

Theoretische Informatik

Prof. Dr. Barbara Staehle

HTWG Konstanz
Fakultaet für Informatik

WS 2021/2022

Teil VI

Berechenbarkeit, Entscheidbarkeit und Komplexität

Ziel dieses Kapitels

Kapitel II - V:

Grammatiken Wie erzeuge ich Sprachen?

Automaten

- Wie kann ich Sprachen erkennen?
- Wie kann ich Systeme / Algorithmen modellieren?
- Wie kann ich ein Eingabe- in ein Ausgabewort umsetzen?

Kapitel VI:

- Welche Funktionen kann ein Computer überhaupt berechnen?
- Welche Modelle kann ich nutzen, um eine Berechnung zu modellieren?
- Was ist und was kann das Modell „Turing-Maschine“?
- Welche Probleme sind überhaupt lösbar?
- Wie schwer ist es, ein Problem zu lösen?

Teil VI BEK

1. Berechenbarkeit

- 1.1 Church-Turing-These
- 1.2 (Turing-)Berechenbarkeit
- 1.3 Universelle Turing-Maschinen

2. Entscheidbarkeit und Unentscheidbarkeit

- 2.1 Entscheidbarkeit
- 2.2 Unentscheidbare Probleme

3. Komplexität

- 3.1 Zeit- und Raum-Komplexität
- 3.2 Komplexitätsklassen
- 3.3 NP-Vollständigkeit

Abschnitt 1

Berechenbarkeit

Theoretische Berechnungsmodelle I

Bemerkung: Turing-Maschinen sind nicht das einzige theoretische Modell, um Berechenbarkeit zu überprüfen. Unterscheidungsmerkmal: Berechnung **partieller** Funktionen möglich?

Loop-Programme Ähnlich zu modernen

Programmiersprachen - enthält Konstanten, Variablen, Operatoren für Vorgänger, Nachfolger, Zuweisungen und Kompositionen, loop ... do ... end Schleife (endlich viele Schleifendurchgänge)
 ⇒ kann nur **totale** Funktionen berechnen

While-Programme sind Loop-Programme mit einer zusätzlichen while ... do ... end Schleife

Goto-Programme sind Loop- bzw.

While-Programme ohne Schleifen, aber mit einem bedingten Sprungbefehl if ... goto ...

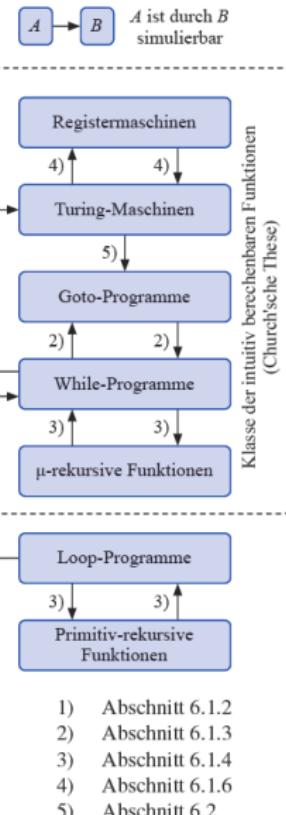


Bild: Äquivalenz der verschiedenen Berechnungsmodelle [Hoffmann, 2011]

Theoretische Berechnungsmodelle II

Primitiv-rekursive Funktionen Mathematisches

Modell: Funktionen werden aus primitiv-rekursiven Basis-Funktionen (Nullfunktion, Nachfolgerfunktion, Projektion) zusammengesetzt
⇒ kann nur **totale** Funktionen berechnen

μ -rekursive Funktionen sind primitiv-rekursiven Funktionen die zusätzlich noch die μ -Rekursion enthalten

Registermaschinen (RAM) Hardwarenahes Modell mit Eingabeband, Ausgabeband, Zentraleinheit mit Arbeitsspeicher, Befehlszähler, Programm und Speicher

Turing-Maschinen Theoretisches, sehr allgemeines Modell mit zentraler Logik, das Lesen und Beschreiben eines Arbeitsbandes erlaubt

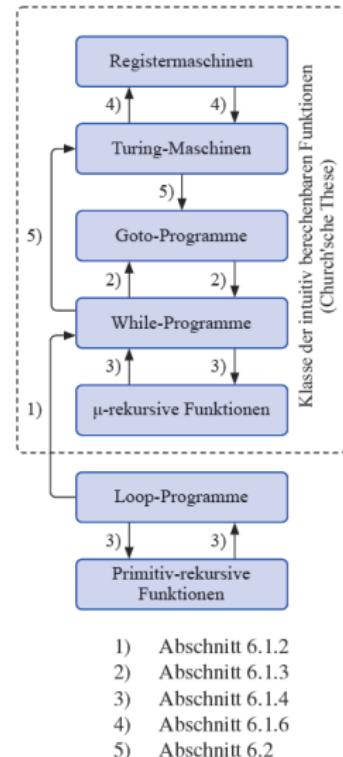


Bild: Äquivalenz der verschiedenen Berechnungsmodelle [Hoffmann, 2011]

Church-Turing-Theorie

Zusammenfassung: Bis auf die Loopsprache und primitive Rekursion sind alle Berechnungsmodelle gleich ausdrucksstark!

Church-Turing-Theorie (1936)

Die Klasse der Turing-berechenbaren Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein.

Bemerkungen:

- benannt nach Alonzo Church und Alan Turing
- nicht beweisbar, da Begriff „Intuitiv berechenbar“ nicht exakt formalisiert ist, aber weitverbreitete Annahme: stimmt.
- Übersetzung: alles was ein moderner Computer berechnen kann, kann eine Turing-Maschine (in mehr Zeit) auch berechnen!

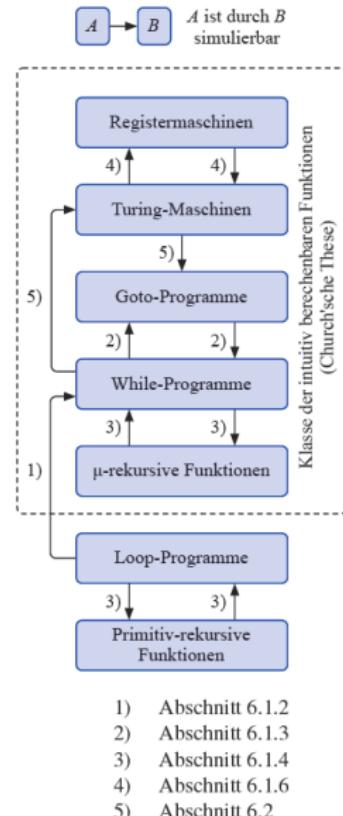
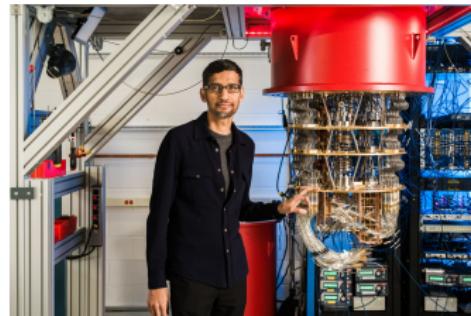


Bild: Äquivalenz der verschiedenen Berechnungsmodelle [Hoffmann, 2011]

Church-Turing-These und Quantum Supremacy

- heise.de, 24.09.19: Google:
Überlegenheit von Quantencomputern bewiesen
- **Frage:** Gilt die Church-Turing-These nicht für Quantencomputer? Können Quantencomputer tatsächlich mehr Probleme lösen als klassische Computer und Turing-Maschinen??
- **Antwort: Nein!** Googles QC Symacore hat **ein** Problem, sehr viel schneller gelöst als ein heutiger Supercomputer.
- Schon bewiesen, technisch noch umzusetzen: Es gibt auch praktisch relevante Probleme, die Quantencomputer schneller lösen können (z.B. Faktorisierung).



Quelle: [blog.google](#)

Arute, F., Arya, K.,
Babbush, R. et al.,
„Quantum supremacy using
a programmable
superconducting processor“
Nature 574, 505–510 (2019)

Erinnerung: Arbeitsweise einer TM I

- Zu Beginn
 - ▶ befindet sich die TM im Zustand q_0 ,
 - ▶ auf dem unendlich langen Arbeitsband steht das Eingabewort ω , links und rechts davon Blanks,
 - ▶ Schreib-/Lesekopf steht auf erstem Zeichen von ω .
- In jedem Verarbeitungsschritt führt die Maschine die folgenden Aktionen aus:
 1. Einlesen des aktuellen Bandzeichens σ
 2. Berechnung (für aktuellen Zustand q) von $\delta(q, \sigma) = (q', \sigma', b)$.
 - 2.1 Ersetzung des aktuellen Bandzeichens σ durch σ' .
 - 2.2 Bewegung des Lesekopfes nach links ($b = \leftarrow$) oder rechts ($b = \rightarrow$)
 - 2.3 Zustandsübergang in den Folgezustand q'
- Nach der Verarbeitung des Eingabewortes hat die TM
 - ▶ das Wort ENTWEDER akzeptiert ODER nicht akzeptiert (letztes Kapitel)
 - ▶ UND ein Ergebnis berechnet (dieses Kapitel)

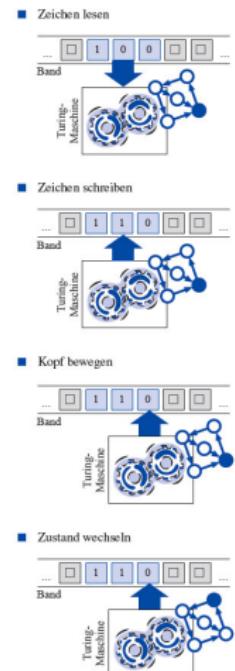


Bild: Arbeitsweise einer TM
[Hoffmann, 2011]

Erinnerung: Arbeitsweise einer TM II

- Die Verarbeitung des Eingabewortes ist zu Ende, falls für das aktuell gelesene Symbol und den aktuellen Zustand kein nächster Schritt definiert ist.
- Falls Endzustand akzeptierender Zustand ist:
 - ▶ Berechnung erfolgreich, Eingabewort akzeptiert.
 - ▶ Der Inhalt des Arbeitsbandes ab dem Schreib-/Lesekopf ist das Ergebnis der Berechnung.
- Andernfalls: Berechnung nicht erfolgreich, Eingabewort nicht akzeptiert.

Weitere Definitionsvarianten:

- „L“, „R“ statt \leftarrow , \rightarrow für die Bewegungen; sowie „nicht bewegen“ erlaubt: \leftarrow , \rightarrow , \circlearrowleft bzw. „L,R,0“
- Kein Endzustand, statt dessen Anweisung „H“ (für Halten) zum Beenden der Berechnung.
- Kein Zurücksetzen des Kopfes, Ergebnis = gesamter Bandinhalt.

Turing-Berechenbarkeit

Beobachtung: Turing-Maschinen können Funktionen berechnen.

Definition

Sei $f : \Sigma^* \rightarrow \Sigma^*$ eine beliebige partielle Funktion über Wörtern des Eingabealphabets Σ . f heißt **Turing-berechenbar**, falls eine Turing-Maschine $T = (Q, \Sigma, \Pi, \delta, q_0, F)$ mit den folgenden Eigenschaften existiert:

1. T terminiert unter Eingabe von ω genau dann in einem Endzustand $q_f \in F$, falls $f(\omega)$ definiert ist ($f(\omega) \neq \perp$).
2. Falls $f(\omega)$ definiert ist, gilt: $(\square, q_0, \omega) \vdash^* (\square^*, q_f, f(\omega)\square^*)$ (T erhält ω als Eingabe, terminiert in q_f und liefert Ausgabe $f(\omega)$).

Bemerkungen: T liefert ein undefiniertes Ergebnis $f(\omega) = \perp$ (notwendig für die Berechnung partieller Funktionen), indem T

- die Berechnung in einem Zustand $q \notin F$ beendet,
- in eine Endlossschleife gerät und die Berechnung nie beendet.

Turing-Berechenbarkeit - weitere Bemerkungen

- Für jede TM T mit Startzustand q_0 ist (\square, q_0, ω) die **Startkonfiguration** aus welcher T die Berechnung für ω startet: T befindet sich im Zustand q_0 , der Kopf ist über dem ersten Zeichen von ω , links vom Kopf steht ein Blank.
- Während der Berechnung von $f(\omega)$ kann (muss nicht) T beliebig viele Felder über das ursprüngliche Eingabewort hinauslaufen. Die Blanks für die besuchten Felder können (müssen nicht) in der **Endkonfiguration** dargestellt werden, weshalb diese die Form $(\square^*, q_f, f(\omega)\square^*)$ (0 oder beliebig viele Blanks links und rechts von $f(\omega)$) hat.
- Die Turing-Berechenbarkeit ist nicht über \mathbb{N} sondern über einem beliebigen Alphabet Σ definiert. Zwei Lösungsansätze, um auch Funktionen über \mathbb{N} berechnen zu können
 - ▶ Zahlen **binär** codieren mit $\Sigma = \{0, 1\}$. Vorteil: tatsächliche Arbeitsweise von Computern. Nachteil: unnötig komplizierte TMs.
 - ▶ Zahlen **unär** codieren mit $\Sigma = \{1\}$. Vorteil: durch TMs sehr einfach zu verarbeiten (siehe T_1). Nachteil: unnötig lange Bearbeitungszeit.

