

F3 — Berechenbarkeit und Komplexität

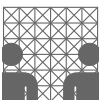
Matthias Jantzen (nach Vorlage von Berndt Farwer)

Fachbereich Informatik

AB „Theoretische Grundlagen der Informatik“ (TGI)

Universität Hamburg

jantzen@informatik.uni-hamburg.de



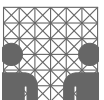
Zielgruppe

1. Informatik (3. Semester)
2. Wirtschaftsinformatik (3. Semester)
3. Allgemeine Ingenieurwissenschaften, Informatikingenieurwesen und TECMA (5. Semester)

Wichtig:

Beginn 12:15

Ende 13:45



Übungsgruppen

... gibt es **nur** für Wirtschaftsinformatiker und Studierende der TU-Harburg!

Mo 12–13 C-101

Heiko Rölke

Mo 13–14 C-101

Heiko Rölke

Do 14–15 C-101

Mathias Jantzen

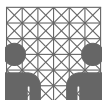
Do 15–16 C-101

Michael Köhler

Do 16–17 C-101

Michael Köhler

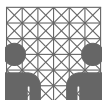
Die Aufgaben sind aber für jede(n) im WEB abrufbar, und sollten (am Besten in AG's) von allen bearbeitet werden!



Scheinkriterien

Bedingungen für die Ausstellung eines Scheines:

- 50% der erreichbaren Punkte
- regelmäßige, aktive Teilnahme an den Übungsgruppen
- Vorrechnen an der Tafel
- max. zweimaliges unentschuldigtes Fehlen



Skript

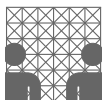
Das Skript zur Vorlesung ist erhältlich:

1. **gedruckt** über

- das Sekretariat von TGI (C-218)
- die Übungsgruppe

2. **elektronisch:** **Benutzer:**
 Passwort:

- <http://www.informatik.uni-hamburg.de> →
Gliederung → TGI → Lehre →
aktuelle Veranstaltungen → Hinweise,
Skript, Aufgaben und Musterlösungen
- <http://www.informatik.uni-hamburg.de/TGI/lehre/vl/WS0304/F3/F3.html>
- ... **dort auch Übungsaufgaben u. Lösungen!**

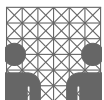


Ägyptische Multiplikation

... ein Zahlenbeispiel: $231 \cdot 101 = 23331$

231	101	
115	202	
57	404	
28	808	(zu streichen)
14	1616	(zu streichen)
7	3232	
3	6464	
1	12928	

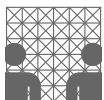
Diese Methode ist korrekt! (Beweis mit Hilfe der Binärdarstellungen der Zahlen oder Induktion.)



Ägyptische Multiplikation (2)

Beispielrechnung: $[23]_{10} \cdot [11]_{10}$

$[23]_2 =$	10111	1011	$= [11]_2$
$[11]_2 =$	1011	10110	$= [22]_2$
$[5]_2 =$	101	101100	$= [44]_2$
$[2]_2 =$	10	1011000	$= [88]_2$
$[1]_2 =$	1	10110000	$= [176]_2$
<hr/>			
		11111101	$= [253]_2$

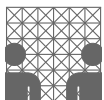


Ägyptische Multiplikation (3)

Warum ist das Verfahren für die Informatik interessant?

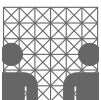
1. **ganzzahlige Division durch 2:**
letztes Bit abschneiden
2. **ganzzahlige Multiplikation mit 2:**
0 anhängen

Diese elementaren Operationen sind leicht zu implementieren!



Korrektheit des Verfahrens

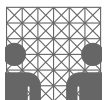
- Beispiele können höchstens **Fehler** im Verfahren aufdecken
- Korrektheit muss **bewiesen** werden, z.B. durch
 - Induktionsbeweis
 - Widerspruchsbeweis
 - andere mathematische Verfahren



Problemtypen

Je nach Aufgabenstellung lassen sich verschiedene Grundtypen von Problemen unterscheiden:

1. **Entscheidungsprobleme**
2. **Suchprobleme**
3. **Optimierungsprobleme**
4. **Abzählungsprobleme**
5. **Anzahlprobleme**



Entscheidungsprobleme

... zum Beispiel:

Gegeben:	Eine natürliche Zahl $n \in \mathbb{N}$.
Gesucht:	Antwort auf die Frage: Ist n eine Primzahl?
Antwort:	JA oder NEIN

Problem Π ist *Entscheidungsproblem*, gdw. Lösungsraum $\mathcal{L}(\Pi)$ besteht aus genau zwei Elementen und jede Instanz $I \in \mathcal{I}(\Pi)$ hat genau eine Lösung.

