigvee_i x_i \wedge \bigwedge_{i \neq j} \neg(x_i \wedge x_j) = \text{genau ein } x_i \text{ ist } \textit{true}.$$

**Beobachtung.**  $R$  ist nur polynomiell groß und kann in polynomieller Zeit erstellt werden.

## SAT ist NP-vollständig: Beweis, 5/7

### Startsituation $S$

Variablenbelegung zum Zeitpunkt  $0$  soll der Anfangssituation der TM entsprechen (Eingabe:  $w = \sigma_0\sigma_1\sigma_2 \dots \sigma_{n-1}$ ):

$$S := Z_{0, Z_{\text{start}}} \wedge SL_{0,0} \wedge \bigwedge_{p=0}^{n-1} B_{0,p,\sigma_p} \wedge \bigwedge_{p<0} B_{0,p,\square} \wedge \bigwedge_{p \geq n} B_{0,p,\square}$$

### Endsituation $E$

Variablenbelegung zum Zeitpunkt  $\pi(n)$  soll „ $M$  in einem Endzustand“ entsprechen:

$$E := \bigvee_{Z \in Z_{\text{end}}} Z_{\pi(n), Z}$$

**Beobachtung.**  $S$  und  $E$  sind nur polynomiell groß und können in polynomieller Zeit erstellt werden.

## SAT ist NP-vollständig: Beweis, 6/7

### Übergänge $U_1$

Zwischen zwei Zeitpunkten ändert sich

- ▶ der aktuelle Zustand,
- ▶ das Symbol unter dem SL-Kopf und
- ▶ die Position des SL-Kopfs.

Die Menge  $\delta(Z, \sigma) \subseteq \mathcal{Z} \times \Gamma \times \{-1, 0, 1\}$  von 3-Tupeln (Zustand, neues Bandsymbol, SL-Kopf Bewegung) bezeichnet die möglichen Übergänge, ausgehend vom Zustand  $Z$  wenn  $\sigma$  vom Band gelesen wird.

$$U_1 := \bigwedge_{t, Z, p, \sigma} \left( (Z_{t,Z} \wedge SL_{t,p} \wedge B_{t,p,\sigma}) \rightarrow \bigvee_{\substack{(Z', \sigma', d) \\ \in \delta(Z, \sigma)}} Z_{t+1, Z'} \wedge SL_{t+1, p+d} \wedge B_{t+1, p, \sigma'} \right)$$

## SAT ist NP-vollständig: Beweis, 7/7

### Übergänge $U_2$

Das Band bleibt an allen anderen Stellen unverändert:

$$U_2 := \bigwedge_{t,p,\sigma} \left( (B_{t,p,\sigma} \wedge \neg SL_{t,p}) \rightarrow B_{t+1,p,\sigma} \right)$$

**Beobachtung.**  $U_1$  und  $U_2$ , und damit auch insgesamt  $F = R \wedge S \wedge E \wedge U_1 \wedge U_2$ , sind nur polynomiell groß und können in polynomieller Zeit erstellt werden.

### Hauptbeobachtung.

- Akzeptierender Rechenweg für  $(M, w) \Rightarrow$  erfüll. Beleg. für  $F$ .
  - Erfüllende Belegung für  $F \Rightarrow$  akzept. Rechenweg für  $(M, w)$ .
- $\Rightarrow$  SAT in Zeit  $\mathcal{O}(f(n))$  lösbar  $\rightarrow$  **jedes** Problem aus NP (kodiert als  $(M, w)$ ) in Zeit  $\mathcal{O}(f(\text{poly}(n)))$  lösbar.  $\square$

**Konsequenz:** Angenommen, SAT wäre in det. Poly.zeit lösbar  $\rightarrow$  **jedes** Problem aus NP wäre in det. Polynomialzeit lösbar!

# Komplexitätstheorie

Weitere NP-vollständige Probleme

## Beweisen von *NP*-Vollständigkeit

Wir haben bereits besprochen:

- ▶ Wir müssen nur für **ein einziges** Problem  $\mathcal{Y}^*$  zeigen, dass sich **alle** Probleme aus *NP* auf  $\mathcal{Y}^*$  reduzieren lassen.
- ▶ Ab dann, für ein Problem  $\mathcal{X}$ , nur mehr **eine** Reduktion von **irgendeinem** bekannten *NP*-vollständigen Problem (am Anfang nur  $\mathcal{Y}^*$ ) auf  $\mathcal{X}$ .

Nun, dank Cook-Levin:  $\mathcal{Y}^* = \text{SAT}$ .

Ab nun gehts einfacher...

## 3SAT

**Gegeben:** Eine aussagenlogische Formel  $F$  in KNF,  
wobei jede Klausel aus maximal 3 Literalen besteht.

**Gefragt:** Ist  $F$  erfüllbar?

**Theorem.** 3SAT ist NP-vollständig.

**Beweis** ( $3SAT \in NP$ ). Trivial, da 3SAT Spezialfall von SAT.

**Beweis** (3SAT ist NP-schwer) durch **Reduktion von SAT**.

Gegeben eine SAT-Formel (nach polynom. Aufwand: in KNF).

Ersetze jede Klausel  $(l_1 \vee l_2 \vee l_3 \vee \dots \vee l_k)$  mit  $k \geq 4$  durch

$k$  Klauseln mit jeweils maximal 3 Literalen. Dabei benutzen

wir  $k - 1$  **neue** Variablen  $y_1, \dots, y_{k-1}$ : (Sinn:  $y_i = \bigvee_{j \leq i} l_j$ )

$(l_1, \bar{y}_1), (y_1, l_2, \bar{y}_2), (y_2, l_3, \bar{y}_3), \dots, (y_{k-2}, l_{k-1}, \bar{y}_{k-1}), (y_{k-1}, l_k)$

**Behauptung:** SAT Instanz erfüllbar  $\Leftrightarrow$  3SAT Instanz erfüllbar.

→ Wenn man 3SAT lösen kann, löst man mit nur  
polynomiellem Mehraufwand auch SAT!



## SUBSETSUM

**Gegeben:** Eine Menge von natürlichen Zahlen

$$A = \{a_1, a_2, \dots, a_n\} \text{ und ein } b \in \mathbb{N}.$$

**Gefragt:** Existiert eine Teilmenge  $S \subseteq A$  mit  $\sum_{a \in S} a = b$ ?

**Theorem.** SUBSETSUM ist NP-vollständig.

**Beweis** (SUBSETSUM  $\in$  NP). Gegeben ein Zeuge  $S$ , prüfe  $S \subseteq A$ , berechne die Summe und vergleiche mit  $b$ .

**Beweis** (NP-schwer) durch **Reduktion von 3SAT**.

- ▶ Sei  $F$  eine 3SAT-Formel mit  $k$  Klauseln  $K_1, \dots, K_k$  und insgesamt  $r$  Variablen  $x_1, \dots, x_r$ .
- ▶ Jede Zahl in unserer generierten SUBSETSUM-Instanz hat  $k + r$  Stellen im Dezimalsystem:  $k$  vordere,  $r$  hintere.
- ▶ Wir setzen  $b := \underbrace{444 \dots 4}_k \underbrace{111 \dots 1}_r$ .

## SUBSETSUM, Fortsetzung

3SAT-Instanz  $F$ : Klauseln  $K_1, \dots, K_k$ , Variablen  $x_1, \dots, x_r$ .

⇒ SUBSETSUM-Instanz: jede Zahl  $k$  vordere,  $r$  hintere Stellen.

$$\blacktriangleright b := \underbrace{444 \dots 4}_{k \text{ Stellen}} \underbrace{111 \dots 1}_{r \text{ Stellen}}$$

- ▶ Für jede Variable  $x_i$  erzeuge zwei Zahlen  $a_i, \bar{a}_i$ :
  - ▶ **Hintere Stellen:** die  $i$ -te Stelle (von  $a_i$  und  $\bar{a}_i$ ) ist  $1$ , alle anderen  $0$ .
  - ▶ **Vordere Stellen:** die  $j$ -te Stelle von  $a_i/\bar{a}_i$  ist  $1$ , genau dann wenn das Literal  $x_i/\bar{x}_i$  in Klausel  $K_j$  vorkommt.
- ▶ Für jedes  $1 \leq j \leq k$  erzeuge zwei Zahlen  $c_j, d_j$  die an allen Stellen  $0$  sind; nur an Stelle  $j$  hat  $c_j$  eine  $1$  und  $d_j$  eine  $2$ .

⇒ SUBSETSUM-Instanz ( $A := \{a_i, \bar{a}_i\}_{1 \leq i \leq r} \cup \{c_j, d_j\}_{1 \leq j \leq k}, b$ )

**Behauptung:** SUBSETSUM  $(A, b)$  erfüllbar  $\Leftrightarrow$  3SAT  $F$  erfüllbar.

⇒ Beweis der Behauptung: siehe nächste Folie

## SUBSETSUM, Fortsetzung

**Beispiel.**  $F = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_1 \vee x_2 \vee x_4).$

$a_1$  100 0001

$\bar{a}_1$  001 0001

$a_2$  101 0010

$\bar{a}_2$  010 0010

$a_3$  000 0100

$\bar{a}_3$  110 0100

$a_4$  011 1000

$\bar{a}_4$  000 1000

---

$c_1$  100 0000

$d_1$  200 0000

$c_2$  010 0000

$d_2$  020 0000

$c_3$  001 0000

$d_3$  002 0000

---

$b$  444 1111

- ▶ Bei Addition entsteht **kein** Übertrag.
- ▶ **Hintere Stellen:** Um  $b$  zu erhalten, muss man, für jedes  $i$ , immer **entweder**  $a_i$  **oder**  $\bar{a}_i$  wählen.  $\iff$  "Variable ist *true* oder *false*".
- ▶ **Vordere Stellen:** Um (inkl.  $c_j$  und  $d_j$ ) auf 4 zu kommen, muss für jede Stelle  $j$  (=für jede Klausel) mindestens ein passendes  $a_i, \bar{a}_i$  gewählt werden ( $\rightarrow$  mind. 1 Literal *true*, max 3 pro Klausel).
- ▶  $\iff$ : Wähle  $a_i/\bar{a}_i$  gemäß Belegung von  $x_i$  & fülle vordere Stellen mit  $c_j, d_j$  bis 4.
- ▶  $\implies$ : Setze  $x_i$  gemäß Auswahl  $a_i/\bar{a}_i$  in  $S$ .  $\square$

## PARTITION

**Gegeben:** Menge von natürlichen Zahlen  $A = \{a_1, a_2, \dots, a_n\}$ .

**Gefragt:** Existiert Teilmenge  $S \subseteq A$  mit  $\sum_{a \in S} a = \sum_{a \in A \setminus S} a$ ?

**Theorem.** PARTITION ist NP-vollständig.

**Beweis** (PARTITION  $\in$  NP).

Gegeben Zeuge  $S$ , prüfe  $S \subseteq A$  und berechne die Summe von  $S$  und  $A \setminus S$ .

**Beweis** (NP-schwer) durch **Reduktion von SUBSETSUM**.

### ACHTUNG!

PARTITION ist Spezialfall von SUBSETSUM bei dem  $b := \frac{1}{2} \sum_{i=1}^n a_i$ .

→ Das hilft **nicht** für den Beweis der NP-Härte! → Es **könnte** ja sein, dass dieser Spezialfall leichter ( $\in P$ ) ist!

## PARTITION, Fortsetzung

**Beweis** (NP-schwer) durch **Reduktion von SUBSETSUM**.

Gegeben eine SUBSETSUM-Instanz  $(A = \{a_1, a_2, \dots, a_n\}, b)$ .

O.b.d.A.:  $\max_i a_i < b$  und  $[A]/2 \neq b$ . Für  $R \subseteq A$ , sei  $[R] := \sum_{a \in R} a$ .

Wähle  $c := 2b$ ,  $d := [A]$ .  $\Rightarrow c \neq d$  and  $c, d \notin A$

PARTITION-Instanz:  $A' := \{a_1, a_2, \dots, a_n, c, d\}$ .

$\rightarrow [A'] = [A] + 2b + [A] = 2[A] + 2b$

$\rightarrow$  PARTITION-Lösung  $S'$  hätte  $[S'] = [A' \setminus S'] = [A']/2 = [A] + b$ .

**Behauptung:** PARTITION  $A'$  erfüllbar  $\Leftrightarrow$  SUBSETSUM  $(A, b)$  erfüllbar.

- ▶ Angenommen SUBSETSUM  $(A, b)$  erfüllbar durch  $S \subseteq A$ .  
 $\Rightarrow$  PARTITION  $A'$  erfüllbar durch  $S' := S \cup \{d\}$ , da  $[S'] = b + [A]$
- ▶ Angenommen PARTITION  $A'$  erfüllbar durch  $S'$ .

Da  $c + d > [A] + b$  landen  $c, d$  in unterschiedlichen Partitionen  $\rightarrow$  o.B.d.A.  $d \in S'$ .

$\Rightarrow$  SUBSETSUM  $(A, b)$  erfüllbar durch  $S := S' \setminus \{d\}$ ,

da  $[S] = [S'] - [A] = b$ .



# BINPACKING

**Gegeben:** Eine Menge von natürlichen Zahlen

$A = \{a_1, a_2, \dots, a_n\}$ , eine „Behältergröße“  $b \in \mathbb{N}$  und eine „Behälteranzahl“  $k \in \mathbb{N}$ .

**Gefragt:** Existiert Aufteilung von  $A$  in  $k$  disjunkte Teilmengen

$B_1, \dots, B_k$ , so dass  $|B_i| \leq b$  für alle  $1 \leq i \leq k$ ?