Turing-berechenbare Funktionen: die Nachfolgerfunktion

- $\Sigma = \{1\}$
- $f : \Sigma^* \rightarrow \Sigma^*$ ist totale Funktion, Ergebnis ist definiert für alle Eingabewerte
 - ▶ Generell: $1^n \mapsto 1^{n+1}$
 - ▶ Beispiel: $, \varepsilon \mapsto 1, 1 \mapsto 11, 11 \mapsto 111, \dots$
- f ist Turing-berechenbar, da die TM

$$T_1 = (Q, \Sigma, \Pi, \delta, q_0, F) = (\{q_0, q_1, q_2\}, \{1\}, \{1, \square\}, \delta, q_0, \{q_2\})$$

existiert, für welche

1. unter Eingabe von allen $\omega \in \Sigma^*$ die Berechnung im Endzustand q_2 endet (da Ergebnis von $f(\omega)$ immer definiert ist),
2. für alle Eingabewort $\omega = 1^n \in \Sigma^*$ gilt: $(\square, q_0, 1^n) \vdash^* (\square\square, q_2, 1^{n+1})$

(siehe Beispiel Arbeitsweise von T_1 im letzten Kapitel)

Turing-berechenbare Funktionen: die Addition I

- Gesucht: TM T_+ welche $f_+(x_1, x_2) = x_1 + x_2$ berechnet.
- Basisideen:
 - ▶ Codiere Eingaben x_1, x_2 unär, trenne beide Eingaben durch eine 0.
 - ▶ T_+ muss unter solch einer Eingabe
 1. die trennende 0 suchen und durch eine 1 ersetzen
(Zwischenergebnis $x_1 + x_2 + 1$),
 2. am Ende des Bandes die letzte 1 löschen (Endergebnis $x_1 + x_2$),
 3. den Schreib-/Lesekopf auf die erste 1 (1. Zeichen des Ergebniswortes) zurückbewegen.
- Lösung:
 - ▶ $T_+ = (Q, \Sigma, \Pi, \delta, q_0, F)$ mit
 - ▶ $Q = \{q_0, q_1, q_2, q_3, q_4\}$
 - ▶ $\Sigma = \{1, 0\}$
 - ▶ $\Pi = \{1, 0, \square\}$
 - ▶ $F = \{q_4\}$
 - ▶ δ siehe Tabelle rechts

δ	1	0	\square
q_0	$(q_0, 1, \rightarrow)$	$(q_1, 1, \rightarrow)$	-
q_1	$(q_1, 1, \rightarrow)$	-	$(q_2, \square, \leftarrow)$
q_2	$(q_3, \square, \leftarrow)$	-	-
q_3	$(q_3, 1, \leftarrow)$	-	$(q_4, \square, \rightarrow)$
q_4	-	-	-

Turing-berechenbare Funktionen: die Addition II

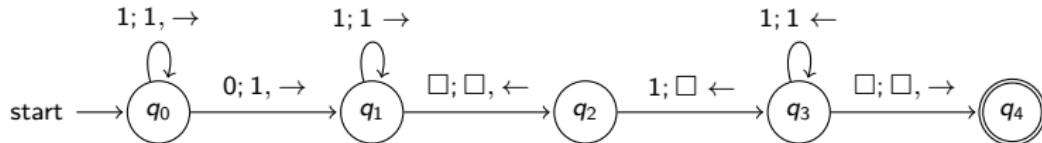


Bild: Erweitertes Zustandsübergangsdiagramm für T_+

Eingabe $\omega = 11011$:

$(\square,$	$q_0,$	$11011)$
$\vdash (\square 1,$	$q_0,$	$1011)$
$\vdash (\square 11,$	$q_0,$	$011)$
$\vdash (\square 111,$	$q_1,$	$11)$
$\vdash (\square 1111,$	$q_1,$	$1)$
$\vdash (\square 11111,$	$q_1,$	$\square)$
$\vdash (\square 1111,$	$q_2,$	$1\square)$
$\vdash (\square 111,$	$q_3,$	$1\square\square)$
$\vdash (\square 11,$	$q_3,$	$11\square\square)$
$\vdash (\square 1,$	$q_3,$	$111\square\square)$
$\vdash (\square,$	$q_3,$	$1111\square\square)$
$\vdash (\square\square,$	$q_4,$	$1111\square\square)$

\Rightarrow Berechnung erfolgreich
Ergebnis ist 1111

Eingabe $\omega = 1111$:

$(\square,$	$q_0,$	$1111)$
$\vdash (\square 1,$	$q_0,$	$111)$
$\vdash (\square 11,$	$q_0,$	$11)$
$\vdash (\square 111,$	$q_0,$	$1)$
$\vdash (\square 1111,$	$q_0,$	$\square)$
\Rightarrow Berechnung nicht erfolgreich		
(Eingabewort ungültig)		
kein Ergebnis		

Hier klicken für
Nachvollziehen der
Arbeitsweise im
Turing-Maschinen-Simulator

Eingabe $\omega = 01$:

$(\square,$	$q_0,$	$01)$
$\vdash (\square 1,$	$q_1,$	$1)$
$\vdash (\square 11,$	$q_1,$	$\square)$
$\vdash (\square 1,$	$q_2,$	$1\square)$
$\vdash (\square,$	$q_3,$	$1\square\square)$
$\vdash (\square\square,$	$q_4,$	$1\square\square)$

\Rightarrow Berechnung erfolgreich
Ergebnis ist 1

Turing-berechenbare Funktionen: die undefinierte Funktion

- $\Sigma = \{1\}$
- $u : \Sigma^* \rightarrow \Sigma^*$ ist partielle Funktion, da undefiniert für alle Eingaben
- Für alle $\omega \in \Sigma$ gilt: $u : \omega \mapsto \perp$
- u ist Turing-berechenbar, durch die TM

$$T_u = (Q, \Sigma, \Pi, \delta, q_0, F) = (\{q_0, q_1\}, \{1\}, \{1, \square\}, \delta, q_0, \{q_1\})$$

mit δ gegeben durch:

δ		1		\square
q_0		$(q_1, 1, \rightarrow)$		$(q_1, \square, \rightarrow)$
q_1		$(q_0, 1, \leftarrow)$		$(q_0, \square, \leftarrow)$

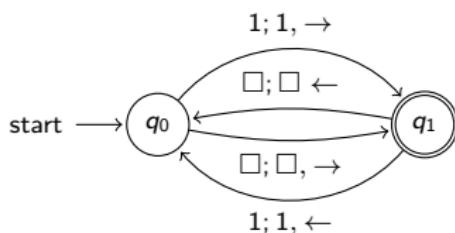


Bild: Erweitertes Zustandsübergangsdiagramm für T_u

Arbeitsweise von T_u für Eingabewort $\omega = 11$:

- $(\square, q_0, 11)$
- $(\square 1, q_1, 1)$
- $(\square, q_0, 11)$
- $(\square 1, q_1, 1)$
- $(\square, q_0, 11)$
- $(\square 1, q_1, 1)$
- $(\square, q_0, 11)$
- $(\square 1, q_1, 1)$
- $(\square, q_0, 11)$
- $(\square 1, q_1, 1)$
- $(\square, q_0, 11)$
- \dots

Turing-berechenbare Funktionen: die Verschiebefunktion I

- **Gesucht:** TM T_V welche jedes nichtleere Eingabewort über dem Alphabet Σ um eine Zelle nach rechts verschiebt.
- **Basisideen:**
 - ▶ Verwende für jedes Symbol aus Σ einen Zustand und gehe bei Lesen des Eingabezeichens σ in den Zustand q_σ über.
 - ▶ Überschreibe jedes gelesene Zeichen τ durch das Vorgängerzeichen σ , welches in q_σ gespeichert ist.
 - ▶ Beende den Lese-/Schreibvorgang durch Schreiben des letzten Zeichens, falls ein Blank gelesen wurde.
 - ▶ Gehe zum Wortanfang zurück und bleibe auf dem ersten gefundenen Blank stehen (verwende hierfür die Abkürzung \circlearrowleft)

- **Lösung für $\Sigma = \{x, y\}$:**

$T_V = (Q, \Sigma, \Pi, \delta, q_0, F)$ mit

- ▶ $Q = \{q_0, q_1, q_x, q_y\}$
- ▶ $\Pi = \Sigma \cup \{\square\}$
- ▶ $F = \{q_1\}$
- ▶ δ siehe Tabelle rechts

δ	\square	x	y	\circlearrowleft
q_0	$(q_x, \square, \rightarrow)$	$(q_y, \square, \rightarrow)$	$(q_0, \square, \circlearrowleft)$	
q_x	(q_x, x, \rightarrow)	(q_y, x, \rightarrow)	(q_1, x, \leftarrow)	
q_y	(q_x, y, \rightarrow)	(q_y, y, \rightarrow)	(q_1, y, \leftarrow)	
q_1	(q_1, x, \leftarrow)	(q_1, y, \leftarrow)	$(q_1, \square, \circlearrowleft)$	

Turing-berechenbare Funktionen: die Verschiebefunktion II

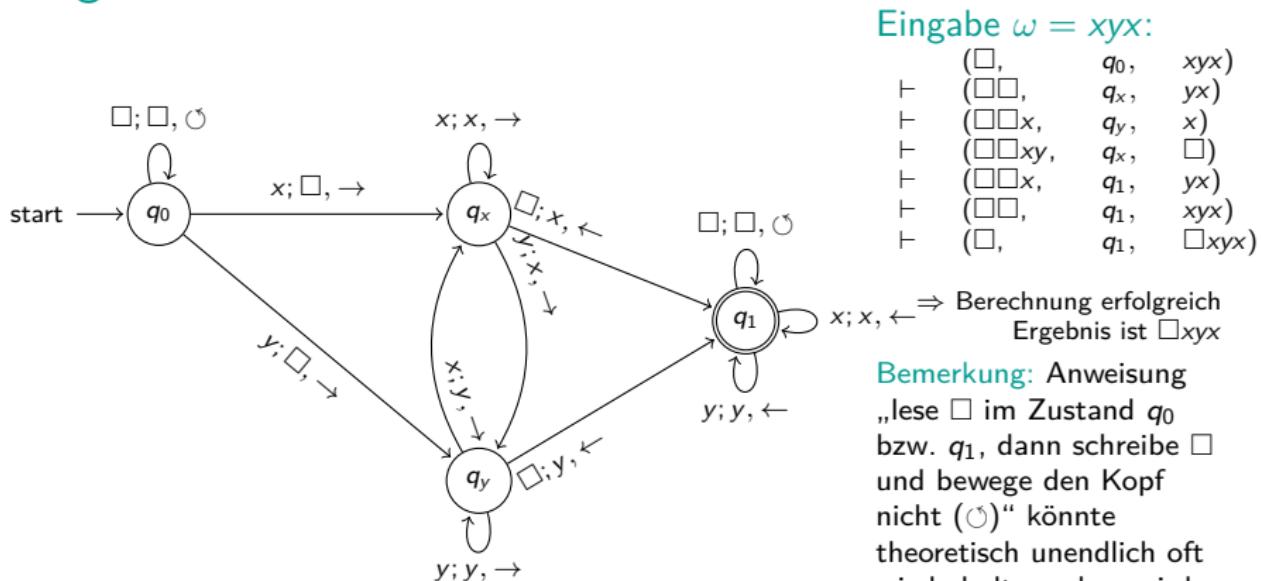


Bild: Erweitertes Zustandsübergangsdiagramm für T_V

Eingabe $\omega = xyx$:

(\square, \square)	$q_0,$	$xyx)$
$\vdash (\square\square,$	$q_x,$	$yx)$
$\vdash (\square\square x,$	$q_y,$	$x)$
$\vdash (\square\square xy,$	$q_x,$	$\square)$
$\vdash (\square\square x,$	$q_1,$	$yx)$
$\vdash (\square\square,$	$q_1,$	$xyx)$
$\vdash (\square,$	$q_1,$	$\square xyx)$

$x; x, \leftarrow \Rightarrow$ Berechnung erfolgreich
Ergebnis ist $\square xyx$

Bemerkung: Anweisung „lese \square im Zustand q_0 bzw. q_1 , dann schreibe \square und bewege den Kopf nicht (\circlearrowright)“ könnte theoretisch unendlich oft wiederholt werden, wird aber nur einmal durchgeführt.

Bemerkung: Die Funktionalität dieser TM ermöglicht einer TM mit linksseitig beschränktem Band die Funktionalität einer TM mit unbeschränktem Eingabeband.

Turing-berechenbare Funktionen: die Kopierfunktion I

- Gesucht: TM T_C welche jedes nichtleere Eingabewort über dem Alphabet $\Sigma = \{0, 1\}$ dupliziert.

Von T_C berechnete Funktion $f_C : f(\omega) = \omega\omega$.