**Theorem.** BINPACKING ist NP-vollständig.

**Beweis** (BINPACKING  $\in$  NP).

Gegeben Zeugen  $B_1, \dots, B_k$ , prüfe  $\bigcup_i B_i = A$  und alle  $|B_i| \leq b$ .

**Beweis** (NP-schwer) durch **Reduktion von PARTITION**.

- ▶ PARTITION ist tatsächlich ein Spezialfall von BINPACKING!
- ▶ Sei  $A$  eine PARTITION-Instanz.  
 $\Rightarrow$  BINPACKING-Instanz  $(A, b = |A|/2, k = 2)$



## CLIQUE

**Gegeben:** Ein ungerichteter Graph  $G = (V, E)$  und ein  $k \in \mathbb{N}$ .  
**Gefragt:** Existiert eine Knoten-Teilmenge  $C \subseteq V$  mit  $|C| \geq k$ ,  
so dass  $\{v, w\} \in E$  für alle  $v, w \in C, v \neq w$ ?

**Theorem.** CLIQUE ist NP-vollständig.

**Beweis** (CLIQUE  $\in$  NP). Überprüfe Zeugen  $C$ . ✓

**Beweis** (NP-schwer) durch **Reduktion von 3SAT**.  
(Nächste Folie)

## CLIQUE, Fortsetzung

**Gegeben:** Ein ungerichteter Graph  $G = (V, E)$  und ein  $k \in \mathbb{N}$ .

**Gefragt:**  $\exists C \subseteq V$  mit  $|C| \geq k \wedge \forall v \neq w \in C : \{v, w\} \in E$ ?

**Beweis** (NP-schwer) durch **Reduktion von 3SAT**.

- ▶ 3SAT-Formel  $F$  mit  $m$  Klauseln  $(l_{i,1} \vee l_{i,2} \vee l_{i,3})$ ,  $1 \leq i \leq m$ .
- ▶ Erstelle CLIQUE-Instanz  $G = (V, E)$ ,  $k := m$  mit:

$$V := \{v_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq 3\},$$

$$E := \{\{v_{i,j}, v_{p,q}\} \mid i \neq p, l_{i,j} \neq \neg l_{p,q}\}$$

- 1 Knoten der selben Klausel sind nicht verbunden  
→ pro Klausel kann nur ein Knoten in  $C$  sein
- 2 Knoten von sich widerspr. Literalen sind nicht verbunden

**Behauptung:** 3SAT  $F$  erfüllbar  $\Leftrightarrow$  CLIQUE  $(G, k)$  erfüllbar

- ▶ Idee:  $v_{i,j} \in C \Leftrightarrow l_{i,j} = \text{true}$  (entspr. Klausel erfüllt).
- ▶ ( $\Leftarrow$ )  $\exists C$  mit  $|C| \geq k = m$ :  $C$  ist gültige (Teil)Belegung (1,2!)
- ▶ ( $\Rightarrow$ )  $F$  erfüllbar: Wähle pro Klausel eines der positiven Literale aus  $\rightarrow C$  mit  $|C| = m = k$ . □

## VERTEXCOVER

**Gegeben:** Ein ungerichteter Graph  $G = (V, E)$  und ein  $k \in \mathbb{N}$ .

**Gefragt:** Existiert eine Knoten-Teilmenge  $W \subseteq V$  mit  $|W| \leq k$ , so dass für jede Kante mindestens einer der beiden Endknoten in  $W$  liegt (d.h.  $\forall \{v, w\} \in E: v \in W \vee w \in W$ )?

**Theorem.** VERTEXCOVER ist NP-vollständig.

**Beweis** (VERTEXCOVER  $\in$  NP). Überprüfe Zeugen  $W$ . ✓

**Beweis** (NP-schwer) durch **Reduktion von CLIQUE**.

- ▶ Sei  $(G = (V, E), k)$  eine CLIQUE-Instanz.
- ▶ Erstelle VERTEXCOVER-Instanz  $(\bar{G} = (V, \bar{E}), \bar{k})$  mit  $\bar{E} := \{\{v, w\} \mid v, w \in V, v \neq w, \{v, w\} \notin E\}$  und  $\bar{k} := |V| - k$ . ( $\bar{G}$  ist der Komplementgraph von  $G$ .)
- ▶ **Behauptung:**  
 $\text{VERTEXCOVER } (\bar{G}, \bar{k})$  erfüllbar  $\Leftrightarrow$   $\text{CLIQUE } (G, k)$  erfüllbar.  
 (Beweis nächste Folie)

## VERTEXCOVER, Fortsetzung des Beweises

- ▶ Falls  $G$  eine Clique  $C$  der Größe  $k$  enthält, dann hat  $\bar{G}$  ein VC der Größe  $\bar{k} = |V| - k$ .

Wähle  $W := V \setminus C$ . Wir zeigen, dass  $W$  ein VC ist.

In  $G$  sind innerhalb von  $C$  alle Knoten verbunden.

⇒ In  $\bar{G}$  gibt es keine Kanten innerhalb von  $C$ .

⇒ In  $\bar{G}$  haben alle Kanten mindestens einen Endknoten  
aus  $V \setminus C = W$ .

⇒  $W$  ist ein VC.

- ▶ Falls  $\bar{G}$  ein VC  $W$  der Größe  $\bar{k} = |V| - k$  hat, dann hat  $G$  eine Clique der Größe  $k$ .

Wähle  $C := V \setminus W$ . Wie zeigen, dass  $C$  eine Clique ist.

In  $\bar{G}$  hat jede Kante mindestens einen Endknoten aus  $W$ .

⇒ In  $\bar{G}$  gibt es innerhalb von  $V \setminus W = C$  keine Kanten.

⇒ In  $G$  bildet  $C$  eine Clique. □

## GERICHTETER HAMILTONKREIS

**Gegeben:** Ein gerichteter Graph  $G = (V, A)$ .

**Gefragt:** Existiert eine Rundtour, die jeden Knoten genau einmal besucht?

**Theorem.** GERICHTETER HAMILTONKREIS ist NP-vollständig.

**Beweis** (GERICHTETER HAMILTONKREIS  $\in$  NP). ✓

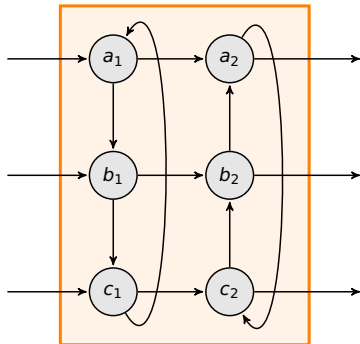
**Beweis** (NP-schwer) durch **Reduktion von 3SAT**.

- ▶ Sei  $F$  eine 3SAT-Formel mit Klauseln  $K_1, \dots, K_m$  und Variablen  $x_1, \dots, x_n$ .
- ▶ **Gadget-Beweis:**  
Erstelle einen Graph  $G$  aus mehreren Bestandteilen...  
(siehe nächste Folie)

# GERICHTETER HAMILTONKREIS, Klauselgadget

**Hilfskonstruktion:** Betrachte folgenden Teilgraphen  $H$ .

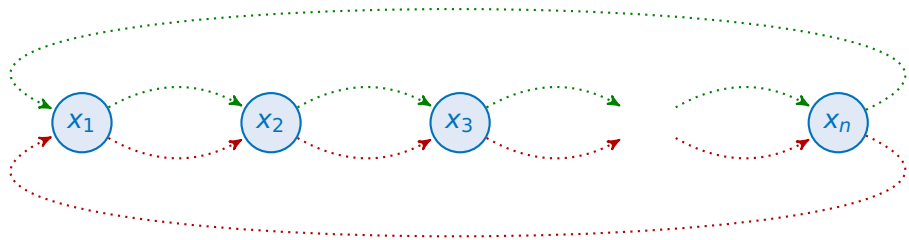
## Beobachtungen.



- ▶ Wenn ein Hamiltonkreis in  $a_1$  ( $b_1, c_1$ ) eintritt, **muss** er  $H$  über  $a_2$  ( $b_2, c_2$ ) verlassen.
- ▶ Ein Hamiltonkreis läuft 1–3 mal durch  $H$ . Bei jedem Durchlauf „sammelt“ er entsprechend 1–3 vollständige Knotenpaare ( $a_1, a_2$ ;  $b_1, b_2$ ;  $c_1, c_2$ ) auf.

⇒ „**Klauselgadget**“: Mind. ein Eingang muss angelaufen werden (und das Gadget wird konsistent verlassen)

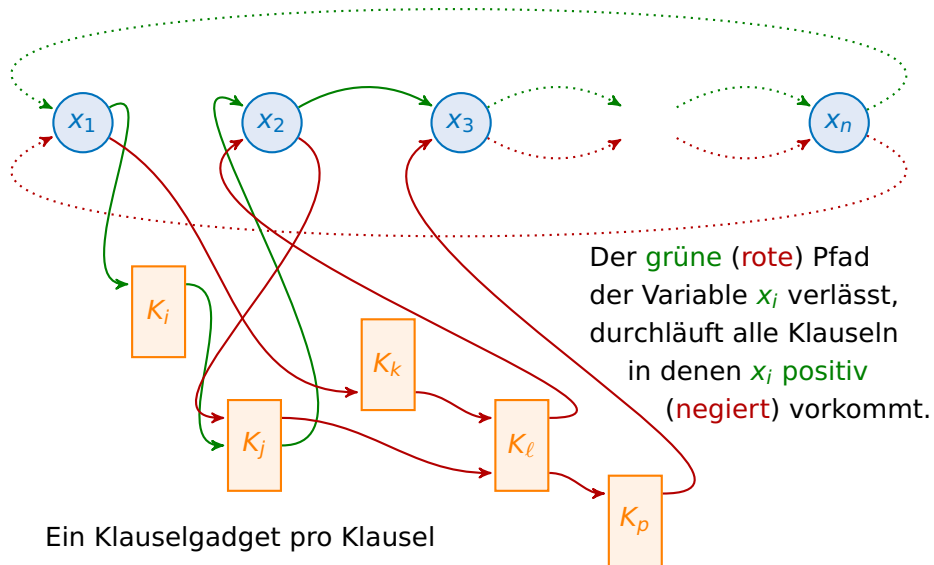
# GERICHTETER HAMILTONKREIS, Konstruktion



**Idee.** Baue einen Graph der genau dann einen Hamiltonkreis erlaubt, wenn die Formel erfüllbar ist.

- ▶ Ein Knoten für jede Variable. Rundtour soll von  $x_1$  nach  $x_2$ , nach  $x_3$ , etc. bis nach  $x_n$  und dann zurück nach  $x_1$ .
- ▶ Von jedem Variablen-Knoten gibt es einen „grünen“ und einen „roten“ Weg zum nächsten Variablen-Knoten. Wird der grüne ausgehende Weg gewählt  $\Rightarrow$  die Variable ist *true*, sonst *false*.

# GERICHTETER HAMILTONKREIS, Konstruktion



## GERICHTETER HAMILTONKREIS, Beweis

**Behauptung:** 3SAT  $F$  erfüllbar  $\Leftrightarrow$  HAMILTONKREIS  $G$  erfüllbar

- ▶ Falls  $F$  erfüllbar ist, enthält  $G$  einen Hamiltonkreis.

Betrachte eine erfüllende Belegung  $x'$  für  $F$ , und wähle die roten/grünen Pfade in  $G$  entsprechend. Da  $x'$  erfüllend ist, ist jede Klausel erfüllt. Für jedes Klauselgadget ist also mindestens ein Pfad (max. 3) gewählt, der durch es läuft. Entsprechend können alle Knoten des Gadgets genau 1x angelaufen werden.

- ▶ Falls  $G$  einen Hamiltonkreis enthält, ist  $F$  erfüllbar.

Betrachte einen Hamiltonkreis. Durch die Eigenschaften des Klauselgadgets, wissen wir, dass ein in  $a_1$  eingehender Pfad das Gadget über  $a_2$  verlassen **muss** (analog für  $b, c$ ). Daher können wir eindeutige Pfade von  $x_i$  nach  $x_{(i+1) \bmod n}$  extrahieren und als Variablenbelegung interpretieren. Da jeder Knoten besucht wird, wird jede Klausel passiert. Die extrahierte Variablenbelegung erfüllt also alle Klauseln. Die Belegung ist eindeutig, da jeder Variablenknoten nur entlang der grünen **oder** roten Ausgangskante verlassen werden kann. □

# HAMILTONKREIS

**Gegeben:** Ein **ungerichteter** Graph  $G = (V, E)$ .

**Gefragt:** Existiert eine Rundtour, die jeden Knoten genau einmal besucht?

**Theorem.** HAMILTONKREIS ist NP-vollständig.

**Beweis.** **1**  $\in NP$  ✓; **2** NP-schwer durch **Reduktion von GERICHTETERHAMILTONKREIS.**

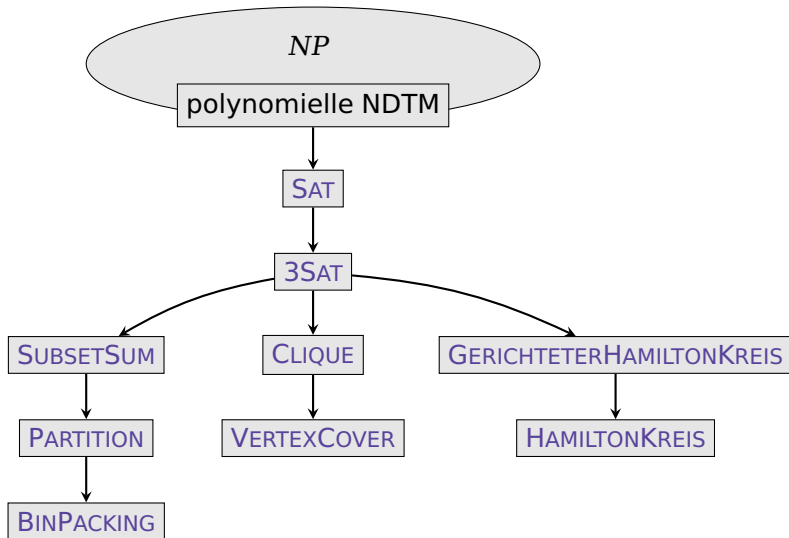
- ▶ Sei  $G'$  ein gerichteter Graph für GERICHTETERHAMILTONKREIS.
- ▶ Ersetze jeden Knoten (mit gerichteten Kanten) durch drei Knoten mit ungerichteten Kanten:



Um den mittleren Knoten zu erreichen, muss man nach einer blauen Kante eine violette Kante entlanggehen.