Beispiele:

$$f_C(0) = 00, f_C(1) = 11, f_C(110) = 110110, f_C(0100) = 01000100$$

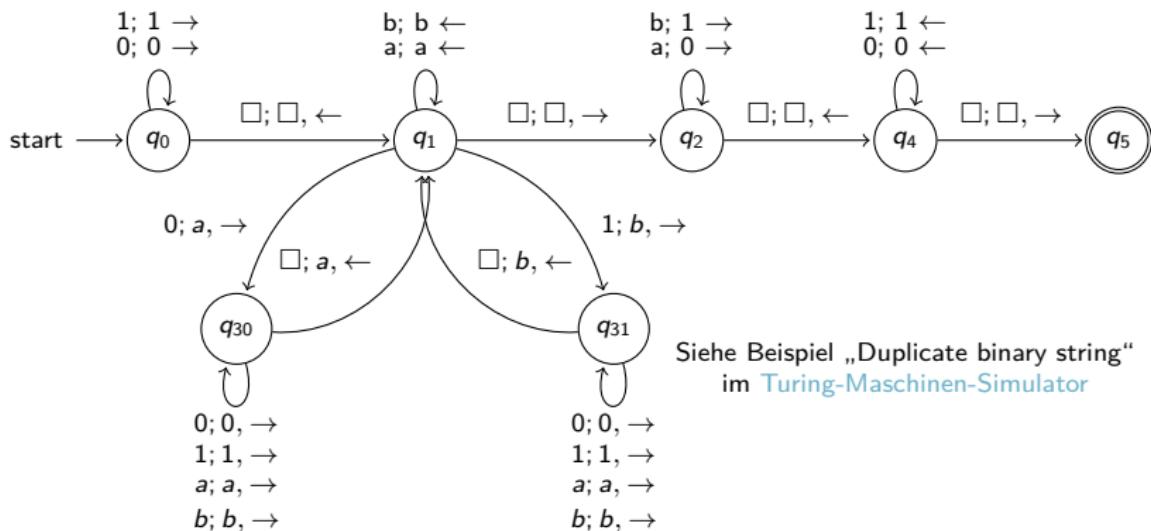
- Basisideen:

- ▶ Verwende andere Zeichen als 0/1 als Markierung bereits bearbeiteter Zeichen.
- ▶ Laufe einmal über das Wort nach rechts.
- ▶ Laufe nach links bis zur ersten gefundenen 0/1 und ersetze dies durch a/b (ignoriere hierbei as und bs).
Laufe nach rechts hänge das geschriebene Symbol (a/b) am Wortende an. Wiederhole diesen Schritt bis beim nach links laufen keine 0en oder 1en mehr gefunden werden.
- ▶ Laufe dann wieder von links nach rechts über das Wort und ersetze alle a/b durch 0/1.
- ▶ Gehe zum Wortanfang zurück und bleibe auf dem Beginn des doppelten Strings stehen.

Turing-berechenbare Funktionen: die Kopierfunktion II

$$T_C = (Q, \Sigma, \Pi, \delta, q_0, F) =$$

$(\{q_0, q_1, q_2, q_{30}, q_{31}, q_4, q_5\}, \{0, 1\}, \{0, 1, a, b, \square\}, \{q_5\}, \delta)$ mit δ siehe unten.



Siehe Beispiel „Duplicate binary string“
im [Turing-Maschinen-Simulator](#)

Bild: Erweitertes Zustandsübergangsdiagramm für T_C

Turing-berechenbare Funktionen: die Kopierfunktion II

Frage: Tut T_c wirklich das was sie soll? Also einen binären String duplizieren, die Funktion $f_C : f(\omega) = \omega\omega$ berechnen?

Antwort: Nein. T_c hängt an einen Binärstring dessen **Spiegelbild** (also den String rückwärts gelesen) an, berechnet also die Funktion $f_{CR} : f(\omega) = \omega\omega^R$.

Beispiele: $f_C(0) = 00, f_C(1) = 11, f_C(110) = 110011, f_C(0100) = 01000010$

Korrekte Lösung: siehe Tafel!!

Erinnerung: Vergleich Turing-Maschinen Modelle

Basismodell: DETMU - Deterministische Einband TM mit unendlich langem Band

- Beschränkung: ein Bandende fest
⇒ Keine funktionale Einschränkung: äquivalent zur DETMU
- Beschränkung: beide Bandenden fest ((N)LBA))
⇒ Funktionale Einschränkung: nicht äquivalent zur DETMU
- Erweiterung: mehrere Spuren
⇒ Keine funktionale Erweiterung: äquivalent zur DETMU
- Erweiterung: mehrere Bänder
⇒ Keine funktionale Erweiterung: äquivalent zur DETMU
- Erweiterung: Nichtdeterminismus (NTM)
⇒ Keine funktionale Erweiterung: äquivalent zur DETMU

Die Idee der universellen Turing-Maschine I

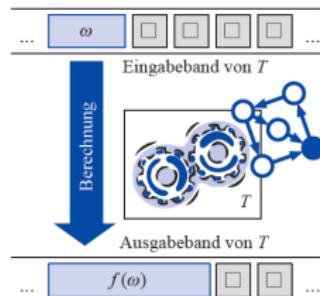


Bild: Turing-Maschine T [Hoffmann, 2011]

Beobachtung: Turing-Maschinen können viele Funktionen berechnen, aber jede TM T kann nur genau eine einzige Funktion berechnen.

Frage: Ist es möglich, eine **universelle** Turing-Maschine zu konstruieren, welche eine beliebige andere TM T als Eingabe erhält und die Ausgabe von T berechnet? So wie ein Computer viele Dinge erledigen kann, wenn er verschiedene Programme lädt?

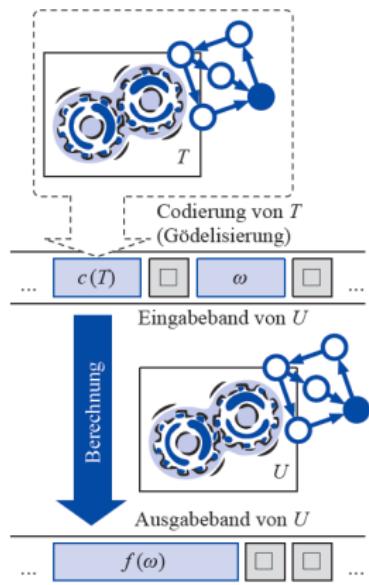


Bild: Universelle Turing-Maschine U [Hoffmann, 2011]

Die Idee der universellen Turing-Maschine II

Antwort: Ja, die Konstruktion einer universellen TM ist möglich, aber nur, wenn man die zu simulierenden TM eindeutig codieren kann.

Lösung: Gödelisierung einer TM $T \in \mathbb{T}$ (mit der Menge aller Turingmaschinen \mathbb{T}) durch Funktion $c : \mathbb{T} \rightarrow \mathbb{N}_0$ mit

- c ist injektiv: $T_1 \neq T_2 \Rightarrow c(T_1) \neq c(T_2)$
- für jedes $x \in c(\mathbb{T})$ ist entscheidbar, ob x die Codierung einer TM ist,
- $c : \mathbb{T} \rightarrow \mathbb{N}_0$ und $c^{-1} : c(\mathbb{T}) \rightarrow \mathbb{T}$ sind berechenbar.

Bemerkungen:

- Für eine Gödelisierungsfunktion c heißt $c(T)$ die **Gödelnummer** der TM $T \in \mathbb{T}$.
- Es gibt mehrere mögliche Gödelisierungsfunktionen über \mathbb{T} .
- Ein möglicher Ansatz: Bringe alle TMs in die **normalisierte Form**

$$\begin{aligned} T &= (Q, \Sigma, \Pi, \delta, q_1, F) \\ &= (\{q_1, q_2, \dots, q_n\}, \{0, 1\}, \{0, 1, \square\}, \delta, q_1, \{q_2\}) \end{aligned}$$

Beispiel: Die Gödelisierung von T_1

Die Funktion $f(n) = n + 1$ wird berechnet durch die normalisierte $T_{n1} = (Q, \Sigma, \Pi, \delta, q_1, F)$ mit

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{1\}$
- $\Pi = \{1, \square\}$
- $F = \{q_2\}$

δ	1	\square
q_1	$(q_1, 1, \rightarrow)$	$(q_2, 1, \leftarrow)$
q_3	$(q_3, 1, \leftarrow)$	$(q_2, \square, \rightarrow)$
q_2	-	-

Beobachtung: In normalisierter Form stecken die wesentlichen Informationen nur in den Funktionswerten von δ .

Symbol	Code	Symbol	Code
q_1	1	\leftarrow	1
q_2	11	\rightarrow	11
q_3	111	(0
0	1	,	0
1	11)	0
\square	111	\mapsto	0

Tabelle: Codierungsschema für Funktionswerte von δ

$$(q_1, 1) \mapsto (q_1, 1, \rightarrow)$$

↓

0101100010110110

$$(q_1, \square) \mapsto (q_3, 1, \leftarrow)$$

↓

010111000111011010

...

Simulation einer beliebigen TM durch eine universelle TM

1. Berechne Gödelisierung von T .

Z.B. $c(T_{n1}) = 010110001...$
 $011011001011100011011...$
 $010011010001101101001...$
 1011100011101110110

2. Schreibe $c(T)$ auf das **Codierungsband** (CB) von U_3 (universelle 3-Band TM).
3. Schreibe Eingabewort ω für T auf das **Arbeitsband** (AB) von U_3 .
4. Schreibe Startzustand von T auf das **Zustandsband** (ZB) von U_3 .
5. U_3 simuliert die Berechnung von T für ω durch Auffinden und Ausführen der korrekten Anweisung (CB) für aktuellen Zustand (ZB) und Zeichen (AB) von T .

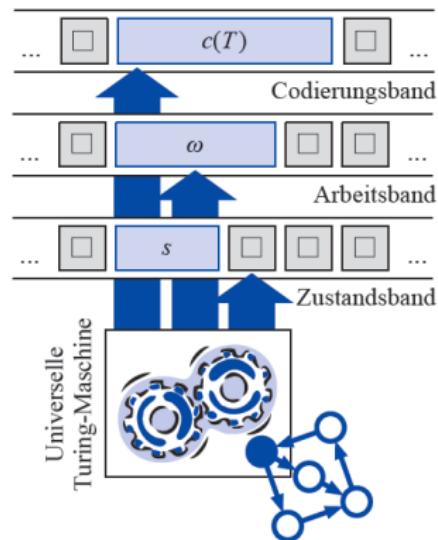


Bild: Simulation einer TM T durch die universelle 3-Band-TM U_3 [Hoffmann, 2011]

Wie komplex muss eine universelle Turing-Maschine sein?

- Turing, 1936: Beschreibung auf 4 Paperseiten [[Turing, 1936](#)]
- Minsky, 1962: universelle TM mit 7 Zuständen und 4 Farben
- Wolfram, 2002: universelle TM mit 2 Zuständen und 5 Farben
- Wolfram, 2002, Beweis durch Smith 2003: **kleinstmögliche**
universelle TM hat 2 Zustände und 3 Farben (Bandsymbole)

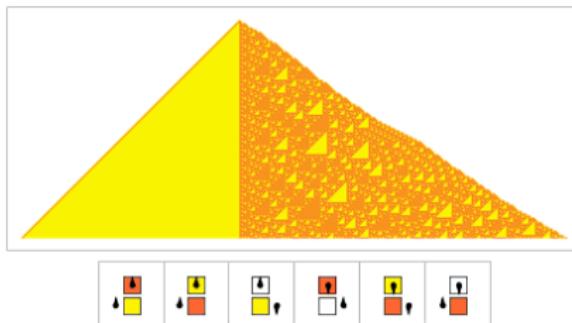


Bild: Die kleinste mögliche universelle TM und ein beispielhaftes Arbeitsergebnis (Darstellung der Veränderung des Bandinhaltes bei Start mit leerem Band von oben nach unten) [Quelle: blog.wolfram.com](#)

Arbeitsweise dieser Maschine: [wolframscience.com](#)

Abschnitt 2

Entscheidbarkeit und Unentscheidbarkeit

Entscheidbarkeit und Berechenbarkeit

Erinnerung:

- Eine **Funktion** heißt **berechenbar**, wenn es einen Algorithmus (z.B. eine Turing-Maschine) gibt, der sie berechnet.
- Alle von einem Menschen berechenbaren Funktionen sind auch von einem Algorithmus (z.B. einer Turing-Maschine) berechenbar (Church-Turing-These).

Übertragung auf formale Sprachen:

- Eine **formale Sprache** heißt **entscheidbar**, wenn es einen Algorithmus (z.B. eine Turing-Maschine) gibt, der für jedes Wort ω aus der Sprache angeben kann, ob das Wort in der Sprache enthalten ist, oder nicht (z.B. endet eine TM für ω in einem Final- oder einem Nicht-Final-Zustand).
- **Frage:** Sind alle formalen Sprachen entscheidbar?
Antwort: Nein. Beispiel: $L_\pi = \{\omega \in \{0, 1, \dots, 9\}^* \mid \text{die Ziffernfolge } \omega \text{ kommt in den Nachkommastellen von } \pi \text{ vor}\}$ ist nicht entscheidbar. Allerdings ist L_π semi-entscheidbar, Details folgen.

Erinnerung: Akzeptierende Turing-Maschinen

Definition

Eine Turing-Maschine bzw. ein **Turing-Akzeptor** $T = (Q, \Sigma, \Pi, \delta, q_0, F)$ **akzeptiert** das Wort $\omega \in \Sigma^*$, falls sie unter Eingabe von ω in einem Endzustand terminiert. Die **von T akzeptierte Sprache** $\mathcal{L}(T)$ ist

$$\begin{aligned}\mathcal{L}(T) &= \{\omega \in \Sigma^* \mid T \text{ akzeptiert } \omega\} \\ &= \{\omega \in \Sigma^* \mid \text{mit der Eingabe } \omega \text{ beendet } T \text{ die Bearbeitung} \\ &\quad \text{nach endlich vielen Schritten in einem Finalzustand } q_f\} \\ &= \{\omega \in \Sigma^* \mid (\square, q_0, \omega) \vdash^* (\alpha, q_f, \beta), q_f \in F, \alpha, \beta \in \Pi^*\}.\end{aligned}$$

Bemerkungen:

- Falls T ω nicht akzeptiert, wird die Berechnung entweder in einem nicht-finalen Zustand $q_x \in Q \setminus F$ beendet, oder T gerät in eine Endlossschleife.
- Der Inhalt des Bandes entscheidet nicht über Akzeptanz, links und rechts des Kopfes können beliebige Wörter $\alpha, \beta \in \Pi^*$ stehen.

Entscheidbarkeit und Semi-Entscheidbarkeit

Definition

Eine Sprache L heißt

- **entscheidbar** oder **rekursiv**, falls eine Turing-Maschine T existiert, die L akzeptiert und zusätzlich für jede Eingabe terminiert.
- **semi-entscheidbar** oder **rekursiv aufzählbar**, falls eine Turing-Maschine T existiert, die L akzeptiert.

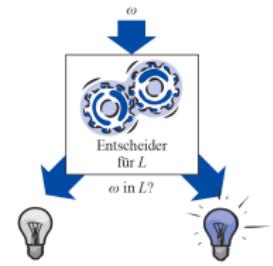


Bild: Entscheidbarkeit
[Hoffmann, 2011]

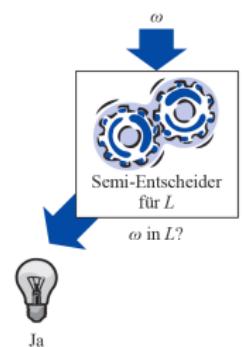


Bild: Semi-Entscheidbarkeit
[Hoffmann, 2011]

Verdeutlichung: (für eine formale Sprache $L \subseteq \Sigma^*$)

- **Entscheidbarkeit:** Es existiert ein Verfahren (z.B. eine TM) die für jedes $\omega \in \Sigma^*$ nach endlicher Zeit sagt „ $\omega \in L$ “ oder „ $\omega \notin L$ “.
- **Semi-Entscheidbarkeit:** Es existiert ein Verfahren (z.B. eine TM) die für jedes $\omega \in \Sigma^*$ nach endlicher Zeit sagt „ $\omega \in L$ “. Falls „ $\omega \notin L$ “ gibt sie keine Antwort.

Vokabular: (Semi-|Un)?entscheidbar, rekursiv (aufzählbar)?

Eine Sprache $L \subseteq \Sigma^*$ heißt

entscheidbar falls eine TM T mit $\mathcal{L}(T)$ existiert, die für jedes Eingabewort $\omega \in \Sigma^*$ anhält.

semi-entscheidbar falls eine TM T mit $\mathcal{L}(T)$ existiert, die (nur für $\omega \in L$ aber) **nicht für jedes beliebige** Eingabewort $\omega \in \Sigma^*$ anhält.

unentscheidbar falls sie nicht entscheidbar ist. Achtung: unentscheidbare Sprachen können semi-entscheidbar sein!

rekursiv aufzählbar falls es eine surjektive und (z.B. von einer TM) berechenbare Funktion $f : \mathbb{N} \rightarrow L$ gibt, die L aufzählt:
$$L = \{f(1), f(2), f(3), \dots\}$$

rekursiv falls eine TM T mit $\mathcal{L}(T)$ existiert, die für jedes Eingabewort $\omega \in \Sigma^*$ anhält.

Bemerkungen:

- Jede entscheidbare Sprache ist auch semi-entscheidbar.
- Jede rekursive Sprache ist auch rekursiv-aufzählbar.

Beispiele zum Mitdenken

- $L_{C3} = \{(ab)^n \mid n \in \mathbb{N}\}$ ist rekursiv und entscheidbar
- $L_{C2} = \{a^n b^n \mid n \in \mathbb{N}\}$ ist rekursiv und entscheidbar
- $L_{C1} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ ist rekursiv und entscheidbar
- $L_H = \{(c(T), \omega) \mid \text{die TM } T \text{ hält bei Eingabe von } \omega \text{ an}\}$ ist rekursiv-aufzählbar und semi-entscheidbar, aber nicht entscheidbar
- $L_D = \{c(T) \mid \text{die TM } T \text{ hält bei Eingabe von } c(T) \text{ nicht an}\}$ ist weder rekursiv-aufzählbar noch rekursiv

Bemerkung:

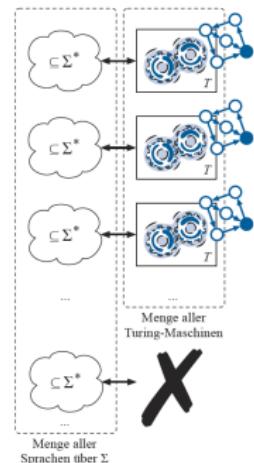
- Alle Sprachen, die von einem endlichen Automaten, einem Kellerautomaten oder einer linear beschränkten Turing-Maschine akzeptiert werden, sind **rekursiv und entscheidbar**.
- Alle Sprachen der Chomsky-Klassen 3,2,1 sind **rekursiv und entscheidbar**.
- Alle echten Chomsky-Typ 0 Sprachen sind nur **rekursiv aufzählbar und semi-entscheidbar**.

Existenz unentscheidbarer Probleme

Frage: Gibt es Probleme die unentscheidbar sind? Fragestellungen von denen ein Computer nicht sicher sagen kann „wahr“ oder „falsch“?

Antwort: Ja. Beweisidee:

- Alle Problem die ein moderner Computer lösen kann, können auch von einer TM entschieden werden.
- Für jedes entscheidbare Problem $P \subseteq \Sigma^*$ (Σ beliebig) existiert eine TM T_P , die P entscheidet.
- Die Menge aller Turing-Maschinen ist **abzählbar unendlich** (siehe Idee der Gödelisierung).
- Die Menge aller Sprachen über Σ^* ist jedoch **überabzählbar unendlich** (Da dies die Potenzmenge der abzählbar unendlichen Menge Σ^* ist).
- Es existieren also schlicht nicht genug Turing-Maschinen, um alle Probleme zu entscheiden, also muss es auch unentscheidbare Probleme geben.



Quelle: [Hoffmann, 2011]

Entscheidbare und unentscheidbare Probleme I

Bemerkung: Problem „Erfüllt ω Eigenschaft E ?“ ist genau dann entscheidbar, wenn Sprache $L_E = \{\omega \in \Sigma^* \mid \omega \text{ erfüllt Eigenschaft } E\}$ entscheidbar ist.

Beispiele:

- Das Problem „Ist $p \in \mathbb{N}$ eine Primzahl?“ ist **entscheidbar**.
 - ▶ Baue $L_p = \{p \in \{0, 1\}^* \mid p \text{ ist Binärdarstellung einer Primzahl}\}$
 - ▶ L_p ist entscheidbar, da TM T_p existiert die immer terminiert und jedes $p \in \{0, 1\}^*$ akzeptiert oder ablehnt.
 - ▶ Idee: Sieb des Eratosthenes
- Das Problem „Kommt $x \in \mathbb{N}$ in den Nachkommastellen von π vor?“ ist **unentscheidbar (aber semi-entscheidbar)**.
 - ▶ Konstruiere $L_\pi = \{\omega \in \{0, 1, \dots, 9\}^* \mid \text{die Ziffernfolge } \omega \text{ kommt in den Nachkommastellen von } \pi \text{ vor}\}$
 - ▶ Offensichtlich kann TM T_π konstruiert werden, die 1, 14, 141, 1415, ... akzeptiert.
 - ▶ Problem für größere Zahlen x : T_π muss weitere Nachkommastellen von π approximieren und mit x vergleichen. Bei Übereinstimmung wird x akzeptiert; sonst rechnet T_π unendlich lange weiter.

Entscheidbare und unentscheidbare Probleme II

Beispiele:

- Das **Halteproblem** „hält die TM T bei Eingabe ω ?“ ist **unentscheidbar (aber semi-entscheidbar)**.
 - ▶ Konstruiere $L_H = \{(c(T), \omega) \mid \text{die TM } T \text{ die durch } c(t) \text{ codiert wird, hält bei Eingabe } \omega \text{ an}\}$
 - ▶ Konstruiere universelle TM U und starte Berechnung mit $c(T)$ und ω (simuliere T unter der Eingabe von w).
 - ▶ U hält genau dann mit einem positiven Ergebnis an, wenn T akzeptiert und anhält.
 - ▶ Hält T für ω nicht an, kann U dies nicht zu erkennen und „nein“ antworten; stattdessen wartet U unendlich lange, ob T anhält und hält also auch nicht an.
- Das **Diagonalproblem** „hält die TM T bei Eingabe ω **nicht** an?“ ist **unentscheidbar (und nicht semi-entscheidbar)**.
 - ▶ Konstruiere $L_D = \{c(T) \mid \text{die TM } T \text{ die durch } c(t) \text{ codiert wird, hält bei Eingabe von } c(T) \text{ nicht an}\}$
 - ▶ Konstruiere universelle TM U_2 , die L_D semi-entscheidet (die immer dann anhält, wenn $c(T)$ nicht anhalten würde) \Rightarrow Widerspruch.

Eine unberechenbare Funktion

Die **fleißiger-Biber Funktion** (busy beaver function, Radó-Funktion) $\Sigma(n)$ ist **nicht berechenbar**. (Ein entsprechend konstruiertes Entscheidungs-Problem ist unentscheidbar und nicht semi-entscheidbar.)

- **Fleißige Biber der Größe n** ($BB(n)$) sind TM mit n Zuständen (plus einem Finalzustand) die möglichst viele Einsen auf ein mit Nullen vorinitialisiertes Band schreiben und nach endlich vielen Schritten terminieren.
- $\Sigma(n)$ gibt an, wie viele Einsen ein $BB(n)$ höchstens schreiben kann.
- Da für jedes n nur endlich mögliche fleißige Biber existieren, ist $\Sigma(n)$ für alle n wohldefiniert, aber nur bis $n = 4$ exakt bekannt. Die genaue Charakterisierung von $\Sigma(n)$ ist aktuelles Forschungsthema
(**Scott Aaronson: The Busy Beaver Frontier**)

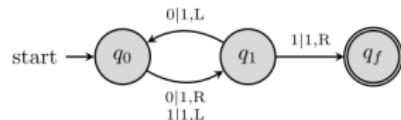


Bild: Fleißiger Biber mit 2 Zuständen
Quelle:[Wikipedia](#)

n	$BB(n)$	$\Sigma(n)$	Reference
1	1	1	Trivial
2	6	4	Lin 1963 (see [9])
3	21	6	Lin 1963 (see [9])
4	107	13	Brady 1983 [2]
5	$\geq 47,176,870$	$\geq 4,098$	Marcus und Baumrock 1990 [10]
6	$> 7.4 \times 10^{36331}$	$> 3.5 \times 10^{18,267}$	Kropitz 2010 [8]
7	$> 10^{2 \times 10^{1278,707,355}}$	$> 10^{10^{10^{1278,707,355}}}$	"Pythagoras" 2014 (see [12, Section 4.6])

Quelle: Quelle:www.scottaaronson.com

Berühmtes unentscheidbares Problem: Das Halteproblem

Allgemeines Halteproblem

Gegeben: Turing-Maschine T und Eingabewort ω .

Frage: Terminiert T unter Eingabe von ω ?

Satz (Turing, 1936)

Das allgemeine Halteproblem ist unentscheidbar.

Bemerkungen:

- Beweis per Widerspruch, siehe [[Hoffmann, 2011](#)].
- Bedeutung für den Informatik-Alltag:
 - ▶ Es wird nie einen Algorithmus geben, der überprüfen kann, ob ein geschriebenes Programm in eine Endlosschleife läuft.
 - ▶ Ein Compiler wird Ihnen also nie sagen können, ob Ihr Programm tatsächlich ein korrektes Ergebnis liefert, oder ob es sich vielleicht manchmal aufhängt.

Berühmtes unentscheidbares Problem: Das Eigenschaftsproblem

Eigenschaftsproblem

Sei E eine nichttriviale funktionale Eigenschaft von Turing-Maschinen.

Gegeben: Turing-Maschine T , Eigenschaft E .

Frage: Besitzt T die Eigenschaft E ?

Bemerkungen: Eine Eigenschaft E von Turing-Maschinen ist

- **nichttrivial**, falls es mindestens eine TM T_E gibt, welche E hat und mindestens eine TM $T_{\neg E}$ gibt, auf welche E nicht zutrifft.
- **funktional**, falls sie die von einer TM berechnete Funktion beschreibt.

Beispiele nichttrivialer funktionaler Eigenschaften einer TM T :

- T berechnet eine konstante Funktion
- Alle Ausgaben von T sind mindestens n Zeichen lang
- T sortiert eine gegebene Buchstabenreihenfolge
- T berechnet die Nachfolgerfunktion

Berühmtes unentscheidbares Problem: Das Eigenschaftsproblem

Satz (Rice, 1953)

Das Eigenschaftsproblem ist unentscheidbar.