$\Rightarrow G'$  ist *true*-Instanz g.d.w.  $G$  ist *true*-Instanz. □

# Baum unserer NP-Vollständigkeitsbeweise



# Komplexitätstheorie

Starke & Schwache NP-Vollständigkeit

## Starke & Schwache NP-Vollständigkeit

- ▶ Komplexität wird stets relativ zu  $n$ , der Länge der **kodierten Eingabe**, betrachtet.
- ▶ Sei  $a$  eine (binär kodierte) Zahl in der Eingabe.  
→  $a$  benötigt mindestens  $\log_2(a)$  Platz.
- ▶ Bei Eingabelänge  $n$  kann der **Wert** einer Zahl exponentiell groß sein ( $2^{\mathcal{O}(n)}$ )!

**Definition.** Sei  $\mathcal{X}$  ein NP-vollständiges Problem, und  $\mathcal{X}_p$  das selbe Problem eingeschränkt auf Instanzen bei denen der Wert aller Zahlen durch ein Polynom von  $n$  beschränkt ist.

$\mathcal{X}$  ist **stark** NP-vollständig, falls sogar  $\mathcal{X}_p$  NP-vollständig ist, und **schwach** NP-vollständig sonst.

## Stark vs. Schwach, Pseudopolynomiell

- ▶ **Stark** *NP*-vollständige Probleme bleiben also selbst bei „kleinen“ Eingabezahlen schwer.
- ▶ **Schwach** *NP*-vollständige Probleme werden nur durch sehr große Zahlen schwer.
- ▶ *NP*-vollständige Probleme bei denen keine Zahlen gegeben sind (z.B. SAT, 3SAT) sind **stark** *NP*-vollständig.

**Definition.** Sei  $a$  der größte Zahlenwert in der Eingabe. Ein **pseudopolynomieller Algorithmus** hat eine Laufzeit polynomiell in  $n$  und  $a$ , d.h.  $\mathcal{O}(n^c \cdot a^d)$  für Konstanten  $c, d \geq 0$ .

**Beobachtung.** Schwach *NP*-vollständige Probleme erlauben pseudopolynomielle Algorithmen (i.d.R. basierend auf **Dynamischer Programmierung**), stark *NP*-vollständige Probleme nicht (außer  $P = NP$ ).

# SUBSETSUM: Pseudopolynomiell

BINPACKING ist **stark** NP-vollständig (ohne Beweis).

SUBSETSUM ist nur **schwach** NP-vollständig

→ es erlaubt einen Algorithmus polynomiell in  $n$  und  $b$ .

## Pseudopolynomieller Algorithmus für SUBSETSUM.

**Gegeben.**  $A = \{a_1, \dots, a_m\}$ ,  $b \in \mathbb{N}$ .

**Gefragt.** Existiert  $S \subseteq A$  mit  $[S] = b$ ?

**Idee einer Dynamischen Programmierung:** Berechne alle möglicherweise interessanten Teillösungen, speichere diese in einem Array, und kombiniere aus kleinen Teillösungen immer größere, bis eine volle Lösung erreicht ist.

Sei  $Q$  ein Bool-Array mit Einträgen für  $0 \leq i \leq m$  und  $0 \leq s \leq b$ .  
Wir möchten:

$Q[i, s] = \text{true} \iff \exists$  Teilmenge von  $\{a_1, \dots, a_i\}$  mit Summe  $s$ .

## SUBSETSUM: Pseudopolynomiell

$Q[i, s] = true \iff \exists$  Teilmenge von  $\{a_1, \dots, a_i\}$  mit Summe  $s$ .

// Initialisierung

$\forall 0 \leq i \leq m : Q[i, 0] := true, \quad \forall 0 < s \leq b : Q[0, s] := false$

// Dynamische Programmierung

**for**  $s = 1, 2, \dots, b$ :

**for**  $i = 1, 2, \dots, m$ :

$Q[i, s] := Q[i - 1, s] \vee (s - a_i \geq 0 \wedge Q[i - 1, s - a_i])$  (\*)

**return**  $Q[m, b]$  // Ergebnis

(\*) Damit  $Q[i, s] = true$  muss:  $\exists S' \subseteq \{a_1, \dots, a_i\}$  mit  $[S'] = s$ .

**Fall** „ $a_i \notin S'$ “:

$S' \subseteq \{a_1, \dots, a_{i-1}\} \iff Q[i - 1, s] = true$

**Fall** „ $a_i \in S'$ “:

$[S'] = s \iff [S' \setminus \{a_i\}] = s - a_i \iff Q[i - 1, s - a_i] = true$

## SUBSETSUM: Pseudopolynomiell

$Q[i, s] = \text{true} \iff \exists$  Teilmenge von  $\{a_1, \dots, a_i\}$  mit Summe  $s$ .

// Initialisierung

$\forall 0 \leq i \leq m : Q[i, 0] := \text{true}, \quad \forall 0 < s \leq b : Q[0, s] := \text{false}$

// Dynamische Programmierung

**for**  $s = 1, 2, \dots, b$ :

**for**  $i = 1, 2, \dots, m$ :

$Q[i, s] := Q[i - 1, s] \vee (s - a_i \geq 0 \wedge Q[i - 1, s - a_i])$

**return**  $Q[m, b]$  // Ergebnis

**Laufzeit:**  $\mathcal{O}(b \cdot m) \subseteq \mathcal{O}(b \cdot n)$  ( $n$  = Eingabelänge in Bits)

Dies ist i.A. **nicht** polynomiell, da  $b = 2^{\Theta(n)}$  sein kann!

→ Algorithmus ist nur **pseudopolynomiell**.

**ABER:** Falls  $b = \mathcal{O}(\text{poly}(n))$  → Gesamtlaufzeit polynomiell.

⇒ SUBSETSUM ist nur **schwach** NP-vollständig.

# **Komplexitätstheorie**

## Fixed Parameter Tractability

# Fixed Parameter Tractability

(deutsch: Fest-Parameter-Handhabbarkeit)

## Fixed Parameter Tractable (FPT).

Ein Entscheidungsproblem ist **in FPT in Bezug auf Parameter  $k$** , wenn es sich in  $\mathcal{O}(f(k) \cdot \text{poly}(n))$  Zeit lösen lässt. Dabei ist

- ▶  $\text{poly}(n)$  ein Polynom in der Eingabegröße, und
- ▶  $f(k)$  eine beliebige berechenbare Funktion in  $k$  (z.B. exponentiell).

**Bedeutung:** Das Problem ist polynomiell lösbar (**tractable**), wenn man  $k$  (**Parameter**) als konstant (**fixed**) annimmt.

**Sinnhaftigkeit:** Instanz soll auch bei beschränktem Parameter immer noch beliebig groß werden können!

# Laufzeiten

## Beispiele für (FPT?) Laufzeiten.

$\mathcal{O}(k^n \cdot n^2)$  **nicht** FPT

$\mathcal{O}(3^k \cdot n^5)$  FPT

$\mathcal{O}(2^{3^{k!}} \cdot n^7)$  FPT

$\mathcal{O}(2^k \cdot n^k)$  polynomielle Laufzeit falls  $k$  konstant,  
aber **nicht** in FPT

## Schwach NP-vollständig $\rightarrow$ FPT in Bitlänge

**Theorem.** Sei  $\mathcal{X}$  ein beliebiges **schwach NP-vollständiges** Problem, und  $k$  die Anzahl der Bits der Binärkodierung der größten vorkommenden Zahl.

Das Problem  $\mathcal{X}$  ist **FPT in Bezug auf die Bitlänge  $k$** .

**Beweis.** Schwach NP-vollständige Probleme erlauben einen Algorithmus mit Laufzeit  $\mathcal{O}(\text{poly}(n, 2^k)) = \mathcal{O}(f(k) \cdot \text{poly}(n))$ .  $\square$

**Beispiel:** Entscheidungsproblem SUBSETSUM. Sei  $k$  die Anzahl der Bits zur Kodierung von  $b \in \mathbb{N}$ .

Da  $b < 2^k$ , können wir die dynamische Programmierung mit Laufzeit  $\mathcal{O}(b \cdot n) \subseteq \mathcal{O}(2^k \cdot n)$  anwenden.

# Parameter

Viele verschiedene Parametrisierungen (= Maße für Eigenschaften der Instanzen) möglich:

- ▶ Bitlänge der größten Zahl,
- ▶ Maximalgrad des Eingabegraphs,
- ▶ „Baumähnlichkeit“ des Eingabegraphs,
- ▶ ... und viele mehr...

und besonders beliebt:

- ▶ **Zielfunktion des zugehörig. Optimierungsproblems**

## **zum Beispiel:**

Existiert ein Vertex Cover der Größe maximal  $k$ ?

Existiert eine Clique mit mindestens  $k$  Knoten? ...

Ist das nicht das zum Optimierungsproblem zugehörige Entscheidungsproblem?

## FPT/Zielfunktion vs. Entscheidungsproblem

Gegeben ein  $NP$ -schweres Minimierungsproblem  $\mathcal{X}$ .

### Zugehöriges Entscheidungsproblem:

Gegeben: Instanz  $\mathcal{I}$  für  $\mathcal{X}$  und  $k \in \mathbb{N}$ .

Gefragt: Gibt es eine Lösung für  $\mathcal{I}$  mit Zielfunktion  $\leq k$ ?

### Entscheidungsproblem mit Zielfunktionsparameter $k$ :

Gegeben: Instanz  $\mathcal{I}$  für  $\mathcal{X}$

Gefragt: Gibt es eine Lösung für  $\mathcal{I}$  mit Zielfunktion  $\leq k$ ?

**Unterschied:** Im  $FPT$ -Setting ist  $k$  **nicht Teil der Eingabe**, sondern eine vorab festgelegte Konstante! Laufzeit wird in Abhängigkeit der Eingabegröße **und**  $k$  angegeben.

- ▶ **VERTEXCOVER** ist z.B. in  $FPT$  in Bezug auf die Zielfunktion.
- ▶ **CLIQUE** ist z.B. **nicht**  $FPT$  in Bezug auf die Zielfunktion.  
(Das Problem ist „ $W[1]$ -schwer“  $\approx$  „ $NP$ -vollständig im  $FPT$  Sinn“; polynomieller Algorithmus unwahrscheinlich)

# VERTEXCOVER: *FPT* in Bezug auf die Zielfunktion

VERTEXCOVER (Optimierungsproblem)

**Gegeben:** Ungerichteter Graph  $G = (V, E)$ .

**Finde:** Kleinstes  $k := |W|$  mit  $W \subseteq V$  und  $\forall e \in E: e \cap W \neq \emptyset$ ?

## Theorem.

VERTEXCOVER ist in *FPT* in Bezug auf die Zielfunktion.

**Beweis.** Wir zeigen dies nun **zweimal**, jeweils mit unterschiedlichen Techniken, die oft hilfreich bei *FPT*-Algorithmen sind:

- 1 Kernelization
- 2 Tiefenbeschränkter Suchbaum

## VC: Kernelization, 1/2

**Kernelization.** Schrumpfe Problem Instanz  $\mathcal{I}$  in polynomieller Zeit auf seinen schwierigen **Kern (Kernel)**  $\mathcal{K}$ , so dass die Größe von  $\mathcal{K}$  nur von  $k$  abhängt (oder zeige, dass es sich um eine Ja- bzw. Nein-Instanz handelt).

→ Löse das Problem auf  $\mathcal{K}$  durch exponentielle Enumeration.

Um ein Vertex Cover in einem Graphen  $G = (V, E)$  mit maximal  $k$  Knoten zu finden...

- ▶ Knoten  $V_F \subseteq V$  mit Grad 0 können **ignoriert** werden.
- ▶ Knoten  $V_T \subseteq V$  mit Grad  $> k$  **müssen gewählt** werden. (Sonst müsste man alle Nachbarn wählen  $\rightarrow |VC| > k$ ).
- ▶ Ein Graph mit Maximalgrad  $\Delta$  und Vertex Cover der Größe  $c$  enthält maximal  $\Delta \cdot c$  Kanten. (Jeder Knoten im VC kann max.  $\Delta$  Kanten abdecken.)