Bemerkungen:

- Beweis per Konstruktion einer TM die das Halteproblem löst aus TMs die das Entscheidungsproblem lösen, siehe [Hoffmann, 2011].
- Bedeutung für den Informatik-Alltag:
 - ▶ Es wird nie einen Algorithmus geben, der für ein beliebiges Programm maschinell verifiziert, ob es sich entsprechend seiner Spezifikation verhält.
 - ▶ Zu beachten: Softwareverifikation liefert gute Verfahren, um für *die meisten* Programme formal zu beweisen, dass sie bestimmte Eigenschaften haben. Diese Verfahren funktionieren aber nicht für *alle* beliebigen Programme.

Weitere unentscheidbare Probleme

Frage: Wieso war und ist es wichtig, unentscheidbare Probleme zu finden?

Antwort: Je mehr unentscheidbare Probleme man kennt, desto einfacher ist es, ein unbekanntes Problem als (un)entscheidbar zu klassifizieren.

- Ist ein Problem A unentscheidbar und kann man es auf ein Problem B reduzieren, weiß man, dass auch B unentscheidbar ist.
- Ist B entscheidbar, und kann man A auf B reduzieren, weiß man, dass auch A entscheidbar ist.
- A kann man auf B **reduzieren**, falls es eine totale, berechenbare Funktion f gibt mit $x \in A \Leftrightarrow f(x) \in B$.

Beispiele:

- Postsches Korrespondenzproblem (PCP): Lassen sich die ersten bzw. zweiten Komponenten von Wortpaaren so kombinieren, dass die selben Wörter entstehen?
- Entscheidungsprobleme über formalen Sprachen (z.B. Leerheits-, Endlichkeitsproblem für Typ-1-Sprachen)

- Alphabet
 $\Sigma := \{0, 1\}$
- Wortpaare

(01 , 1)
(0 , 000)
(01000 , 01)

- Lösung
- | | | | |
|-------|-----|-----|----|
| 01000 | 0 | 0 | 01 |
| 01 | 000 | 000 | 1 |

Bild: PCP-Instanz
[Hoffmann, 2011]

In a Nutshell: Formale Sprachen und Entscheidbarkeit

Typ3-, Typ2-, Typ1-Sprachen sind entscheidbar. Es existiert für jede

Typ1-Sprache L z.B. eine TM, die für jedes Wort ω definitiv angeben kann, ob $\omega \in L$ oder $\omega \notin L$ gilt.

Begründung: Da L von einer Grammatik mit nicht-verkürzenden Regeln erzeugt wird, kann eine TM alle Ableitung aus dem Startsymbol nachvollziehen und stoppen falls ω erzeugt wurde (Antwort: $\omega \in L$), die Ableitung zu lang wird oder ein schon einmal abgeleitetes Wort gefunden wurde (Antwort: $\omega \notin L$).

Typ0-Sprachen sind semi-entscheidbar. Es existiert für jede Typ0-Sprache L z.B. eine TM, die für jedes Wort $\omega \in L$ dies nach endlicher Zeit bestätigen kann. Falls $\omega \notin L$ gibt es keine Antwort.

Insgesamt gilt also folgende **echte** Inklusionskette:

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_{\text{entscheidbar}} \subset \mathcal{L}_0 = \mathcal{L}_{\text{semi-entscheidbar}}.$$

Wobei \mathcal{L}_x für die Klasse der Sprachen vom Chomsky-Typ x steht,
 $\mathcal{L}_{(\text{semi-})\text{entscheidbar}}$ für die Klasse der (semi-)entscheidbaren Sprachen.

Abschnitt 3

Komplexität

Motivation

Done: Ist ein Problem lösbar bzw. berechenbar bzw. entscheidbar?

TODO: In welcher Zeit kann ich das Problem lösen und wie viel Arbeitsspeicher brauche ich dafür?

Wieso ist das relevant?

- Das Problem „Ist p in Primfaktoren zerlegbar?“ ist entscheidbar.
- Allerdings ist der Rechenaufwand schon für Primzahlen mit mehr als 100 Stellen nicht mehr vertretbar.
- Dies ist einerseits theoretisch bewiesen, aber nur gültig, wenn $P \neq NP$. (Später mehr hierzu)
- ⇒ Grundlage von RSA

Ziel: Bestimmung der **Laufzeit und des Speicherplatzbedarfs** eines Algorithmus unabhängig von Programmiersprache, Betriebssystem, Prozessorleistung, Arbeitsspeicher, Computerarchitektur.

Lösung: Keine Angabe von Sekunden oder Megabyte, sondern abstrakte, ungefähre Maße.

Bestimmung von Zeit- und Raumkomplexität I

Idee: Verwende **deterministische (TM)** bzw. **nichtdeterministische (NTM)** Mehrband-Turing-Maschine T und bestimme

Zeitkomplexität als Anzahl der Schritte, welche T zur Beantwortung der Frage mindestens ausführen muss,

Raumkomplexität als Anzahl der Bandzellen, die T zusätzlich zu den Zellen, in welchen das Eingabewort steht, zur Beantwortung der Frage mindestens benötigt.

Problem: Turing-Maschinen sind sehr ineffizient im Vergleich zu moderner Hardware - ist das überhaupt sinnvoll?

Antwort: „JA!“ (Quantencomputer: „Vielleicht“.)

Erweiterte Church-Turing-These

Jeder Algorithmus der in einem beliebigen Berechnungsformalismus gegeben ist, lässt sich mit nur polynomialen Mehraufwand in das Modell der Turingmaschine umformen. Der entsprechende Zeitaufwand vergrößert sich also nur um einen polynomialen Faktor.

Bestimmung von Zeit- und Raumkomplexität II

Definition

Sei T eine akzeptierende Mehrband-Turing-Maschine (TM oder NTM). Die **Laufzeit-** bzw. **Bandplatzfunktion** $t_T, s_T : \Sigma^* \rightarrow \mathbb{N}_0$ sind **nur definiert**, für alle Worte aus Σ^* , die von T akzeptiert werden (ansonsten undefiniert):

- $t_T(\omega) = \min\{n \mid T \text{ akzeptiert } \omega \text{ nach } n \text{ Schritten}\}$
- $s_T(\omega) = \min\{n \mid T \text{ akzeptiert } \omega \text{ mit } n \text{ zusätzlichen Bandzellen}\}$

Bemerkungen:

- Die Laufzeit- und Bandplatzfunktion geben an, in welcher Zeit bzw. mit welchem Platzbedarf T das Wort ω **im Optimalfall** akzeptiert.
- Verwendung von Mehrband-Turing-Maschinen, um modernen Computerarchitekturen ähnlicher zu werden und einfache Berechnungen wie z.B. Multiplikationen schneller durchführen zu können.

TIME, NTIME, SPACE und NSPACE

Falls für eine Sprache L

- Mehrband-Turing-Maschinen T_1, T_2 existieren ($T_1 = T_2$ kann gelten, muss aber nicht sein), die L akzeptieren
- eine Funktion $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ existiert, so dass alle Wörter der Länge n von T_1 in höchsten $f(n)$ Schritten akzeptiert werden:
 $\forall_{\omega \in L} t_{T_1}(\omega) \leq f(|\omega|)$
- eine Funktion $g : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ existiert, so dass alle Wörter der Länge n von T_2 mit einem zusätzlichen Platzbedarf von höchsten $g(n)$ Bandzellen akzeptiert werden: $\forall_{\omega \in L} s_{T_2}(\omega) \leq g(|\omega|)$

dann liegt L in

- **TIME($f(n)$)**, falls T_1 eine deterministische TM ist,
- **NTIME($f(n)$)**, falls T_1 eine nichtdeterministische TM ist.
- **SPACE($g(n)$)**, falls T_2 eine deterministische TM ist,
- **NSPACE($g(n)$)**, falls T_2 eine nichtdeterministische TM ist

O-Notation

Problem: Es ist schwierig, die für die Angabe der Raum- und Zeitkomplexität als Obergrenze benötigten Funktionen genau zu treffen.

Lösung: Betrachte **Äquivalenzklassen** von Funktionen, um „ähnlich“ schnelle bzw. schlanke Algorithmen erkennen zu können.

Definition

Die Funktion $f : \mathbb{N} \rightarrow \mathbb{R}^+$ fällt in die Komplexitätsklasse

$O(g(n))$, Schreibweise: $f(n) = O(g(n))$, $f(n) \in O(g(n))$,

falls eine Konstante $c \in \mathbb{R}^+$ und ein $n_0 \in \mathbb{N}_0$ existieren, so dass

$f(n) \leq c \cdot g(n)$, $g : \mathbb{N}_0 \rightarrow \mathbb{R}^+$,
für alle $n \geq n_0$ gilt.

Bemerkungen:

- n steht für die Größe des zu verarbeitenden Problems.
- Die Bedingung „ $n \geq n_0$ “ bedeutet, dass initiale „Warmlaufphasen“ ignoriert werden und nur das Verhalten für sehr große n interessiert.
- $f(n) = O(g(n))$ heißt „ f wächst höchstens so schnell wie g .“

Beispiele für die O-Notation

- $n^2 \in O(n^2)$
- $2n^2 \in O(n^2)$
- $\frac{1}{2}n^2 \in O(n^2)$
- $n \in O(n^2)$
- $n + n^2 \in O(n^2)$
- $n^2 + n^3 \in O(n^3)$
- $6n^4 + 3n^2 + 1 \in O(n^4)$
- $e^n + n^5 \in O(e^n)$

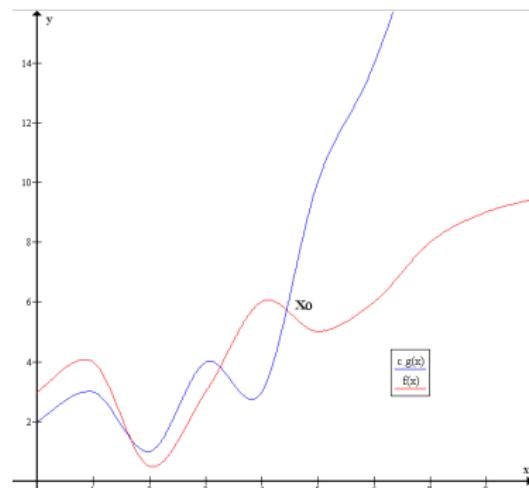


Bild: $f(n) \in O(g(n))$ Quelle: Wikipedia

Folgerung: Mit Hilfe der O-Notation kann das asymptotische Wachstum einer Funktion f nach **oben** abgeschätzt werden.

Exkurs: Weitere Notationen

Die O-Nation ist nur eines der sogenannten **Landau-Symbole**. Weitere Symbole existieren, mit welchen eine Funktion (mit mehr Aufwand!) nach unten, bzw. genauer abgeschätzt werden können:

Obere asymptotische Schranke (f wächst höchstens so schnell wie g)		
$f(n) = \mathcal{O}(g(n))$	$\exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 f(n) \leq c \cdot g(n)$	$0 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$
Untere asymptotische Schranke (f wächst mindestens so schnell wie g)		
$f(n) = \Omega(g(n))$	$\exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 f(n) \geq c \cdot g(n)$	$0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty$
Exakte asymptotische Schranke (f wächst genauso schnell wie g)		
$f(n) = \Theta(g(n))$	$\exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$
Starke obere asymptotische Schranke (f wächst langsamer als g)		
$f(n) = o(g(n))$	$\forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 c \cdot f(n) \leq g(n)$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
Starke untere asymptotische Schranke (f wächst schneller als g)		
$f(n) = \omega(g(n))$	$\forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 c \cdot f(n) \geq g(n)$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$

Quelle: [Hoffmann, 2011]

Problemkomplexität

Definition

Sei P ein beliebiges Ja-Nein-Problem. Es gilt:

- $P \in (N)TIME(O(f(n)))$, falls für eine Mehrband (N)TM T gilt:
 T entscheidet P und die Anzahl für alle Eingaben der Länge n benötigten Schritt ist in $O(f(n))$: $\forall_{\omega \in L_P} t_T(\omega) \in O(f(|\omega|))$
- $P \in (N)SPACE(O(g(n)))$, falls für eine Mehrband (N)TM T gilt:
 T entscheidet P und die Anzahl der für alle Eingaben der Länge n zusätzlich benötigten Bandzellen ist in $O(g(n))$:
 $\forall_{\omega \in L_P} s_T(\omega) \in O(g(|\omega|))$

Bemerkungen:

- „Erfüllt ω Problem P ?“ wird genau dann von einer (N)TM T entschieden, wenn die Sprache $L_P = \{\omega \in \Sigma^* \mid \omega \text{ erfüllt Problem } P\}$ von T entschieden wird (T akzeptiert L_P und hält immer an).
- **Achtung:** Auch Probleme in $TIME(O(2^n))$ können eine effiziente Lösung haben, $O(2^n)$ für die Laufzeit ist nur Worst-Case Schätzung.