## VC: Kernelization, 2/2

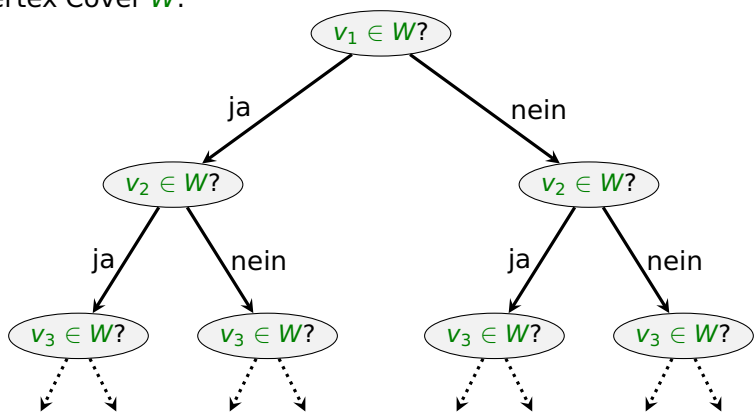
### FPT-Algorithmus von [Buss, Goldsmith 1993]

- ▶ Sei  $V_T := \{v \in V(G) : \deg(v) > k\}$ .
- ▶ Starte mit  $H \leftarrow G$ . Lösche alle Knoten  $V_T$  aus  $H$ . Lösche alle Knoten  $v$  mit  $\deg(v) = 0$  aus  $H$ .
- ▶ Neues Ziel: Finde Vertex Cover der Größe  $k' := k - |V_T|$  in  $H$  (Kernel)
- ▶ **Falls**  $E(H) > k' \cdot k$ : VC existiert nicht.
- ▶ **Sonst**: Enumeriere alle Lösungsmöglichkeiten in  $H$ !

**Folgerung:** In der letzten Zeile ist die Größe von  $H$  nur  $\mathcal{O}(k^2)$ . Enumerieren benötigt also nur  $\mathcal{O}(f(k))$  Zeit, für eine berechenbare Funktion  $f$  abhängig nur von  $k$ . Alle anderen Schritte benötigen nur  $\mathcal{O}(kn)$  Zeit.  $\Rightarrow$   
 $\mathcal{O}(kn + f(k)) \subseteq \mathcal{O}(f(k) \cdot n)$  Gesamtzeit  $\Rightarrow$  FPT.

## VC: Tiefenbeschränkter Suchbaum, 1/2

Enumeriere in einem Suchbaum alle Möglichkeiten für ein Vertex Cover  $W$ .



**Problem.** Tiefe des Baums:  $|V| \Rightarrow \mathcal{O}(2^{|V|})$  Suchbaum-Knoten  
 $\Rightarrow$  Wir möchten gerne eine maximale Tiefe  $k$ !

## VC: Tiefenbeschränkter Suchbaum, 2/2

### So klappts...

- ▶ Vergrößere  $W$  bei **jedem** Suchbaumabstieg
  - Lösung mit  $\leq k$  Knoten innerhalb der Suchbaumtiefe  $k$
  - $\mathcal{O}(2^k)$  Suchbaumknoten
- ▶ Für jede Kante muss (mind.) einer der beiden Endknoten ins VC → dies ist die Entscheidung pro Suchbaumknoten

### (Einfacher) Suchbaumalgorithmus

- ▶ Starte Suchbaum mit  $W = \emptyset$
- ▶ Wähle beliebige Kante  $(uv)$  im Graph  $G - W$  (= bisher gewählte Knoten inkl. inzidenten Kanten löschen)
- ▶ Erstelle zwei neue Suchbaumknoten mit  $W := W \cup \{u\}$  bzw.  $W := W \cup \{v\}$
- ▶ Es existiert eine Lösung **genau dann wenn** innerhalb der ersten  $k$  Suchebenen ein  $G - W$  keine Kanten hat

# VERTEXCOVER: *FPT* in Bezug auf die Zielfunktion

## Theorem.

VERTEXCOVER ist in *FPT* in Bezug auf die Zielfunktion.

Um dies zu beweisen, haben wir zwei Möglichkeiten gesehen.

**Genauere Laufzeiten:** (ohne Beweis)

- |   |                             |                          |
|---|-----------------------------|--------------------------|
| 1 | Kernelization               | $O(k V  + 2^k k^{2k+2})$ |
| 2 | Tiefenbeschränkter Suchbaum | $O(2^k  V )^2$           |
|   |                             | $O(k V  + 1,286^k)^3$    |

**Beobachtung:** Tiefenbeschränkter Suchbaum ist sogar **echt polynomiell** wenn  $k = O(\log n)$ .

---

<sup>2</sup> wie auf den Folien beschrieben

<sup>3</sup> schlaue Verzweigungsstrategien [Chen, Kanj, Jia 2010];

Praxis:  $k \leq 400$ ,  $n$  beliebig

# Komplexitätstheorie

Co-NP

## Co-NP

Wir erinnern uns:

- ▶  $P$  = Entscheidungsprobleme, die wir deterministisch in polynomieller Zeit entscheiden können.
- ▶  $NP$  = Entscheidungsprobleme, bei denen wir deterministisch in polynomieller Zeit einen Zeugen für eine **ja**-Instanz überprüfen können.

### Äquivalente Definitionen.

$Co-NP$  = Entscheidungsprobleme, bei denen wir deterministisch in polynomieller Zeit einen Zeugen für eine **nein**-Instanz überprüfen können.

$Co-NP$  =  $\exists$  nicht-determ. TM die akzeptiert gdw. **alle** möglichen Rechenwege in polynomieller Zeit akzeptieren.

$Co-NP$  = Entscheidungsprobl., deren Komplement in  $NP$  liegt.

## Co-NP, Beispiel

Betrachte:

### SUBSETSUM

**Gegeben:** Menge  $A = \{a_1, a_2, \dots, a_n\} \subset \mathbb{N}$ ,  $b \in \mathbb{N}$ .

**Gefragt:** Existiert eine Teilmenge  $S \subseteq A$  mit  $\sum_{a \in S} a = b$ ?

Dann ist das Komplement von SUBSETSUM:

### Co-SUBSETSUM

**Gegeben:** Menge  $A = \{a_1, a_2, \dots, a_n\} \subset \mathbb{N}$ ,  $b \in \mathbb{N}$ .

**Gefragt:** Gilt für alle Teilmengen  $S \subseteq A$ , dass  $\sum_{a \in S} a \neq b$ ?

Da SUBSETSUM  $\in$  NP, ist Co-SUBSETSUM  $\in$  Co-NP.

## Co-NP-Vollständigkeit

Man kann *Co-NP*-Schwere und *Co-NP*-Vollständigkeit („die schwersten Probleme in *Co-NP*“) analog zu *NP* definieren:

**Definition.**  $\mathcal{X}$  ist **Co-NP-schwer** genau dann wenn:

- ▶ Jedes Problem aus *Co-NP* lässt sich deterministisch und in polynomieller Zeit auf  $\mathcal{X}$  reduzieren

**Definition.**  $\mathcal{X}$  ist **Co-NP-vollständig** genau dann wenn:

- 1  $\mathcal{X} \in \text{Co-NP}$ , und
- 2  $\mathcal{X}$  ist *Co-NP*-schwer.

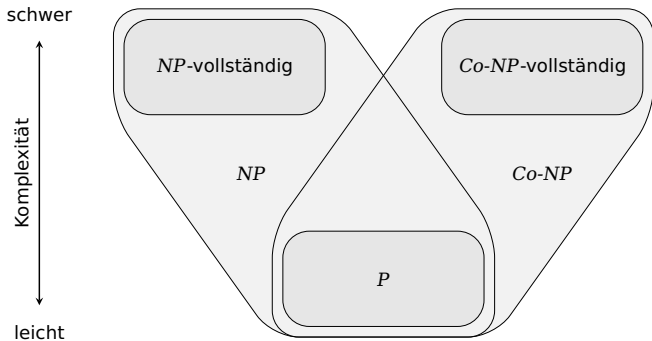
Beispiel:

### TAUTOLOGIE

**Gegeben.** Aussagenlogische Formel  $F$ .

**Gefragt.** Ist  $F$  immer erfüllt?

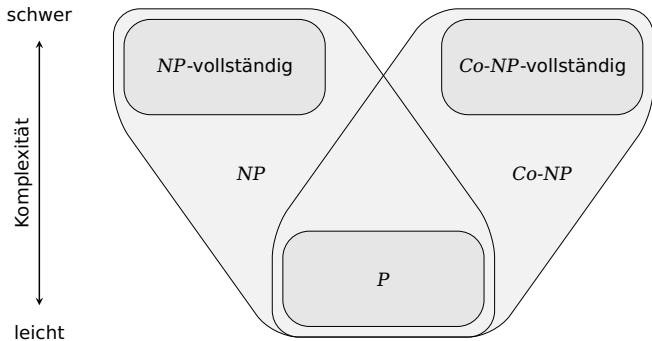
# Co-NP



## Wichtige Eigenschaften.

- ▶  $P \subseteq NP \cap Co-NP$ .
- ▶ Sei  $\mathcal{X}$  NP-vollständig und  $\bar{\mathcal{X}}$  das zugehörige Co-Problem (z.B.  $\mathcal{X} = \text{SUBSETSUM}$ ,  $\bar{\mathcal{X}} = \text{Co-SUBSETSUM}$ ). Dann ist  $\bar{\mathcal{X}}$  Co-NP-vollständig.

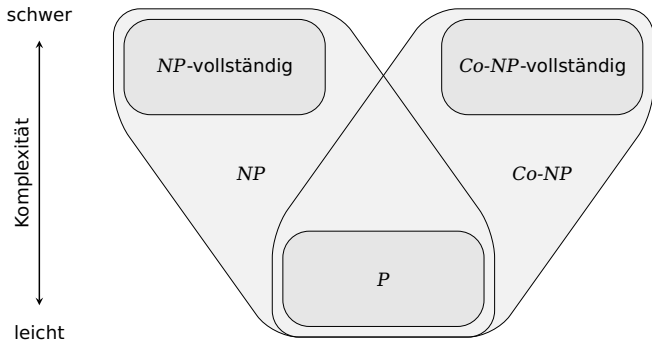
# Co-NP



**Vermutungen.** (Ähnlich zur Vermutung, dass  $P \neq NP$ )

- ▶  $NP \neq Co-NP$ .
- ▶  $P \neq Co-NP$ .
- ▶ Man weiß, dass **Faktorisieren** in  $NP \cap Co-NP$  liegt, aber man weiß nicht ob in  $P$ . (Vielleicht liegt es in  $NPI$ ).

# Co-NP



## Wissenswertes.

- ▶ Falls  $P = NP$ :  $NP = Co-NP = P$
- ▶ Falls  $Co-NP = NP$ : Frage  $P \stackrel{?}{=} NP$  **bleibt**
- ▶ Sei  $\mathcal{X}$  NP-vollständig:  $\mathcal{X} \in Co-NP \implies NP = Co-NP$ .
- ▶ Sei  $\mathcal{X}$  Co-NP-vollständig:  $\mathcal{X} \in NP \implies NP = Co-NP$ .

# Primzahlproblem: PRIMES

## PRIMES

**Gegeben.** Gegeben eine Zahl  $p$ .

**Gefragt.** Ist  $p$  eine Primzahl?

- ▶ Eingabegröße:  $n = \lceil \log_2 p \rceil = \Theta(\log p)$ .
- ▶ **Naiver Algorithmus:** Teste jede Zahl  $d = 2, \dots, \sqrt{p}$  ob  $d|p$ .  
**Laufzeit:**  $\mathcal{O}(\sqrt{p}) = \mathcal{O}(2^{n/2})$  Teilbarkeitstests  $\Rightarrow$  exponentiell!
- ▶ **PRIMES liegt in Co-NP:**  
**Zeuge:** Ein Teiler  $d$  von  $p$  beweist, dass  $p$  keine Primzahl ist. Da  $d < p$  ist dieser Zeuge nur  $\lceil \log_2 d \rceil = \mathcal{O}(n)$  groß.  
**Überprüfen** des Zeugen benötigt nur  $\text{poly}(n)$  Zeit (Division durchführen und auf Rest=0 testen).
- ▶ **Komplizierter:** PRIMES liegt auch in NP. ( $\exists$  positive Zeugen!)

# Faktorisierung: FACTORIZATION

## FACTORIZATION

**Gegeben.** Gegeben eine Zahl  $p$ .

**Gesucht.** Finde einen Teiler von  $p$ , oder schlussfolgere, dass  $p$  prim ist.

**Kein** Entscheidungsproblem  $\Rightarrow$  **Function Problem!**

Komplexitätsklassen **FP** und **FNP** („function (non-deterministic) polynomial time“) analog zu  $P/NP$ :

- ▶  $FP \approx \exists$  determ. TM die das Problem in polynom. Zeit löst.
- ▶  $FNP \approx$  man kann eine Lösung mittels einer deterministischen TM in polynomieller Zeit überprüfen.

**FACTORIZATION** liegt in  $FNP$ , aber es ist **unbekannt** ob in  $FP$ .

## FACTORIZATION (Entscheidungsproblem)

**FACTORIZATION** (als Funktionsproblem)

**Gegeben.** Gegeben eine Zahl  $p$ .

**Gesucht.** Finde einen Teiler von  $p$ , oder schlussfolgere, dass  $p$  prim ist.

**FACTORIZATION** (als Entscheidungsproblem)

**Gegeben.** Gegeben eine Zahl  $p$  und eine Zahl  $m \leq p$ .

**Gefragt.** Gibt es ein  $d < m$  so dass  $d|p$ ?

- ▶ Mittels binärer Suche kann man dann das Funktionsprobl. lösen. ( $\leq \log_2 2^n \in \mathcal{O}(n)$  viele Suchschritte)
- ▶ **FACTORIZATION (als EP) ist in NP:**  
Zeuge  $d$  mit Größe  $\mathcal{O}(n)$  in  $\mathcal{O}(\text{poly}(n))$  prüfbar (Teilbarkeitstest)
- ▶ **FACTORIZATION (als EP) ist in Co-NP**  
→ später genauer

## PRIMES vs. FACTORIZATION: Gegeben $p$ ...

PRIMES. Ist  $p$  prim?

FACTOR./FP. Finde Teiler von  $p$  falls  $\exists$ .

Co-PRIMES.  $\exists$  Teiler von  $p$ ?

FACTOR./EP.  $\exists$  Teiler  $< m$  von  $p$ ?

- ▶ **Teiler-Zeuge:** PRIMES  $\in$  Co-NP,  
Co-PRIMES & FACTOR./EP  $\in$  NP, FACTOR./FP  $\in$  FNP
- ▶  $\exists$  „**Primalitätszeuge**“  $\rightarrow$  PRIMES  $\in$  NP,  
Co-PRIMES  $\in$  Co-NP, FACTOR./FP  $\in$  Co-FNP
- ▶ **FACTOR./EP**  $\in$  **Co-NP**, denn  $\exists$  Nein-Zeuge „**Primfaktoren von  $p$  (alle  $\geq m$ )**, inkl. ihrer **Primalitätszeugen**“; Faktoren als prim validieren, und dann ausmultiplizieren.

Durchbruch „**AKS**“ (Agrawal, Kayal, Saxena, 2002)

**PRIMES** (und damit auch Co-PRIMES) **liegt sogar in P!**

$\Rightarrow$  Algorithmus der PRIMES determ. in poly-Zeit beantwortet.

AKS liefert **keinen Teiler-Zeugen** bei Nicht-Primalität!

$\Rightarrow$  Es ist **offen** ob FACTOR./EP  $\in$  P (bzw. FACTOR./FP  $\in$  FP)

Mailüfterl

DEZIMALER VOLLTRANSISTOR - RECHENAUTOMAT

**Heinz Zemanek**

österreichischer Computerpionier

1. Januar 1920 – 16. Juli 2014

Erster Transistorrechner  
in Kontinentaleuropa

1955–57: gebaut

1958: positiver Primzahltest  
für **5 073 548 261** in 66 min

# **Komplexitätstheorie**

Probabilistische Komplexitätsklassen

## Warum soll Zufall helfen?

- ▶ Nicht in Fallen geraten, in die der Determinismus gerät
- ▶ Oft algorithmisch **einfacher** oder **effizienter** oder beides
- ▶ Ganzen Suchraum abzusuchen ist teuer, aber oft gar nicht so gründlich notwendig?

**1948**: Nicholas **Metropolis** & Stanisław **Ulam**

Entwicklung eines **Monte-Carlo Algorithmus**

(Berechnungen zur Kernwaffenentwicklung)

Rechner damals langsam und wenig Speicher!

Algorithmus nicht 100%ig exakt, aber schnell&einfach!

- ▶ Kryptographie: **Trapdoor function**  
Umkehrung/Invertierung soll **tatsächlich** schwer sein:
  - nicht nur im Worst-Case,
  - sondern auch, wenn man randomisierte Alg. benutzt

## Stochastik für den Hausgebrauch, 1/2

- ▶  $X$  = Zufallsvariable = wird auf Basis eines Zufalls einen Wert annehmen

Beispiel:  $X$  kann die Werte 1, 2, 4, 8 annehmen.

- ▶ **Wahrscheinlichkeit** (W'keit, **Probability**) ist ein Wert zw. 0 (=nie) und 1 (=immer). Ggf. auch in Prozent 0%–100%.
- ▶  $\mathbb{P}[\text{Ereignis A}]$  = W'keit, dass **Ereignis A** eintritt.

$\mathbb{P}[X > 5]$  = wie wahrscheinlich ist es, dass  $X$  einen Wert größer 5 annehmen wird?

- ▶ **Summe von W'keiten** wenn sich Ereign. gegens. ausschliessen

$$\mathbb{P}[X \geq 4] = \mathbb{P}[X=4] + \mathbb{P}[X=8]$$

- ▶ **Gegenwahrscheinlichkeit:**

$$\mathbb{P}[\text{nicht Ereignis A}] = 1 - \mathbb{P}[\text{Ereignis A}]$$

## Stochastik für den Hausgebrauch, 2/2

- W'keit mehrerer **unabhängiger Versuche** (z.B. Algor.aufrufe):

$$\begin{aligned} \mathbb{P}[\text{Bei 2 Versuchen jeweils } X=2] &= \mathbb{P}[X=2] \cdot \mathbb{P}[X=2] \\ \mathbb{P}[\text{Versuch 1: } X=2 \text{ und Versuch 2: } X=4] &= \mathbb{P}[X=2] \cdot \mathbb{P}[X=4] \\ \mathbb{P}[\text{Bei 2 Versuchen 1x } X=2 \text{ \& } 1x X=4] &= 2 \cdot \mathbb{P}[X=2] \cdot \mathbb{P}[X=4] \\ &= \mathbb{P}[X=2] \cdot \mathbb{P}[X=4] + \mathbb{P}[X=4] \cdot \mathbb{P}[X=2] \end{aligned}$$

$$\begin{aligned} \mathbb{P}[\text{Bei } k \text{ Versuchen jeweils } X=2] &= \mathbb{P}[X=2]^k \\ \mathbb{P}[\text{Bei } k \text{ Versuchen nie } X=2] &= \mathbb{P}[X \neq 2]^k = (1 - \mathbb{P}[X=2])^k \\ \mathbb{P}[\text{Bei } k \text{ Versuchen mind. 1x } X=2] &= 1 - (1 - \mathbb{P}[X=2])^k \\ \mathbb{P}[\text{Bei } k \text{ Versuchen genau 1x } X=2] &= k \cdot \mathbb{P}[X=2] \cdot (1 - \mathbb{P}[X=2])^{k-1} \end{aligned}$$

- $\mathbb{E}[X]$  = Erwartungswert (**Expectation**) von  $X$   
 $\approx$  mit W'keit gewichteter Mittelwert der möglichen Werte von  $X$   
 $\approx$  „Was kommt wahrscheinlich raus?“ (hm...)

$$\mathbb{E}[X] = 1 \cdot \mathbb{P}[X=1] + 2 \cdot \mathbb{P}[X=2] + 4 \cdot \mathbb{P}[X=4] + 8 \cdot \mathbb{P}[X=8]$$

# Art der Randomisierung: Las Vegas

## Beispiel: Sortieren

### ▶ BogoSort:

1 Wähle eine zufällige Reihenfolge der Elemente

2 Falls die Reihenfolge nicht sortiert ist: goto 1

▶ Ergebnis **korrekt**.

▶ **Erwartete** Laufzeit:  $\mathcal{O}(n!)$ , statt Worst-Case  $\infty$

### ▶ Randomisierter QuickSort:

▶ Ergebnis **korrekt**.

▶ **Erwartete** Laufzeit  $\mathcal{O}(n \log n)$ , statt Worst-Case  $\mathcal{O}(n^2)$

▶ **Erwartete** Laufzeit gilt **immer**, auch bei „schlechten“ Instanzen (z.B. vorsortiert)

**Immer** korrekt, **wahrscheinlich** schnell  $\Rightarrow$  **Las Vegas**

## Durchschnitt $\neq$ Erwartungswert

### Durchschnittliche Laufzeit: ( $\emptyset$ LZ)

Deterministischer polynomieller Algorithmus

Durchschnitt über alle Instanzen

= erwartete Laufzeit bei **zufällig gewählter** Instanz

### Erwartete Laufzeit: ( $\mathbb{E}$ LZ)

Randomisierter Algorithmus

„Durchschnitt über Runs für beliebige feste Instanz“

= (obere Schranke für) erwartete Laufzeit für **jede** Instanz

### Beispiel: QuickSort

**Pivot deterministisch:** Worst-Case  $\mathcal{O}(n^2)$ ,  $\emptyset$ LZ  $\mathcal{O}(n \log n)$

Bei vorsortierter Folge **immer**  $\mathcal{O}(n^2)$

**Pivot zufällig:**  $\mathbb{E}$ LZ  $\mathcal{O}(n \log n)$

Für **jede** Instanz (insb. auch für vorsortierte Folge)

**wahrscheinlich**  $\mathcal{O}(n \log n)$

## Art der Randomisierung: Monte Carlo

**Beispiel:** Geg.:  $n$  Zahlen  $Z$ . Gesucht: eine der  $n/3$  kleinsten.

**Deterministisch:** Minimum der ersten  $\frac{2}{3}n + 1$  Zahlen.

- ▶ Laufzeit:  $\mathcal{O}(n)$ , Ergebnis korrekt.

**Primitiv:** Wähle (zufällig uniform verteilt) eine Zahl aus  $Z$ .

- ▶ Laufzeit:  $\mathcal{O}(1)$
- ▶ Ergebnis mit Wahrscheinlichkeit  $w = 1/3$  korrekt.

**Mehrfach:** Führe **Primitiv**  $k$  mal aus, und gib Minimum zurück.

- ▶ Laufzeit:  $\mathcal{O}(k)$  Unabhängig von  $n!$
- ▶ Ergebnis mit Wahrscheinlichkeit  $w_k = 1 - (2/3)^k$  korrekt.  
 $\Rightarrow \lim_{k \rightarrow \infty} w_k = 1$   $\Rightarrow k = 10 \rightarrow 98\%$ ,  $k = 20 \rightarrow 99.96\%$

**Wahrscheinlich** korrekt, **immer** schnell  $\Rightarrow$  **Monte Carlo**

## Erwartungswert muss immer gelten!

Monte Carlo: Erwartungswert für **Korrektheit**

Las Vegas: Erwartungswert für **Laufzeit**

Atlantic City: Erwartungswert für **beides**

**Egal** welches Algorithmenmodell:

Eine Aussage bzgl. eines Erwartungswertes muss **für alle möglichen Eingaben** gelten!

Es reicht insbesondere **nicht**, wenn der Algorithmus bei fast allen Instanzen schnell/korrekt ist, aber bei **einer bestimmten** Instanz **immer** langsam/falsch ist!

- **Sie** geben den Algorithmus  $\mathcal{A}$  an.
- Ein **Feind** (engl. **Adversary**) sucht nun eine besonders gemeine Instanz  $\mathcal{I}$  aus.
- Wir berechnen nun **mehrmals**  $\mathcal{A}(\mathcal{I})$ .

⇒ Im Erwartungswert müssen die Garantien erfüllt werden!

## Art der Randomisierung: Fehler

Betrachte ein **Entscheidungsproblem**.

Sei  $w_J$  (bzw.  $w_N$ ) die Wahrscheinlichkeit, dass der Algorithmus bei einer JA-Instanz (bzw. NEIN-Instanz) **korrekt** antwortet.

▶ **Zweiseitiger Fehler:**

Monte Carlo Algorithmen (oder Atlantic City)

$$w_J < 1 \text{ und } w_N < 1$$

▶ **Einseitiger Fehler:**

Monte Carlo Algorithmen (oder Atlantic City)

$$w_J < 1 \text{ oder } w_N < 1 \text{ (nicht beide!)}$$

▶ **Nullseitiger Fehler:**

- Las Vegas Algorithmen (oder deterministisch-korrekt)

$$w_J = w_N = 1$$

- Antwortmöglichkeiten „ja“, „nein“, „keine Ahnung“

## Einseitiger Fehler: Randomisierte TM

**Randomisierte TM (RTM):** Betrachte eine NDTM. Wenn nicht eindeutig, wie weiter gehen: wähle **zufällig** eine der Möglichkeiten. RTM akzeptiert, falls der so gewählte Rechenweg in einem Endzustand endet.

**NDTM** akzeptiert, wenn mindestens einer der möglichen Rechenwege in einem Endzustand endet.

- ▶ NDTM akzeptiert  $\Rightarrow$  RTM **könnte** akzeptieren (wenn es zufälligerweise den Weg findet), muss aber nicht.  
W'keit dafür ist also  $> 0$ , aber i.A. sehr klein.
- ▶ NDTM akzeptiert nicht  $\Rightarrow$  RTM wird **nie** akzeptieren.

**Formale Finesse:** RTM ist **deterministisch!**

„Zufälligkeit“ mittels eines (vorab zufällig gener.) Bitstrings als Teil der Eingabe. Damit ist Rechenweg deterministisch.

## Einseitiger Fehler: $RP$

JA-Instanz der Größe  $n \iff \exists$  ein Weg  $W$  in der (N)DTM zu einem akzeptierenden Zustand.

$P$  &  $NP$ :  $W$  hat Länge  $w \leq \text{poly}(n)$

- ▶  $P$  – **Deterministic** Polynomial-time  
 $W$  ist in  $\mathcal{O}(w)$  Zeit findbar
- ▶  $RP$  – **Randomized** Polynomial-time  
 $W$  ist durch eine **RTM** in  $\mathcal{O}(w)$  Zeit mit W'keit  $\geq 1/\text{poly}(n)$  findbar  $\Rightarrow$  wir haben also eine sinnvolle Chance!
- ▶  $NP$  – **Non-deterministic** Polynomial-time  
 $W$  ist in  $\mathcal{O}(w)$  Zeit validierbar  $\Leftrightarrow$   
 $W$  ist durch eine **RTM** in  $\mathcal{O}(w)$  Zeit mit W'keit  $> 0$  findbar

**Konsequenz.**  $P \subseteq RP \subseteq NP$

# RP

## Definition RP (Randomized Polynomial-time)

Entscheidungsprobleme, für die ein **im Worst-Case polynomieller** Monte Carlo Algorithmus (mit einseitigem Fehler) existiert, der

- ▶ NEIN-Instanzen mit Wahrscheinlichkeit **1** (=immer) korrekt erkennt, und
- ▶ JA-Instanzen mit Wahrscheinlichkeit  $\geq 1/2$  korrekt erkennt.