Hierarchie der Wachstumsfunktionen

Wachstumsfunktion	
$f_0(n)$	$= O(1)$
$f_1(n)$	$= O(\log \log n)$
$f_2(n)$	$= O(\log \log n)$
$f_3(n)$	$= O(\sqrt{\log n})$
$f_4(n)$	$= O(\log n)$
$f_5(n)$	$= O((\log n)^2)$
$f_6(n)$	$= O(\sqrt[3]{n})$
$f_7(n)$	$= O(n)$
$f_8(n)$	$= O(n \log n)$
$f_9(n)$	$= O(\sqrt{n^3})$
$f_{10}(n)$	$= O(n^2)$
$f_{11}(n)$	$= O(n^3)$
$f_{12}(n)$	$= O(n^{\log n})$
$f_{13}(n)$	$= O(2^{\sqrt{n}})$
$f_{14}(n)$	$= O(2^n)$
$f_{15}(n)$	$= O(e^n)$
$f_{16}(n)$	$= O(n!)$
$f_{17}(n)$	$= O(n^n)$
$f_{18}(n)$	$= O(2^{n^2})$
$f_{19}(n)$	$= O(2^{2^n})$

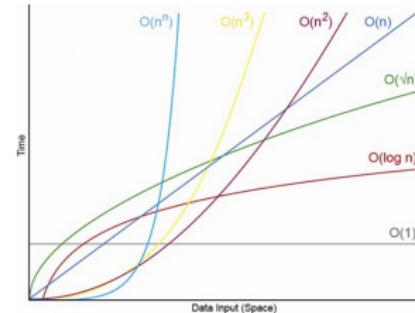


Bild: Vergleich von Wachstumsfunktionen Quelle: The Illustrated Primer

Beobachtung:

- Schon wenige Wachstumsfunktionen reichen, um die Effizienz von Algorithmen, bzw. die Schwere von Problemen eindeutig zu unterscheiden.
- Die Hierarchie der Wachstumsfunktionen gilt nur für große Probleme (Wichtig für Skalierbarkeit), siehe Bild für kleine Probleme.

Bild: Wachstumsfunktionen steigender Komplexität [Hoffmann, 2011]

Wichtige Komplexitätsklassen I

$O(1)$ Konstanter Ressourcenverbrauch

- Effizienz des Algorithmus unabhängig von der Problemgröße
- Hauptsächlich relevant für Speicherplatzverbrauch
- Beispiel (Zeit): Average Case Zugriff auf eine Hashtabelle
- Beispiel (Speicherplatz): Binäre Suche, Bubblesort

$O(\log n)$ Logarithmisches Wachstum

- Wachstumsklasse der meisten Divide-and-Conquer Algorithmen
- Problemgröße wird in jedem Schritt halbiert
- Beispiele (Zeit): Einfügen in sortierte Liste, binäre Suche

$O(n)$ Lineares Wachstum

- Häufige Wachstumsklasse für unoptimierte Verfahren
- Laufzeit, bzw. Speicherplatz wachsen proportional mit der Problemgröße
- Beispiele (Zeit): Suchen in einer unsortierten Liste, Worst Case Zugriff auf eine Hashtabelle

Wichtige Komplexitätsklassen II

$O(n \log n)$ Linear-logarithmisches Wachstum

- In der Praxis häufig auftretende Wachstumsklasse
- Beispiele (Zeit): Average Case Quicksort, Bestimmung der kürzesten Route

$O(n^k)$ Polynomielles Wachstum

- Tritt auf, wenn für jedes Element Schleifen über alle anderen Elementen durchlaufen werden müssen
- Obergrenze für sinnvoll implementierbare Algorithmen
- Beispiele (Zeit): Multiplikation von Matrizen, Bubblesort, Worst Case Quicksort

$O(k^n)$ Exponentielles Wachstum

- Laufzeit bzw. Speicherplatzbedarf wachsen schneller als jedes Polynom - Vervielfachung für jedes zusätzliche Element
- Schon für kleine Problemgrößen praktisch nicht mehr einsetzbar
- Beispiel: Traveling Salesperson (Zeit)

Grafischer Vergleich der Wachstumsklassen

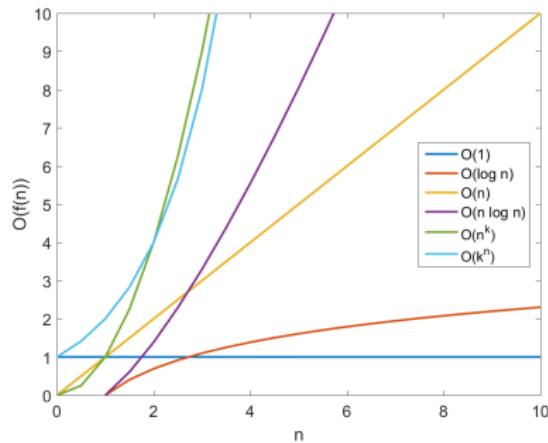


Bild: Vergleich Wachstumsklassen, $f(n) \leq 10$, $k = 2$

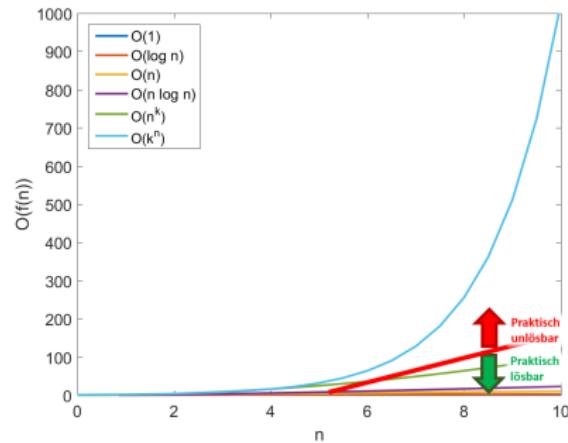


Bild: Vergleich Wachstumsklassen, $f(n) \leq 1000$, $k = 2$

Schlussfolgerung:

- Ein Algorithmus der auf kleinen Problemen effektiv läuft, **skaliert** nicht unbedingt gut für große Probleme.
- Oft ist ein schnell programmiertes Algorithmus mit $O(n^2)$ besser geeignet, als ein in $O(n \log n)$ laufender, aber schwer zu schreibender.

Vergleich (In)effizienter Algorithmen [Schöning, 2008]

- $A_1 \in \text{TIME}(O(n))$ Finde einen Weg zwischen zwei Knoten in einem Graphen der Größe n (Kürzester Weg).
- $A_2 \in \text{TIME}(O(n^3))$ Erzeuge aus einem Programm mit n Token einen Syntaxbaum als Grundlage für den Assemblercode (Parser).
- $A_3 \in \text{TIME}(O(2^n))$ Finde unter n Gegenständen mit verschiedenen Werten und Größen die wertvollste Kombination unterhalb einer bestimmten maximalen Größe (Rucksackproblem).
- $A_4 \in \text{TIME}(O(n!))$ Finde in einem Graphen der Größe n die kürzeste Strecke, die alle n Ecken miteinander verbindet (Traveling Salesperson Problem).

Algorithmus	Laufzeit	$n = 10$	$n = 20$	$n = 50$	$n = 100$
A_1	n	10 ES 10^{-8} sec	20 ES $2 \cdot 10^{-8}$ sec	50 ES $5 \cdot 10^{-8}$ sec	100 ES 10^{-7} sec
A_2	n^3	1000 ES 10^{-6} sec	8000 ES $8 \cdot 10^{-6}$ sec	10^5 ES 10^{-4} sec	10^6 ES 0.001 sec
A_3	2^n	1024 ES 10^{-6} sec	10^6 ES 0.001 sec	10^{15} ES 13 Tage	10^{30} ES $4 \cdot 10^{13}$ Jahre
A_4	$n!$	$3 \cdot 10^6$ ES 0.003 sec	$2 \cdot 10^{18}$ ES 77 Jahre	$3 \cdot 10^{64}$ ES 10^{48} Jahre	10^{158} ES $3 \cdot 10^{141}$ Jahre

Bild: Annahme: ein PC schafft 10^9 ES (Elementarschritte) pro Sekunde [Schöning, 2008]

Zeit- und Raumkomplexitätsklassen

Erinnerung: TIME, NTIME, SPACE, NSPACE enthalten alle Probleme, die sich von einer (N)TM innerhalb von einer bestimmten Anzahl von Schritten oder mit einem zusätzlichen Speicherplatzbedarf lösen lassen.

Bemerkungen:

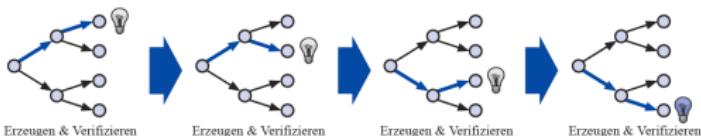
- Die **theoretische** Komplexität eines Problems sagt wenig über seine **praktische** Lösung: Es gilt z.B.
 $L_{C3} = \{(ab)^n \mid n \in \mathbb{N}\} \in \text{TIME}(O(n))$, da eine effiziente TM, diese Sprache in linearer Zeit erkennen könnte. Es existieren aber auch ineffiziente TM, die mehr Zeit benötigen!
- Wachstumsklassen sind hierarchisch ineinander enthalten: wenn ein Problem in $\text{SPACE}(O(n))$ liegt, ist es automatisch auch in $\text{SPACE}(O(n^2)), \text{SPACE}(O(n^{100})), \text{SPACE}(O(2^n)), \dots$ enthalten.
- Da jede TM auch eine NTM ist, gilt für alle Funktionen f
 - ▶ $\text{TIME}(f(n)) \subseteq \text{NTIME}(f(n))$
 - ▶ $\text{SPACE}(f(n)) \subseteq \text{NSPACE}(f(n))$

Determinismus versus Nichtdeterminismus

Erinnerung: TM und NTM können die gleichen Probleme lösen, aber NTM benötigen durch Parallelverarbeitung weniger Platz und Zeit als die entsprechenden TM.

■ Deterministische Lösungsstrategie

Der Suchraum wird systematisch durchkämmt (Brute-Force-Methode).



■ Nichtdeterministische Lösungsstrategie

Der richtige Berechnungspfad wird geraten und die Lösung anschließend verifiziert.



Quelle: [Hoffmann, 2011]

Bemerkung:

- NTM sind für die Analyse von Problemen wichtig, aber leider nur ein theoretisches Modell (unendliche Parallelität bzw. Orakel)
- Probleme, welche in $\text{NTIME}(f(n))$ bzw. $\text{NSPACE}(f(n))$ liegen, aber nicht in $\text{TIME}(f(n))$ bzw. $\text{SPACE}(f(n))$ sind
 - ▶ mit realistischer Hardware meist nicht in Zeit bzw. Platz $f(n)$ lösbar
 - ▶ meist schwieriger als Probleme aus $\text{TIME}(f(n))$ bzw. $\text{SPACE}(f(n))$

P und NP

Bemerkung: Von besonderem Interesse sind die Sprachen, die von (N)TMs in polynomieller Laufzeit akzeptiert werden können:

Definition

Die polynomiellen Komplexitätsklassen **P** und **NP** sind definiert als

$$\begin{aligned} P &:= \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k) \\ \text{NP} &:= \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k) \end{aligned}$$

Bemerkungen:

- (N)P enthält alle Sprachen / Probleme, die von einer (N)TM in **polynomieller Laufzeit** akzeptiert /gelöst werden.
- P wird auch als die Klasse der **effizient** entscheidbaren Sprachen, der berechenbaren Funktionen oder lösbarren Probleme bezeichnet.
- Es gilt $P \subseteq NP$. Ob $P = NP$ oder $P \neq NP$ gilt, ist offen, 1 Million Dollar wert und eines der **7 Millennium-Probleme**.

P - die Klasse der effizient lösbaren Problem

Erinnerung: Vergleich von Algorithmen mit Laufzeiten $O(n)$, $O(n^3)$, $O(2^n)$, $O(n!)$ zeigt, dass nur $O(n)$, $O(n^3)$ für große Eingaben innerhalb einer sinnvollen **Effizienzgrenze** lösbar sind.

Dies bleibt auch nach der Einführung schnellerer Hardware so:

Algorithmus	Laufzeit	Effizienzgrenze auf heutigem Computer	Effizienzgrenze auf 10-mal schnellerem Computer	Effizienzgrenze auf 100-mal schnellerem Computer
A_1	n	n_1	$10 \cdot n_1$	$100 \cdot n_1$
A_2	n^3	n_2	$\sqrt[3]{10} \cdot n_2 = 2.15 \cdot n_2$	$\sqrt[3]{100} \cdot n_2 = 4.64 \cdot n_2$
A_3	2^n	n_3	$\log_2(10 \cdot n_3) = n_3 + 3.3$	$\log_2(100 \cdot n_3) = n_3 + 6.6$
A_4	$n!$	n_4	$\approx n_4 + 1$	$\approx n_4 + 2$

Bild: Schnellere Hardware beschleunigt nur polynomiale Laufzeiten deutlich [Schöning, 2008]

Achtung: Die Grenzen zwischen „effizient“ und „ineffizient“ sind unscharf. $P_1 \in O(n^{1000})$ gilt strenggenommen als effizient lösbar, $P_2 \in O(1.1^n)$ gilt als ineffizient lösbar. Allerdings fordert P_1 nur für $n \gtrapprox 100000$ mehr Rechenaufwand.

NP - die Klasse der (vermutlich) nicht effizient lösbarer Probleme

Erinnerung: Das Problem an NP ist das “N“: Ein Problem das nur von einer NTM in Polynomialzeit lösbar ist, die rein theoretisch unendlich viele Berechnungen parallel durchführen kann, ist in Realität nicht in polynomieller, sondern eher in exponentieller Zeit lösbar.

Alternative Charakterisierungen:

Wikipedia NP enthält alle Entscheidungsprobleme, bei denen „Ja“-Antworten effizient in Polynomialzeit verifiziert werden können.