**Huch!** Warum steht hier jetzt  $1/2$  statt  $1/\text{poly}(n)$ ?

Und selbst wenn  $1/2$  „besser“ ist als  $1/\text{poly}(n)$ : Was hilft ein Algorithmus, der in der Hälfte aller Fälle lügt?

## Fehler-W'keit pumpen: Konstanten

**Gegeben:** Polynomieller Alg.  $\mathcal{A}$  mit einseitigem Fehler:

- ▶  $w_N = 1$ : bei NEIN-Inst. antwortet  $\mathcal{A}$  immer korrekt „NEIN“
- ▶  $w_J = 0.2$ : bei JA-Inst. antwortet  $\mathcal{A}$  mit 20%iger W'keit korrekt „JA“, sonst „NEIN“

**Gesucht:** Polynomieller Algorithmus  $\mathcal{B}$  mit  $w_N = 1$ ,  $w_J = 0.99$ .

$\mathcal{B}$  := Führe  $\mathcal{A}$   $k$ -mal aus. Insgesamt *true* falls mind. 1x *true*.

W'keit für Fehler bei: **a** NEIN-Instanz: 0 ✓, **b** JA-Instanz:

- ▶  $k = 1$ :  $1 - w_J = 1 - 0.2 = 0.8 = \mathbf{80\%}$
- ▶  $k = 2$ :  $(1 - w_J)^2 = (1 - 0.2)^2 = 0.8^2 = 0.64 = \mathbf{64\%}$
- ▶ allg.  $k$ :  $(1 - w_J)^k = (1 - 0.2)^k = 0.8^k \dots \lim_{k \rightarrow \infty} = \mathbf{0\%}$

⇒ **99%** korrekt → W'keit für Fehler: **1% = 0.01**

Suche minimales  $k$  mit  $(1 - w_J)^k \leq 0.01 \Rightarrow 0.8^k \leq 0.01$

$k \geq \lceil \log_{0.8} 0.01 \rceil = \lceil 20.6377\dots \rceil = \mathbf{21} \Rightarrow \mathbf{\text{Konstant oft!}}$

## Fehler-W'keit pumpen: Polynome, $1/2$

**Gegeben:** Polynomieller Alg.  $\mathcal{A}$  mit  $w_N = 1$ ,  $w_j = 1/\text{poly}(n)$ .

$n$  = Eingabelänge;  $\text{poly}(\cdot)$  = beliebige polynomielle Funktion  
 $\Rightarrow$  bei großen Eingaben seltener, dass  $\mathcal{A}$  korrekt antwortet.

**Gesucht:** Polynomieller Algorithmus  $\mathcal{B}$  mit  $w_N = 1$ ,  $w_j = 0.99$ .

Wisse: **a**  $\lim_{m \rightarrow \infty} (1 - 1/m)^m = 1/e \approx 0.367879\dots$

**b**  $\forall m > 1: (1 - 1/m)^m < 1/e$  da monoton wachsend

**Hilfsalgorithmus**  $\mathcal{H}$  := Führe  $\mathcal{A}$   $\text{poly}(n)$  oft aus.

Fehler-W'keit von  $\mathcal{H} \leq (1 - \frac{1}{\text{poly}(n)})^{\text{poly}(n)} < 1/e$

**Algorithmus**  $\mathcal{B}$  := Führe  $\mathcal{H}$   $k$  mal aus.

Fehler-W'keit bei  $k$  Ausführungen von  $\mathcal{H}$ :  $< (1/e)^k$

Wir wollen:  $(1/e)^k \leq 0.01 \Rightarrow 100 \leq e^k$

$\Rightarrow k \geq \lceil \ln 100 \rceil = \lceil 4.605\dots \rceil = 5$

## Fehler-W'keit pumpen: Polynome, 2/2

**Gegeben:** Polynomieller Alg.  $\mathcal{A}$  mit  $w_N = 1$ ,  $w_j = 1/\text{poly}(n)$ .

$n$  = Eingabelänge;  $\text{poly}(\cdot)$  = beliebige polynomielle Funktion  
 $\Rightarrow$  bei großen Eingaben seltener, dass  $\mathcal{A}$  korrekt antwortet.

**Gesucht:** Polynomieller Algorithmus  $\mathcal{B}$  mit  $w_N = 1$ ,  $w_j = 0.99$ .

**Hilfsalgorithmus**  $\mathcal{H}$  := Führe  $\mathcal{A}$   $\text{poly}(n)$  oft aus.

**Algorithmus**  $\mathcal{B}$  := Führe  $\mathcal{H}$   $k = 5$  mal aus.

### Beweis: Polynomielle Laufzeit

Laufzeit von  $\mathcal{A}$ : polynomiell ✓

Laufzeit von  $\mathcal{H}$ :  $\text{poly}(n) \cdot \text{polyLaufzeit}_{\mathcal{A}}(n) \rightarrow$  polynomiell ✓

Laufzeit von  $\mathcal{B}$ :  $5 \cdot \text{polyLaufzeit}_{\mathcal{H}}(n) \rightarrow$  polynomiell ✓

## Fehler-W'keit pumpen bei $RP$

**Wir haben also gesehen:** Wir geben uns eine Wunschwahrscheinlichkeit  $0 < p < 1$  vor (z.B.  $p = 0.999$ ).

Gegeben ein Algorithmus  $\mathcal{A}_{\text{const}}$  mit Laufzeit  $\mathcal{O}(n^c)$ ,  $w_N = 1$  und  $w_j \geq q$  für irgendeine **Konstante**  $0 < q < 1$ .

**Konstant oft**es Ausführen von  $\mathcal{A}_{\text{const}}$   
 $\Rightarrow$  Algorithmus  $\mathcal{A}_{\text{const}}^*$  mit Laufzeit  $\mathcal{O}(n^c)$ ,  $w_N = 1$  und  $w_j \geq p$ .

Gegeben ein Algorithmus  $\mathcal{A}_{\text{poly}}$  mit Laufzeit  $\mathcal{O}(n^c)$ ,  $w_N = 1$  und  $w_j \geq 1/q(n)$  für irgendein **Polynom**  $q(n) := n^d$ .

**Polynomiell oft**es Ausführen von  $\mathcal{A}_{\text{poly}}$   
 $\Rightarrow$  Algorithmus  $\mathcal{A}_{\text{poly}}^*$  mit Laufzeit  $\mathcal{O}(n^{c+d})$ ,  $w_N = 1$  und  $w_j \geq p$ .

Es ist **egal** ob wir bei der Definition von  $RP$   $1/\text{poly}(n)$ ,  $1/2$  oder **99.99%** fordern! Schon ersteres führt **auch** zu letzterem!

## *RP* (nochmal) & *Co-RP*

### Definition *RP* (Randomized Polynomial-time)

Entscheidungsprobleme, für die ein **im Worst-Case polynomieller** Monte Carlo Algorithmus (mit einseitigem Fehler) existiert, der

- ▶ NEIN-Instanzen mit Wahrscheinlichkeit **1** (=immer) korrekt erkennt, und
- ▶ JA-Instanzen mit Wahrscheinlichkeit  $\geq 1/2$  korrekt erkennt.

**Beobachtung.** Der Wert  $1/2$  ist recht arbiträr. Durch polynomiell-fache Ausführung genügt jeder Wert  $\geq 1/\text{poly}(n)$ .

### Definition *Co-RP* (Complementary Randomized Poly.-time)

Analog zu *RP*, nur dass die Wahrscheinlichkeiten zu den JA- und NEIN-Instanzen vertauscht sind.

## Was sagt die Antwort eines $RP$ -Algorithmus $\mathcal{A}$ ?

JA-Instanz:  $\mathcal{A}$  findet mit W'keit  $w_J$  einen JA-Zeugen

$$\mathcal{A}(\text{JA-Instanz}) = \begin{cases} \text{„JA“} & \text{mit W'keit } w_J \\ \text{„NEIN“} & \text{mit W'keit } 1 - w_J \end{cases}$$

NEIN-Instanz:  $\mathcal{A}$  findet nie einen JA-Zeugen

$$\mathcal{A}(\text{NEIN-Instanz}) = \begin{cases} \text{„NEIN“} & \text{immer (= W'keit } w_N = 1) \end{cases}$$

**ACHTUNG:** Umdenken bei der Interpretation einer Antwort!

$$\mathcal{A}(\mathcal{I}) = \begin{cases} \text{„JA“} & \rightarrow \mathcal{I} \text{ ist } \mathbf{\text{immer}} \text{ eine JA-Instanz } ^1 \\ \text{„NEIN“} & \rightarrow \mathcal{I} \text{ ist } \mathbf{\text{entweder JA- oder NEIN-Instanz}} ^{2,3} \end{cases}$$

Lüge bei **JA-Instanz**  $\Rightarrow$  Lüge bei **NEIN-Antwort**. (und umgekehrt)

<sup>1</sup> denn es wurde ein JA-Zeuge gefunden

<sup>2</sup> kein JA-Zeuge gefunden: vielleicht  $\nexists$ , vielleicht hatten wir nur Pech

<sup>3</sup> man kann **nichts** über die jeweilige W'keit sagen,  
denn wir wissen nicht, wie häufig JA-Instanzen sind!

## Co-RP Beispiel: STRINGEQUALITY, 1/3

**Gegeben:** Bitstrings  $S_1, S_2$  der Länge  $n$ , auf zwei Rechnern.

**Frage:**  $S_1 = S_2?$  – ohne viele Bits über Netzwerk zu schicken!

Berechne einzeln auf jedem Rechner  $j \in \{1, 2\}$  mit String  $S_j$ :  
(gleicher Pseudozufallsgenerator/Seed auf den Rechnern)

**1** Bitpositionen  $T \subseteq \{1, \dots, n\}$ : Jede Position mit W'keit  $\frac{1}{2}$  in  $T$

**2**  $x_j = \bigoplus_{i \in T} S_j[i]$  // XOR über die identen Bitpos. in  $S_1$  &  $S_2$

Sende Bit  $x_2$  an Rechner **1**. Dort prüfe:

**■** **if**( $x_1 = x_2$ ) **return true** **else return false**

$S_1 = S_2 \Rightarrow$  immer *true*

$S_1 \neq S_2 \Rightarrow$  W'keit  $\alpha$  für *true*?

**Theorem.** Obiger Co-RP-Alg hat Fehler-W'keit  $\alpha = \frac{1}{2}$ .

## Co-RP Beispiel: STRINGEQUALITY, 2/3

Auf jedem Rechner  $j \in \{1, 2\}$  mit String  $S_j$  & gleichem Zufall:

- 1 Bitpositionen  $T \subseteq \{1, \dots, n\}$ : Jede Position mit W'keit  $\frac{1}{2}$  in  $T$
- 2  $x_j = \bigoplus_{i \in T} S_j[i]$  // XOR über die identen Bitpos. in  $S_1$  &  $S_2$

Sende Bit  $x_2$  an Rechner 1. Dort prüfe:

- **if**( $x_1 = x_2$ ) **return true** **else return false**

**Theorem.** Obiger Co-RP-Alg hat Fehler-W'keit  $\alpha = \frac{1}{2}$ .

**Beweis.** Wähle  $f$  mit  $S_1[f] \neq S_2[f]$ . Sei  $y_j = \bigoplus_{i \in T \setminus \{f\}} S_j[i]$ .

$$(x_j = y_j \oplus S_j[f] \text{ falls } f \in T, \quad x_j = y_j \text{ falls } f \notin T)$$

Falls  $y_1 \neq y_2$ : Alg. macht Fehler falls  $f \in T \rightarrow$  W'keit  $\frac{1}{2}$

Falls  $y_1 = y_2$ : Alg. macht Fehler falls  $f \notin T \rightarrow$  W'keit  $\frac{1}{2}$

Sei  $\beta$  die W'keit von  $y_1 = y_2$  ( $\Rightarrow y_1 \neq y_2$  hat W'keit  $1 - \beta$ )

W'keit, dass der Alg. Fehler macht:  $\beta \cdot \frac{1}{2} + (1 - \beta) \cdot \frac{1}{2} = \frac{1}{2}$  □

## Co-RP Beispiel: STRINGEQUALITY, 3/3

**Gegeben:** Bitstrings  $S_1, S_2$  der Länge  $n$ , auf zwei Rechnern.

**Frage:**  $S_1 = S_2?$  – ohne viele Bits über Netzwerk zu schicken!

**Theorem.** Unser Co-RP-Alg hat Fehler-W'keit  $\alpha = \frac{1}{2}$ .

Iteriere Algorithmus z.B. 256 mal  $\Rightarrow$  256 bits Kommunikation.

W'keit, dass Fehler nicht entdeckt wird:  $1/2^{256}$  .... ca.  $1 : 10^{77}$

Unabhängig von der Bitstringlänge  $n!!!$

**Vergleich:** 6-aus-49  $\approx 1 : 10^8 \rightarrow$  9x **hintereinander**  $1 : 10^{74}$

Alter des Universums  $\approx 10^{17}$  Sekunden

Atome im Universum  $\approx 10^{80}$

## Zweiseitiger Fehler: *PP*

### Definition *PP* (Probabilistic Polynomial-time)

Entscheidungsprobl., für die ein **im Worst-Case polynom.** Monte Carlo Algorithmus existiert, der JA- **und** NEIN- Instanzen jeweils mit Wahrscheinlichkeit  $w > \frac{1}{2}$  korrekt erkennt.

Warum sollte  $> \frac{1}{2}$  reichen? W'keit hochpumpen!

Nimm an,  $w = 0.75$ . Wende Algorithmus  $k$ -mal an ( $k \in \mathbb{N}_U$ ), und nimm die häufigste Antwort:

- ▶  $k = 3$ : Insg. korrekt wenn mind. 2x einzeln korrekt.  
W'keit, dass insg. korrekt: **0.84**
- ▶  $k = 5$ : Insg. korrekt wenn mind. 3x einzeln korrekt.  
W'keit, dass insg. korrekt: **0.90**

**Problem:** Wähle  $0 < \varepsilon \leq 1/\exp(n)$  sehr klein. Mit  $w = 0.5 + \varepsilon > 0.5$  mehr als polynomiell viele Wiederholungen nötig, damit Sicherheit  $\gg 0.5$ .  $\Rightarrow$  *PP* nutzt nicht viel!

## Zweiseitiger Fehler: *BPP*

**Definition *BPP*** (Bounded-error Probabilistic Poly.-time)

Entscheidungsprobl., für die  $\exists$  ein **im Worst-Case polynom.** Monte Carlo Algorithmus, der JA- **und** NEIN- Instanzen jeweils mit Wahrscheinlichkeit  $w \geq \frac{2}{3}$  korrekt erkennt.

Definition ident zu *PP*, bis auf Fehlerwahrscheinlichkeit!

**Intuition:** Fehler-W'keit ausreichend von  $\frac{1}{2}$  abgegrenzt (bounded), so dass Mehrfachausführung tatsächlich nennenswerte Fortschritte bzgl. gewünschter Antwortsicherheit bringt.

**Analog zu *RP*:** Der Wert  $\frac{2}{3}$  ist recht arbiträr.

Durch konstant-ofte Ausführung reicht jede **Konstante**  $> \frac{1}{2}$ .

Durch polynomiell-fache Ausführung genügt jeder Wert

$$\geq \frac{1}{2} + \frac{1}{\text{poly}(n)}.$$

## BPP Beispiel: KAUGUMMI, 1/2

**Gegeben:** Automat wurde durch  $2k + 1$  Kaugummipakete (jeweils rot oder blau) gefüllt. In roten (blauen) Paketen sind  $m$  rote (blaue) Kaugummis. Wir können am (undurchsicht.) Automaten einzelne Kaugummis (zufällig aus Vorrat) kaufen.  
**Gesucht:** Enthält Automat mehr rote oder blaue Kaugummis?

∈ **P**: Kaufe  $2km + 1$  Kaugummis. Die häufigere Farbe ist's.

∈ **BPP**: Kaufe  $1$  Kaugummi. Antworte seine Farbe.

**Formal:** Wort  $w \in \{R, B\}^{(2k+1)m}$  mit  $|\{i : w_i = R\}| = jm, j \in \mathbb{N}$ .  
 $|\{i : w_i = R\}| > |\{i : w_i = B\}|?$  Eingabelänge:  $n = (2k+1)m$

**Beweis.** Annahme: JA-Instanz  $\Rightarrow |\{i : w_i = R\}| \geq (k+1)m$ .

$$\mathbb{P}[\text{Kaugummi ist rot}] \geq \frac{(k+1)m}{(2k+1)m} = \frac{2k+1+1}{4k+2} = \frac{1}{2} + \frac{1}{4k+2}$$

$$\geq \frac{1}{2} + \frac{1}{\text{poly}(n)} \checkmark$$

## BPP Beispiel: KAUGUMMI, 2/2

∈ **P**: Kaufe  $2km + 1$  Kaugummis. Die häufigere Farbe ist's.

∈ **BPP**.: Kaufe  $1$  Kaugummi. Antworte seine Farbe.

Voriger Beweis hat nur „∈ BPP“ gezeigt. Nicht, dass der Algorithmus **in der Praxis** besser als der  $P$ -Alg. ist!

Wie oft muss man den **BPP-Algorithmus** anwenden, um eine Sicherheit von 99% zu erhalten?

**1** Falls  $m = 1$ :  $\#R \approx \#B \Rightarrow \mathbb{P}[\text{korrekt}] \approx \frac{1}{2} + \frac{1}{\mathcal{O}(n)}$  ✗

⇒  $\mathcal{O}(n)$  Versuche → kein Vorteil ggü.  $P$ -Algorithmus

**2** Falls  $k = 1$ :  $\#R = 2\#B \Rightarrow \mathbb{P}[\text{korrekt}] = \frac{2}{3}$  ✓

⇒ Nur **47** Versuche, unabhängig von  $m, n$ !

Wenn konstant viele Pakete (aber mit jew.  $m$  Kaugummis,  $m$  beliebig!): Für beliebige konstante Antwortsicherheit (z.B. 99.99%) muss man nur **konstant viele** Kaugummis ansehen!

## Nullseitiger Fehler: *ZPP*

**Definition *ZPP*** (Zero-error Probabilistic Polynomial-time)

**Zwei äquivalente Definitionen:**

(Beweis folgt)

- ▶ Ähnlich zu *PP* und *BPP*:

***ZPP*** – in Monte Carlo Form

*ZPP*(MC)

Entscheidungsprobleme, für die ein **im Worst-Case polynom.** Monte Carlo Algorithmus existiert, der JA- und NEIN-Instanzen jeweils mit Wahrscheinlichkeit  $w \geq \frac{1}{2}$  korrekt erkennt, und sonst „k.A.“ antwortet.

- ▶ Standard in Literatur:

***ZPP*** – in Las Vegas Form

*ZPP*(LV)

Entscheidungsprobleme, für die ein Las Vegas Alg. mit **erwartet polynomieller** Laufzeit existiert, der JA- und NEIN-Instanzen **immer** korrekt erkennt.

## ZPP(MC) = ZPP(LV), 1/2

**Theorem.**  $ZPP(MC)$  und  $ZPP(LV)$  sind das selbe.

**Beweisteil 1:**  $ZPP(MC)$ -Alg  $\mathcal{A}_{MC} \rightarrow ZPP(LV)$ -Alg  $\mathcal{A}_{LV}$

$\mathcal{A}_{MC}$  findet Antwort mit W'keit  $w \geq \frac{1}{2}$  in Laufzeit  $\mathcal{O}(n^c)$ .

$\mathcal{A}_{LV}(\mathcal{I}) :=$  **repeat**  $r := \mathcal{A}_{MC}(\mathcal{I})$  **until**  $r \neq$  „k.A.“; **return**  $r$

**Korrektheit:** immer ✓

**Laufzeit:**

$K$  = Anzahl der Iterationen (Zufallsvariable!).

$$\mathbb{P}[K = i] = (1 - w)^{i-1} w$$

Wann die meisten Iterationen?  $w = \frac{1}{2} \Rightarrow \mathbb{P}[K = i] = \frac{1}{2^i}$

$$\Rightarrow \mathbb{E}[K] \leq 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{16} + \dots = \sum_{i=1}^{\infty} i/2^i = 2$$

$\Rightarrow$  Erwartete Laufzeit: polynomiell ( $2 \times \mathcal{O}(n^c)$ )  $\Rightarrow ZPP(LV)$  ✓

## $ZPP(MC) = ZPP(LV)$ , 2/2

**Theorem.**  $ZPP(MC)$  und  $ZPP(LV)$  sind das selbe.

**Beweisteil 2:**  $ZPP(LV)$ -Alg  $\mathcal{A}_{LV} \rightarrow ZPP(MC)$ -Alg  $\mathcal{A}_{MC}$

$\mathcal{A}_{LV}$  findet korrekte Antwort in **erwarteter** poly-Laufzeit  $\mathcal{T}(n)$ .

$\mathcal{A}_{MC}(\mathcal{I}) :=$  Rechne die ersten  $2\mathcal{T}(n)$  Schritte von  $\mathcal{A}_{LV}(\mathcal{I})$   
**if** irgendein Resultat  $r$  erhalten **then return**  $r$   
**else return** „k.A.“

**Laufzeit:**  $\mathcal{O}(2 \cdot \mathcal{T}(n)) \rightarrow$  polynomiell ✓

**Korrektheit:** Markov'sche Ungleichung:  $\mathbb{P}[X \geq \alpha] \leq \mathbb{E}[X]/\alpha$

$\Rightarrow T =$  Laufzeit (Zufallsvariable!) von  $\mathcal{A}_{LV}(\mathcal{I})$  (ohne Abbruch)

$$\mathbb{P}[T \geq 2\mathcal{T}(n)] \leq \frac{\mathbb{E}[T]}{2\mathcal{T}(n)} = \frac{\mathcal{T}(n)}{2\mathcal{T}(n)} = \frac{1}{2}$$

$\Rightarrow$  W'keit, dass  $\mathcal{A}_{MC}(\mathcal{I})$  „k.A.“ liefert:  $\leq \frac{1}{2}$  ✓

□

# Las Vegas, Monte Carlo, Atlantic City

		polynomielle Laufzeit	
		immer	wahrscheinlich
Korrektheit	immer	deterministische, polynomielle Algorithmen  <i>P</i>	<b>Las Vegas</b>  <i>ZPP(LV)</i>
	wahrscheinlich	<b>Monte Carlo</b>  <i>RP, Co-RP,</i> <i>PP, BPP,</i> <i>ZPP(MC)</i>	<b>Atlantic City</b>

# Übersicht

Komplexitätsklasse		Poly. Laufzeit	$w_J$	kA	$w_N$
<i>RP</i>	Randomized Poly-time	Worst-Case	$\geq \frac{1}{2}^*$		1
<i>Co-RP</i>	Compl. Rand. Poly-time	Worst-Case	1		$\geq \frac{1}{2}^*$
<i>PP</i>	Prob. Poly-time	Worst-Case	$> \frac{1}{2}$		$> \frac{1}{2}$
<i>BPP</i>	Bounded-error Prob. Poly-time	Worst-Case	$\geq \frac{2}{3}^*$		$\geq \frac{2}{3}^*$
<i>ZPP</i> <sup>(MC)</sup>	Zero-error Prob. Poly-time	Worst-Case	$\geq \frac{1}{2}^*$	$\leq \frac{1}{2}$	$\geq \frac{1}{2}^*$
<sup>(LV)*</sup>		erwartet	1		1

\*\* = Durch polynomiell-häufige unabhängige Ausführung des Algorithmus' (= polynomielle Gesamtlaufzeit) kann man dann immer sogar die Wahrscheinlichkeit  $1 - \frac{1}{2^{\text{poly}(n)}}$  erreichen.

\* = Dazu reicht statt  $\frac{1}{2}$  sogar nur  $\frac{1}{\text{poly}(n)}$ .

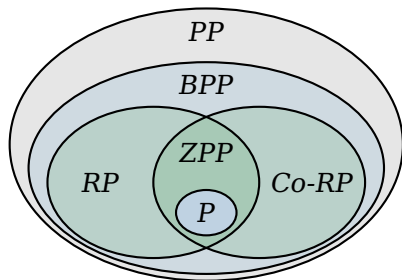
\* = Dazu reicht statt  $\frac{2}{3}$  sogar nur  $\frac{1}{2} + \frac{1}{\text{poly}(n)}$ .

\* = Las Vegas Algorithmus; alle anderen sind Monte Carlo.

## Folgerungen aus Definitionen

Komplexitätsklasse		Poly. Laufzeit	$w_j$	kA	$w_N$
$RP$	Randomized Poly-time	Worst-Case	$\geq \frac{1}{2}^*$		1
$Co-RP$	Compl. Rand. Poly-time	Worst-Case	1		$\geq \frac{1}{2}^*$
$PP$	Prob. Poly-time	Worst-Case	$> \frac{1}{2}$		$> \frac{1}{2}$
$BPP$	Bounded-error Prob. Poly-time	Worst-Case	$\geq \frac{2}{3}^*$		$\geq \frac{2}{3}^*$
$ZPP^{(MC)}$	Zero-error Prob. Poly-time	Worst-Case	$\geq \frac{1}{2}^*$	$\leq \frac{1}{2}$	$\geq \frac{1}{2}^*$
$(LV)^*$		erwartet	1		1

- ▶  $P \subseteq RP \subseteq NP$   
 $P \subseteq Co-RP \subseteq Co-NP$
- ▶  $Co-PP = PP$   
 $Co-BPP = BPP$   
 $Co-ZPP = ZPP$
- ▶  $RP \subseteq BPP$   
 $Co-RP \subseteq BPP$



## Klassenvergleiche: Zu zeigen...

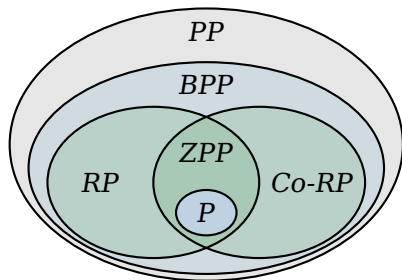
Komplexitätsklasse		Poly. Laufzeit	$w_j$	kA	$w_N$
$RP$	Randomized Poly-time	Worst-Case	$\geq \frac{1}{2}^*$		1
$Co-RP$	Compl. Rand. Poly-time	Worst-Case	1		$\geq \frac{1}{2}^*$
$PP$	Prob. Poly-time	Worst-Case	$> \frac{1}{2}$		$> \frac{1}{2}$
$BPP$	Bounded-error Prob. Poly-time	Worst-Case	$\geq \frac{2}{3}^*$		$\geq \frac{2}{3}^*$
$ZPP^{(MC)}$	Zero-error Prob. Poly-time	Worst-Case	$\geq \frac{1}{2}^*$	$\leq \frac{1}{2}$	$\geq \frac{1}{2}^*$
$(LV)^*$		erwartet	1		1