Problem: Es ist meist aufwändig (falls Problem nicht in P selbst ist), eine solche „Ja“-Antwort zu finden.

[Schöning, 2008] Ein Problem ist in NP, falls es zu jeder Eingabe x eine zweite Eingabe y gibt, so dass ein effizienter Algorithmus bei Eingabe von x, y 1 ausgibt, falls x eine „Ja“-Antwort ist und 0 sonst.

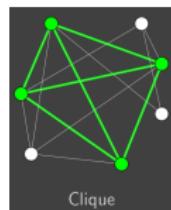
Problem: y muss gefunden werden. Brute Force-Vorgehensweise: Untersuchung von (Größe des Eingabealphabets) $^{\text{polynom}(x)}$ vielen Möglichkeiten \Rightarrow exponentielle, nicht-polynomiale Laufzeit.

Beispiele für Probleme aus P (Quelle: Uni Magdeburg)

- Das Eindeutigkeitsproblem (UNIQUE)
 - ▶ informell: enthält ein String nur eindeutige Teilstrings?
 - ▶ $\text{UNIQUE} = \{w_1\#w_2\#\dots\#w_k \mid w_1, \dots, w_k \in \{0, 1\}^*\text{ und } w_i \neq w_j \text{ für alle } i \neq j\}$
 - ▶ UNIQUE enthält also alle Wörter $\omega \in \{0, 1, \#\}^*$, so dass von den durch # getrennten 0/1-Strings keiner doppelt vorkommt.
- Das Pfadexistenzproblem (PATH)
 - ▶ informell: existiert in einem Graphen ein Weg zwischen zwei Knoten?
 - ▶ $\text{PATH} = \{\langle G, s, t \rangle \mid G \text{ ist ein ungerichteter Graph mit Knoten } s \text{ und } t \text{ und in } G \text{ gibt es einen Pfad der } s \text{ und } t \text{ verbindet}\}$
 - ▶ PATH enthält also alle Wörter die in einer geeigneten Codierung einen gerichteten Graphen und zweier seiner Knoten angeben, so dass diese in G durch einen Pfad (Folge aus mehreren Hops) verbunden sind.
- Primzahltest (PRIMES)
 - ▶ informell: ist eine Zahl eine Primzahl?
 - ▶ $\text{PRIMES} = \{\omega \in \{0, 1\}^* \mid \omega \text{ ist eine binär codierte Primzahl}\}$
 - ▶ PRIMES enthält also alle binär codierten Primzahlen.

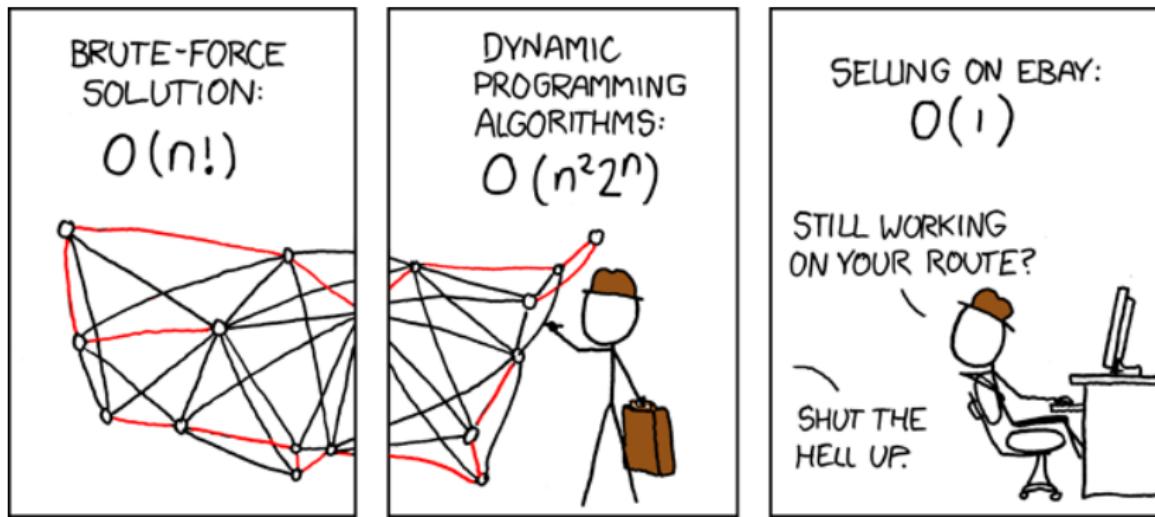
Beispiele für Probleme aus NP (Quelle: Uni Magdeburg)

- Erfüllbarkeitsproblem der Aussagenlogik (SAT)
 - ▶ informell: ist eine Formel erfüllbar?
 - ▶ $SAT = \{F \mid F \text{ ist eine erfüllbare logische Formel in KNF}\}$
 - ▶ Eine Formel ist in konjunktiver Normalenform (KNF), falls sie nach dem Schema $F = \bigwedge_i \bigvee_j (\neg)x_{ik}$ aufgebaut ist (wobei x_{ij} atomare Formeln sind).
- Verbundene Knoten in einem Graphen (CLIQUE)
 - ▶ informell: gibt es in einem Graphen eine Clique?
 - ▶ $CLIQUE = \{\langle G, k \rangle \mid G \text{ ist ein ungerichteter Graph, } k \in \mathbb{N}, \text{ und es gibt in } G \text{ eine Clique der Größe } k\}$
 - ▶ Eine Clique ist eine Teilmenge von Knoten eines Graphens, in der jeder Knoten zu jedem anderen benachbart ist.
- Traveling Salesperson Problem (TSP)
 - ▶ informell: was ist die kürzeste Rundreise?
 - ▶ $TSP = \{\langle D_{ij}, K \rangle \mid D_{ij} \text{ ist die Distanzmatrix zwischen } n \text{ Städten und } K \text{ sind höchstens die Kosten der kürzesten Rundreise}\}$
 - ▶ TSP enthält also alle K , für welche die Kosten der kürzesten existierenden Rundreise höchstens K sind.



Quelle: Uni
Magdeburg

TSP in der Praxis



Quelle: xkcd.com

PSPACE und NPSPACE

Definition

Die polynomiellen Komplexitätsklassen **PSPACE** und **NPSPACE** sind definiert als

$$\text{PSPACE} := \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k)$$

$$\text{NPSPACE} := \bigcup_{k \in \mathbb{N}} \text{NSPACE}(n^k)$$

Bemerkungen:

- (N)PSPACE enthält alle Sprachen / Probleme, die von einer (N)TM in **polynomiellem (zusätzlichem) Platz** akzeptiert / gelöst werden.
- (N)P ⊆ (N)PSPACE, da jede benutzte Bandzelle in mindestens einem Rechenschritt beschrieben werden muss.
- PSPACE = NPSPACE, da jede NTM in eine TM übersetzt werden kann, welche die gleiche Sprache akzeptiert und nur quadratisch höheren Platzbedarf hat (Savitch, 1970).
- Beispiel: Erfüllbarkeitsproblem für aussagenlogische Formeln mit Quantoren (QSAT)

EXP und NEXP

Definition

Die exponentiellen Komplexitätsklassen **EXP** und **NEXP** sind definiert als

$$\text{EXP} := \bigcup_{c,k \in \mathbb{N}} \text{TIME}(c^{n^k})$$

$$\text{NEXP} := \bigcup_{c,k \in \mathbb{N}} \text{NTIME}(c^{n^k})$$

Bemerkungen:

- (N)EXP enthält alle Sprachen / Probleme, die von einer (N)TM in **exponentieller Laufzeit** akzeptiert / gelöst werden.
- Probleme aus (N)EXP sind (für größere Instanzen) nicht mehr effizient lösbar.
- Beispiel: $L = \{\langle M, x, k \rangle \mid M \text{ ist eine TM, die bei Eingabe } x \text{ nach höchstens } k \text{ Schritten hält}\}$

(Nahezu) vollständige Hierarchie von Komplexitätsklassen

- Es gibt nicht nur Komplexitätsklassen basierend auf deterministischen und nichtdeterministischen Turing-Maschinen sondern noch viele weitere Modelle. Eine gute Übersicht bietet der [Complexity Zoo](#).
- Übersicht über die wichtigsten Klassen im [Petting Zoo](#).



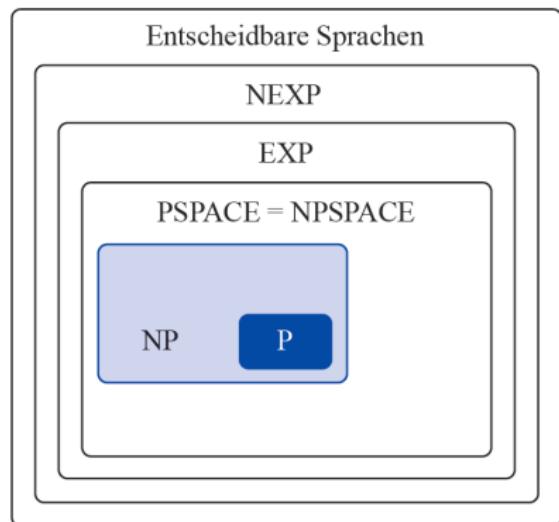
Quelle: Complexity Zoology

Vereinfachte Hierarchie von Komplexitätsklassen

Es gilt

$$P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXP \subseteq NEXP$$

- **Legende:** Ein Problem, das in einer heller bzw. weiß markierten Klasse liegt, ist nur für **sehr kleine** Instanzen lösbar.
- Es ist bekannt, dass $P \subset EXP$ und $NP \subset NEXP$ ([Zeit-Hierarchiesatz](#)); daher wird vermutet, dass alle Inklusionen echt sind, jedoch konnte dies bisher weder bewiesen noch widerlegt werden.



Quelle: [Hoffmann, 2011]

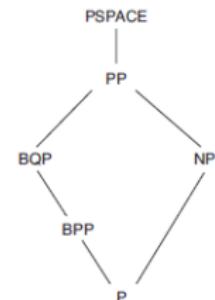
Vereinfachte Hierarchie mit Quanten-Komplexitätsklassen

Es gilt

$$P \subseteq BPP \subseteq BQP \subseteq PP \subseteq PSPACE$$

Legende

- **BPP** Alle Probleme, die von einer **probabilistischen TM** in Polynomialzeit mit Fehlerwahrscheinlichkeit höchstens **1/3** lösbar sind
- **BQP** Alle Probleme, die von einer **Quanten-TM** in Polynomialzeit mit Fehlerwahrscheinlichkeit höchstens **1/3** lösbar sind
- **PP** Alle Probleme, die von einer **probabilistischen TM** in Polynomialzeit mit Fehlerwahrscheinlichkeit höchstens **1/2** lösbar sind



Quelle: [Aaronson, 2013]

Achtung: Die Beziehung zwischen BPP, BQP, PP und NP ist unklar!!
Insbesondere unklar: Können Quantencomputer NP-vollständige Probleme schnell lösen? (Vermutung: nein)

Mehr Infos: siehe z.B. [Aaronson, 2013] oder [Homeister, 2018].

Polynomielle Reduktion

Bemerkung: Nicht alle Probleme aus NP sind gleich schwer zu lösen.

- Ein „besonders schweres“ NP-Problem K nennt man **NP-vollständig**.
- „Besonders schwer“ bedeutet, dass sich alle anderen Probleme in NP auf K **reduzieren** lassen.

Definition

Seien $L \subseteq \Sigma^*$, $K \subseteq \Gamma^*$ zwei Sprachen. L ist genau dann **polynomiell reduzierbar** auf K , $L \leq_P K$, falls $f : \Sigma^* \rightarrow \Gamma^*$, total, existiert mit:

- f ist in polynomieller Zeit berechenbar
- $\omega \in L \Leftrightarrow f(\omega) \in K$

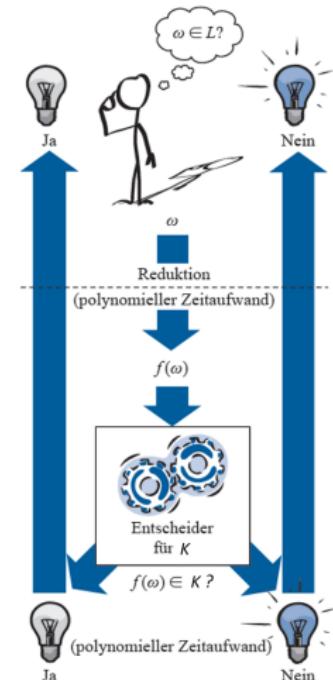


Bild: [Hoffmann, 2011]

NP-hart, NP-vollständig

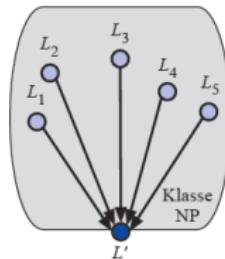
Definition

Eine Sprache $K \subseteq \Sigma^*$ heißt

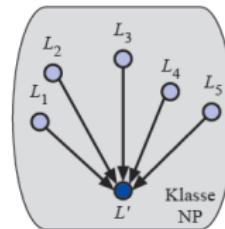
- **NP-hart**, falls für alle $L \in \text{NP}$ gilt: $L \leq_P K$.
- **NP-vollständig**, falls K NP-hart und $K \in \text{NP}$.