### Beweise folgen für:

- ▶  $ZPP \subseteq BPP$
- ▶  $RP \cap Co-RP = ZPP$
- ▶  $NP \subseteq PP$

### Große offene Frage:

- ▶  $P \stackrel{?}{=} BPP$



# ZPP $\subseteq$ BPP

**Theorem.**  $ZPP \subseteq BPP$

**Beweis.** Gegeben:  $ZPP(MC)$ -Algorithmus  $\mathcal{A}_Z$

Wir erstellen damit einen  $BPP$ -Algorithmus  $\mathcal{A}_B$ :

```

1  $r := \mathcal{A}_Z(I)$ 
 $\mathcal{A}_B(I) :=$  2 if( $r \neq$  „k.A.“) return  $r$ 
3 50:50 Chance ob return JA oder return NEIN

```

@**2**: Antwort korrekt;      @**3**: Antwort mit W'keit  $\frac{1}{2}$  (in)korrekt

$\Rightarrow$  W'keit, dass Schritt **3** eintritt:  $\leq \frac{1}{2}$

$\Rightarrow$  W'keit, dass  $\mathcal{A}_B$  falsch antwortet:  $\leq \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4} \leq \frac{1}{3}$  □

# $RP \cap Co-RP = ZPP$

**Theorem.**  $RP \cap Co-RP = ZPP$

**Beweisteil  $\supseteq$ .** Gegeben:  $ZPP_{(MC)}$ -Alg  $\mathcal{A}_Z$

$RP$ -Algorithmus  $\mathcal{A}_R(I) :=$

```

1 if( $\mathcal{A}_Z(I) = JA$ ) return JA
2 return NEIN
  
```

Invertiere Logik für  $Co-RP$  Algorithmus

@**1**: Antwort korrekt;      @**2** Nur korrekt für **NEIN**-Instanzen

$\Rightarrow$  **NEIN**-Instanz: immer richtig beantwortet  $\rightarrow w_N = 1 \checkmark$

$\Rightarrow$  W'keit, dass  $\mathcal{A}_R(I)$  **JA**-Instanz korrekt beantwortet

= W'keit, dass  $\mathcal{A}_Z(I)$  **JA**-Instanz korrekt beantwortet:  $\geq \frac{1}{2} \checkmark$

**Beweisteil  $\subseteq$ .** Gegeben:  $RP$ -Alg  $\mathcal{B}_R$  und  $Co-RP$ -Alg  $\mathcal{B}_C$

$ZPP_{(MC)}$ -Alg.  $\mathcal{B}_Z(I) :=$

```

1 if( $\mathcal{B}_R(I) = JA$ ) return JA
2 if( $\mathcal{B}_C(I) = NEIN$ ) return NEIN
3 return „k.A.“
  
```

Ergebnis in **1** und **2** immer korrekt. W'keit, dass **3**:  $\leq \frac{1}{2}$   $\square$

$NP \subseteq PP$ **Theorem.**  $NP \subseteq PP$ **Beweis.** Betrachte bel. Problem aus  $NP$  mit zugeh. NDTM  $M$ .  
 $M$ -als-RTM akzeptiert mit W'keit  $w > 0$ .Wir zeigen, es gibt dann auch einen  $PP$ -Algorithmus. Wähle  
 $0 < \varepsilon \ll w$ .**1 Falls  $M$ -als-RTM liefert JA: return JA** $PP$ -Alg.: **2 Mit W'keit  $\frac{1+\varepsilon}{2}$ : return NEIN****3 return JA**▶ **NEIN-Instanz liefert NEIN mit W'keit  $> \frac{1}{2}$** Algorithmus endet nie in **1**  $\rightarrow$  endet in **2** mit W'keit  $> \frac{1}{2}$  ✓▶ **JA-Instanz liefert JA mit W'keit  $> \frac{1}{2}$   $\Leftrightarrow$** **JA-Instanz liefert NEIN mit W'keit  $< \frac{1}{2}$** Ende @**1** mit W'keit  $w \rightarrow$  erreiche Zeile **2** mit W'keit  $1-w$ . $\Rightarrow$  W'keit für NEIN @**2**:  $(1-w)\frac{1+\varepsilon}{2} < \frac{1}{2}$  (da  $\varepsilon \ll w$ ) ✓  $\square$

## Monte Carlo — Weitere Beispiele

### ► Polynomial Identity Testing (PIT)

Sind zwei multivariate Polynome hohen Grades „gleich“?  
Äquivalente Frage: Ist ein gegebenes Polynom immer 0?

Kanonisierendes Ausmultiplizieren: exponentiell :-)

Nicht bekannt, ob in  $P$ .

**ABER:** Trivialer  $Co-RP$  Algorithmus!

(Setze ein paar zufällige Werte in das Polynom ein)

### ► Miller-Rabin Primzahl Test

Monte Carlo Algorithmus mit einseitigem Fehler:  $Co-RP$   
Findet mit  $W$ 'keit  $\geq 3/4$  heraus, wenn die Zahl  
zusammengesetzt ist

In der Praxis sehr schnell und „sicher“  
z.B. Primzahl-Generierung in OpenPGP

## Randomisierung — Weitere Beispiele

- ▶ **Cache Eviction Strategien** (Cache:  $k$  Elemente)

**Farthest-in-Future:** perfekt, aber i.A. Zukunft unbekannt.

Sei  $\gamma$  diese minimale # an Cache-Misses.

**Least Recently Used:** deterministischer Klassiker.

$\Rightarrow \exists$  Instanzen mit  $\mathcal{O}(k \cdot \gamma)$  Cache-Misses.

**Randomized Marking:** wählt **zufälliges** altes Element.

$\Rightarrow$  Nur  $\mathcal{O}(\log(k) \cdot \gamma)$  erwartete Cache-Misses!

- ▶ oft bei **Approximationsalgorithmen**

betrachte polynomielle Algorithmen für NP-schwere Optimierungsprobleme

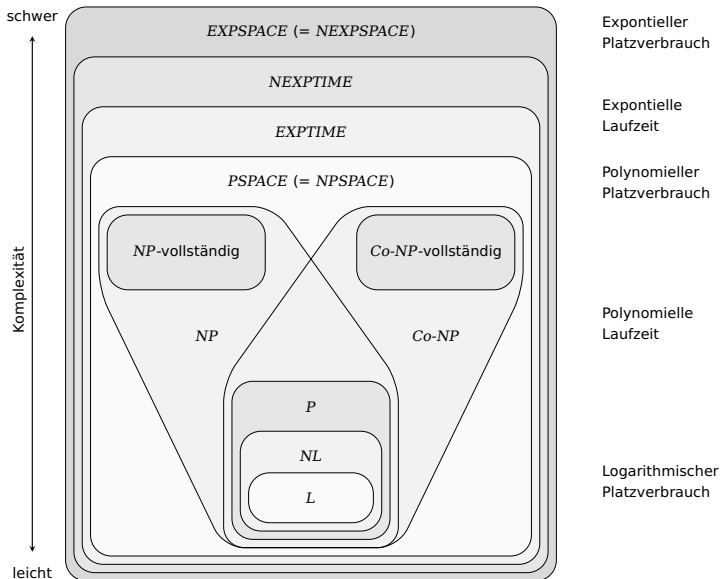
(z.B. **MINIMUMVERTEXCOVER**, **MAXIMUMCLIQUE**,...)

**Erwartete Approximationsgüte:** Wie weit wahrscheinlich maximal vom Optimum entfernt?

# **Komplexitätstheorie**

Jenseits polynomieller Laufzeit  
und Ausblick

# Mehr... Zwischen allen benachbarten Klassen ist offen ob $\subset$ oder $=$ !



## Wissenswertes

**Beobachtung.** Ein Problem aus  $PSPACE$  kann immer in exponentiell viel Zeit gelöst werden!

**Beweis.** Determ. Alg.: aktueller Speicherzustand determiniert den Rechenweg; jeder Rechenschritt ändert Speicherzustand.

$b$  Bits  $\rightarrow 2^b$  verschiedene Speicherzustände.

$\Rightarrow PSPACE$ : Speicherbedarf  $\text{poly}(n)$

$\Rightarrow 2^{\text{poly}(n)} = \text{exp}(n)$  Speicherzustände

$\Rightarrow \max \text{exp}(n)$  Schritte oder Endlosschleife. □

### Rätsel und Spiele:

- ▶ Viele **Puzzle und Rätsel** sind **NP-vollständig**: Kreuzworträtsel, verallgemeinertes Sudoku, Minesweeper, Lemmings, Bejeweled/CandyCrush,...
- ▶ Viele **Spiele** sind **PSPACE-vollständig**: verallg. Dame, vereinfachtes Go, Sokoban, Prince of Persia (1989),...

## Beispiele

### ***PSPACE*-vollständig (*PSPACE*-complete):**

- ▶ Wortproblem in kontextsensitiven Sprachen
- ▶ Canadian Traveler Problem

### ***NL*-vollständig (*NL*-complete):**

- ▶ 2-SAT
- ▶ *st*-connectivity in gerichteten Graphen
  - ▶ Laufe von *s* aus nicht-deterministisch durch den Graphen (=rate den richtigen Weg, falls er existiert).
  - ▶ Zähle Anzahl *c* der Schritte: JA-Instanz falls *t* gefunden; NEIN-Instanz falls  $c > |V(G)|$ .
  - ▶ *c* benötigt nur  $\mathcal{O}(\log |V(G)|) = \mathcal{O}(\log |G|)$  Bits.

## Und noch mehr...

- ▶ **Approximationsalgorithmen:** Finde eine Lösung für ein Optimierungsproblem, die beweisbar nicht **allzu weit** vom Optimum entfernt ist. Komplexitätsklassen (z.B. **APX, PTAS, FPTAS,...**) geben an, wie gut sich ein Optimierungsproblem approximieren lässt.
- ▶ **Abzählprobleme:** **Wieviele** zulässige Lösungen hat ein Entscheidungsproblem?
- ▶ **Schaltkreiskomplexität:** Wie groß muss ein Schaltkreis sein, um ein Problem lösen/berechnen zu können?
- ▶ **Beweissysteme:** Interaktive Beweise („Arthur-Merlin“) (**IP**), Probabilistically Checkable Proofs (**PCP**), Zero-Knowledge Proofs (**ZKP**)

Für all das und vieles weiteres leider keine Zeit... **apropos!**

# Jedes Ende ist ein neuer Anfang...

## Tolle Vorlesung im nächsten Wintersemester!

### ▶ **Approximationsalgorithmen**

2VO + 2UE, 6 LP, Bachelor, ALG-Säule

- ▶ NP-schwere Optimierungsprobleme:  
Exakt? Heuristik? ⇒ **Approximation:**  
*beweisbar* gute Lösung in *polynomieller* Zeit  
(„maximal 10% teurer als Optimum“)
- ▶ Allgemeine Techniken & spezifische Probleme
- ▶ Umzugskisten packen (BINPACKING),
- ▶ Museum überwachen (VERTEXCOVER),
- ▶ Museum ausrauben (KNAPSACK),
- ▶ Urlaubstouren planen (TRAVELINGSALESMAN),
- ▶ Abläufe in Fabriken planen (SCHEDULING),
- ▶ Fernheiznetzwerke und CPU-Designs planen (STEINERTREE),...

# Jedes Ende ist ein neuer Anfang...

## Und sonst so...

### ▶ Bachelor (ALG-Säule):

**Komplexitätstheorie** (SoSe 26) 2V+2U, 6 LP (*Fritz Bökler*)

**Algorithmen II** (WiSe 26/27) 2V+2U, 6 LP

### ▶ Master (MA-Säule):

**Fortgeschrittene Graphenalgorithmen** (WiSe 25/26)

4V+2U, 9 LP

**Invitation to Research in Graph Theory and**

**Algorithms** (SoSe 26) 4VU, 6 LP

**Algorithm Engineering** (WiSe 26/27) 2V+4U, 9 LP

**Algorithmische Mehrkriterielle Optimierung** (SoSe 27)

2V+2U, 6 LP (*Fritz Bökler*)

# Ende

**Vielen Dank für Ihre Aufmerksamkeit und  
viel Erfolg bei der Klausur!**

**Fragen?**

- 1 Einleitung**
  - Thema der Vorlesung
  - Organisatorisches
  - Formales Handwerkszeug
- 2 Informationstheorie**
  - Information
  - Entropie
  - Huffman-Codes
- 3 Formale Sprachen & Automaten**
  - Sprachen & Chomsky-Hierarchie
  - Reguläre Ausdrücke
  - Endliche Automaten
  - Beschreibungsäquivalenz regulärer Sprachen
  - Reguläres Pumping Lemma und Abschlusseigenschaften
  - Kontextfreie Sprachen & Kellerautomaten
  - Rekursiv Aufzählbare Sprachen & Turingmaschinen
- 4 Berechenbarkeitstheorie**
  - Rechnende Turingmaschinen
  - Von der TM zum Computer
  - Berechenbarkeit
  - Entscheidbarkeit
  - Halteproblem
  - Weitere unentscheidbare Probleme und unberechenbare Funktionen
- 5 Komplexitätstheorie**
  - P vs. NP
  - NP-Vollständigkeit
  - Erstes NP-vollständiges Problem
  - Weitere NP-vollständige Probleme
  - Starke & Schwache NP-Vollständigkeit
  - Fixed Parameter Tractability
  - Co-NP
  - Probabilistische Komplexitätsklassen
  - Jenseits polynomieller Laufzeit und Ausblick