Bemerkungen:

- Ein Problem X ist genau dann NP-hart, wenn sich alle nichtdeterministisch in Polynomialzeit lösbar Probleme auf X reduzieren lassen.
- Ist X NP-hart und selbst nichtdeterministisch in Polynomialzeit lösbar, so ist X ein NP-vollständiges Problem.
- NP-hart ist eine etwas krumme Übersetzung des englischen „NP-hard“, korrekter wäre „NP-schwer“.



Ein Problem L' ist **NP-hart**, wenn sich alle Probleme aus der Klasse NP darauf reduzieren lassen. L' kann selbst in NP liegen, muss es aber nicht.

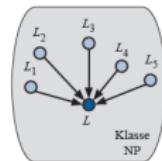


Ein Problem L' ist **NP-vollständig**, wenn sich alle Probleme aus der Klasse NP darauf reduzieren lassen und L' selbst in NP liegt.

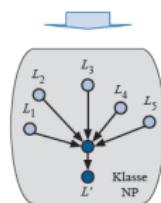
Quelle: [Hoffmann, 2011]

Wieso sind NP-vollständige Probleme so interessant?

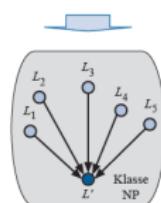
- Kann man für ein einziges NP-vollständiges Probleme zeigen, dass es **deterministisch** in Polynomialzeit gelöst werden kann, so wären alle anderen NP-vollständigen Probleme ebenfalls deterministisch in Polynomialzeit lösbar.
- Und das wiederum würde bedeuten, dass **alle** Probleme aus NP deterministisch in Polynomialzeit lösbar wären, also $P = NP$ gilt.
- Insbesondere ließen sich also alle Probleme aus NP (z.B. auch das Faktorisierungsproblem ganzer Zahlen) auf **reeller Hardware** in Polynomialzeit lösen.
- Erstes als NP-vollständig identifiziertes Problem (Cook, 1970): Erfüllbarkeitsproblem der Aussagenlogik (SAT).
- Wer mehr wissen will: siehe [[Hoffmann, 2015](#)]



Ist L NP-vollständig, so lassen sich alle Probleme aus NP auf L reduzieren.



Lässt sich L polynomell auf $L' \in$ NP reduzieren, so folgt aus der Transitivität, ...



... dass sich alle Probleme aus NP auch auf L' reduzieren lassen. L' ist damit ebenfalls NP-vollständig.

Quelle: [[Hoffmann, 2011](#)]

Das Rucksackproblem (KNAPSACK)

- Gegeben: N Gegenstände mit Gewicht w_i und Nutzen v_i haben
- Gesucht: Auswahl von $1 \leq k \leq N$ Gegenständen, die leichter als Obergrenze W sind und deren Nutzsumme V maximal ist
- Formell: Bestimme $a_i \in \{0, 1\}$ so dass
 - ▶ $\sum_{i=1}^N a_i w_i \leq W$ gilt
 - ▶ $\sum_{i=1}^N a_i v_i = V$ maximal ist
- Anwendungen:
Gewinn-/Nutzenmaximierung in Groß- und Privat-Logistik,
Finanztransaktionen, ...

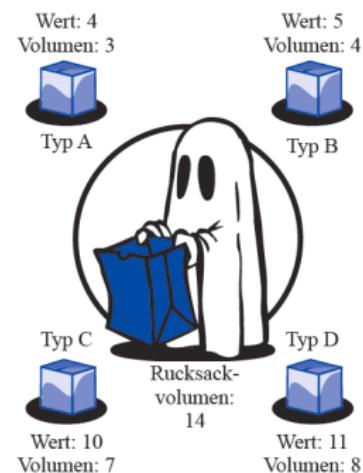


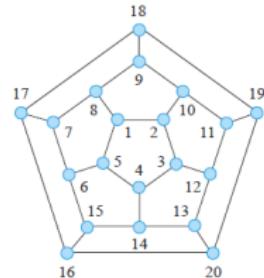
Bild: Rucksackproblem am Beispiel eines Einbrechers
[Hoffmann, 2011]

Beweis via Reduktion von EXACT COVER (Problem der exakten Überdeckung) auf KNAPSACK

Das Cliquesproblem (CLIQUE))

- Gegeben: Graph G mit Knotenmenge V und Kantenmenge E , $n \in \mathbb{N}$, $n \leq |V|$
- Frage: Existiert eine Clique C mit $|C| \geq n$, also eine Menge C von mindestens n Knoten v_1, \dots, v_n die paarweise durch eine Kante aus E verbunden sind?
- Beweis via Reduktion von 3SAT (Erfüllbarkeitsproblem für Aussagen in KNF mit höchstens 3 Literalen) auf CLIQUE
- Anwendungen: Optimierung von chemischen Molekülen, Generierung automatischer Testmuster, Analyse sozialer Netzwerke, ...

■ Beispiel 1



■ Beispiel 2

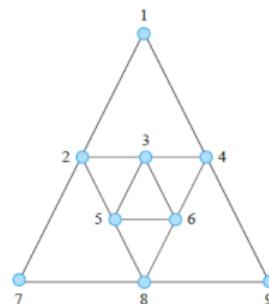


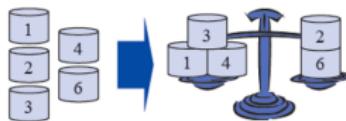
Abbildung 7.30: Eine Clique der Größe k ist eine Menge von k paarweise verbundenen Knoten. Während der obere Graph ausschließlich Cliques der Größe 2 enthält, besitzt der untere Graph 4 Cliques der Größe 3.

Quelle: [Hoffmann, 2015]

Ausgewählte NP-vollständige Probleme

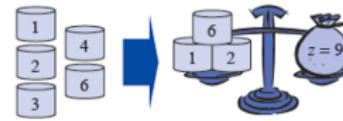
■ PARTITION

Gegeben sei eine Folge natürlicher Zahlen x_1, \dots, x_n . Lassen sich die Zahlen so aufteilen, dass die Summe beider Partitionen gleich ist? Mit anderen Worten: Existiert eine Teilmenge $S \subseteq \{1, \dots, n\}$ mit $\sum_{i \in S} x_i = \sum_{i \notin S} x_i$?



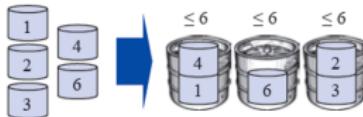
■ SELECT

Gegeben seien eine Folge natürlicher Zahlen $x_1, \dots, x_n \in \mathbb{N}$ und eine weitere natürliche Zahl $z \in \mathbb{N}$. Existiert eine Auswahl von Elementen x_{i_1}, \dots, x_{i_k} , deren Summe z ergibt? Mit anderen Worten: Existiert eine Teilmenge $S \subseteq \{1, \dots, n\}$ mit $\sum_{i \in S} x_i = z$?



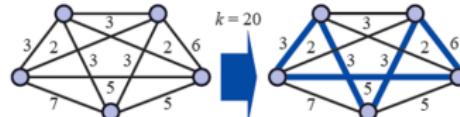
■ BIN PACKING

Gegeben seien k Container mit dem gleichen Fassungsvermögen V . Für n Zahlen x_1, \dots, x_n ist zu entscheiden, ob sich diese auf die Container verteilen lassen, ohne das Fassungsvermögen zu überschreiten. Mit anderen Worten: Existiert eine Zuordnung $f : \{1, \dots, n\} \rightarrow \{1, \dots, k\}$ mit $\sum_{f(i)=j} x_i < V$?



■ TRAVELING SALESMAN

Gegeben sei eine Menge von Städten $\{S_1, \dots, S_n\}$ sowie eine Straßenkarte, auf der die Entfernungen zwischen S_i und S_j verzeichnet sind. Für eine gegebene Konstante k gilt es zu entscheiden, ob alle Städte auf einem Rundweg besucht werden können, der die Gesamtstrecke k nicht überschreitet.



Quelle: [Hoffmann, 2011]

Listen NP-vollständiger Problem

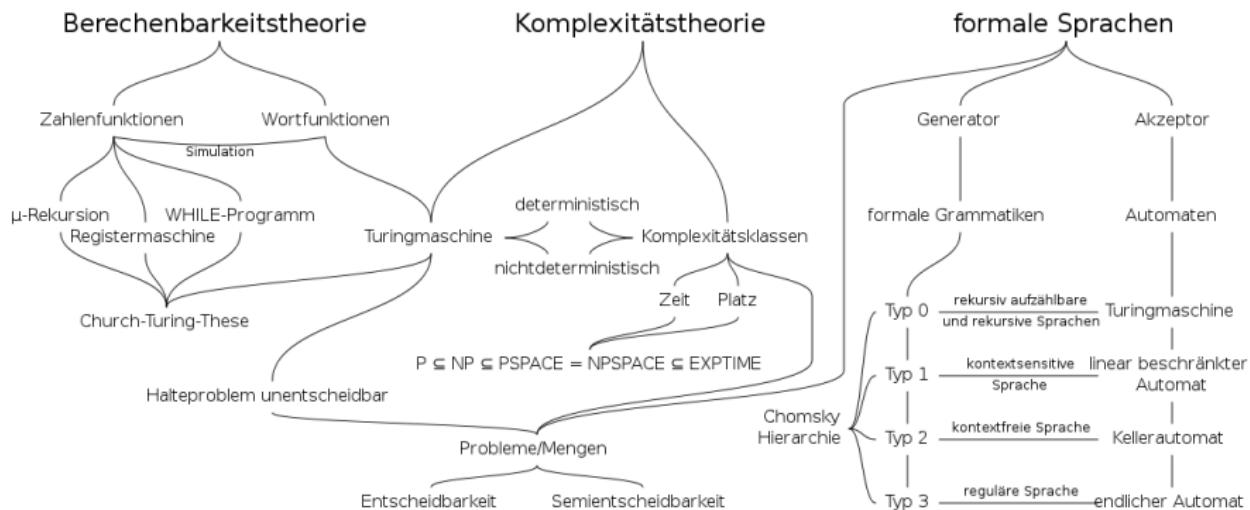
- An Annotated List of Selected NP-complete Problems
- A Useful List of NP-Complete Problems
- Wikipedia
- ...

Games and puzzles [edit]

- Battleship
- Bulls and Cows, marketed as [Master Mind](#): certain optimisation problems but not the game itself.
- Eternity II
- (Generalized) FreeCell^[52]
- Fillomino^[53]
- Hashiwokakero^[54]
- Heyawake^[55]
- (Generalized) Instant Insanity^[56]
- Kakuro (Cross Sums)
- Kingdomino^[57]
- Kuromasu (also known as Kurodoko)^[58]
- LaserTank^[59]
- Lemmings (with a polynomial time limit)^[60]
- Light Up^[61]
- Masyu^[62]
- Minesweeper Consistency Problem^[63] (but see Scott, Stege, & van Rooij^[64])
- Nimber (or Grundy number) of a directed graph.^[65]
- Numberlink
- Nonograms
- Nurikabe
- (Generalized) Pandemic^[66]
- Optimal solution for the $N \times N \times N$ Rubik's Cube^[67]
- SameGame
- (Generalized) Set^[68]
- Slither Link on a variety of grids^{[69][70][71]}
- (Generalized) Sudoku^{[69][72]}
- Problems related to Tetris^[73]
- Verbal arithmetic

Quelle: [Wikipedia](#)

Zum Abschluss: The Big Picture



Quelle: [Wikipedia](#)

Verwendete oder empfohlene Literatur I

[Aaronson, 2013] Aaronson, S. (2013).

Quantum computing since Democritus.

Cambridge University Press.

[Hedtstück, 2012] Hedtstück, U. (2012).

Einführung in die theoretische Informatik: formale Sprachen und Automatentheorie.

Oldenbourg Verlag.

Als eBook in der HTWG-Bibliothek verfügbar.

[Hoffmann, 2011] Hoffmann, D. W. (2011).

Theoretische Informatik.

Carl Hanser Verlag GmbH & Co. KG, 2. Auflage.

Als eBook in der HTWG-Bibliothek verfügbar.

Verwendete oder empfohlene Literatur II

[Hoffmann, 2015] Hoffmann, D. W. (2015).

Theoretische Informatik.

Carl Hanser Verlag GmbH & Co. KG, 3. Auflage.

Als eBook in der HTWG-Bibliothek verfügbar.

[Homeister, 2018] Homeister, M. (2018).

Quantum Computing verstehen.

Springer Vieweg, 5. Auflage.

Als eBook in der HTWG-Bibliothek verfügbar.

[Hopcroft et al., 2011] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2011).

Einführung in die Automatentheorie, formale Sprachen und Berechenbarkeit (bzw. Komplexitätstheorie); engl.: Introduction to automata theory, languages and computation.

Pearson, 3. Auflage.

Als eBook in der HTWG-Bibliothek verfügbar.

Verwendete oder empfohlene Literatur III

[Schöning, 2008] Schöning, U. (2008).

Ideen der Informatik: Grundlegende Konzepte der Theoretischen Informatik.

Oldenbourg Verlag.

Als eBook in der HTWG-Bibliothek verfügbar.

[Turing, 1936] Turing, A. M. (1936).

On computable numbers, with an application to the entscheidungsproblem.

Journal of Math, 58(345-363):5.

[Wagenknecht and Hielscher, 2014] Wagenknecht, C. and Hielscher, M. (2014).

Formale Sprachen, abstrakte Automaten und Compiler.
Springer Vieweg, 2. Auflage.

Als eBook in der HTWG-Bibliothek verfügbar.