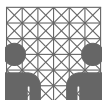
Z.B. $\mathcal{L}(\Pi) = \{0, 1\}$.



Suchprobleme

... zum Beispiel:

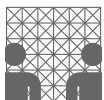
Gegeben:	ungerichteter Graph $G := (V, E)$ mit $E \subseteq V \times V$ und Knoten $v_1, v_2 \in V$.
Gesucht:	ein Weg von v_1 nach v_2
Antwort:	Kantenfolge eines Pfades oder „Es gibt keinen!“



Optimierungsprobleme

... zum Beispiel:

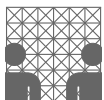
Gegeben:	gerichteter, bewerteter Graph $G := (V, E)$ mit $E \subseteq V \times \mathbb{N} \times V$ und Knoten $v_1, v_2 \in V$.
Gesucht:	ein günstigster Weg von v_1 nach v_2
Antwort:	Kantenfolge eines Pfades (evtl. mit Kosten) oder „Es gibt kei- nen!“



Abzählungsprobleme

... zum Beispiel:

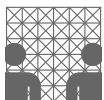
Gegeben:	endliche Menge von Objekten
Gesucht:	alle binären Suchbäume für diese Objekte
Antwort:	Aufzählung der binären Suchbäume



Anzahlprobleme

... zum Beispiel:

Gegeben:	zwei Klammersymbole [und]
Gesucht:	Wieviele korrekt geklammerte Terme mit $2n$ Klammersymbolen gibt es?
Antwort:	Eine Zahl.(aber welche?)



Der Begriff des **Algorithmus** und die **Turing-Maschine**

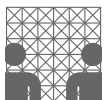


Algorithmus

... einige Algorithmus-Definitionen:

Ein **Algorithmus** liegt genau dann vor, wenn gegebene Größen, auch **Eingabegrößen**, **Eingabeinformationen** oder **Aufgaben** genannt, auf Grund eines Systems von **Regeln**, Umformungsregeln, in andere Größen, auch **Ausgabegrößen**, **Ausgabeinformationen** oder **Lösungen** genannt, umgeformt oder umgearbeitet werden.

Kleine Enzyklopädie MATHEMATIK, 1968

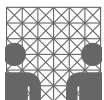


Algorithmus (2)

Ein **Algorithmus** ist eine präzise, d.h. in einer festgelegten Sprache abgefaßte, endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte.

Bauer, Goos, Informatik I, 1991

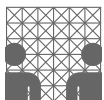
Wichtig: Es gibt *terminierende* und *nicht terminierende* Algorithmen!



Algorithmus (3)

Ein Algorithmus soll also

- schrittweise arbeiten (**Diskretheit**),
- nach jedem Schritt eindeutig bestimmen, was der nächste Schritt ist (**Determiniertheit**),
- einfache Schritte enthalten (**Elementarität**).
- auf eine hinreichend große Klasse von Instanzen anwendbar sein (**Generalität**).
- sich mit endlichen Mitteln beschreiben lassen (**endliche Beschreibbarkeit**)
 - nach endlichen vielen Schritten zu einer Lösung führen (**Konklusivität**). Dies ist **nicht** notwendig, aber oft erwünscht!

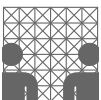


Berechenbarkeit

Bereits vor der Prägung des Algorithmusbegriffs:
Suche nach formaler Definition von
Berechenbarkeit.

Wichtige Observationen:

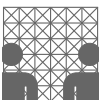
1. Unlösbare Probleme werden auch bei Verwendung immer schnellerer Rechner unlösbar bleiben.
2. Unter den lösbaren Problemen: Existenz schwer-lösbarer Probleme. Auch schnelle Computer können daran nichts ändern.



Beschreibungsformen für Alg.

- verbal
- ähnlich der Formulierung in einer Programmiersprache
- graphisch
- mathematisch

... aber wie soll man einen Algorithmus mathematisch notieren?!



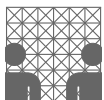
Beispiel

Summe der ersten n natürlichen Zahlen:

Gegeben: $n \in \mathbb{N}$
Gesucht: $s = \sum_{i=1}^n i$
Antwort: ?

(a) verbale Beschreibung:

Beginne mit $Summe := 0$. Addiere sukzessive zu $Summe$ die Zahlen 1 bis n . Am Ende enthält $Summe$ das gesuchte Resultat.



Beispiel (2)

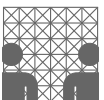
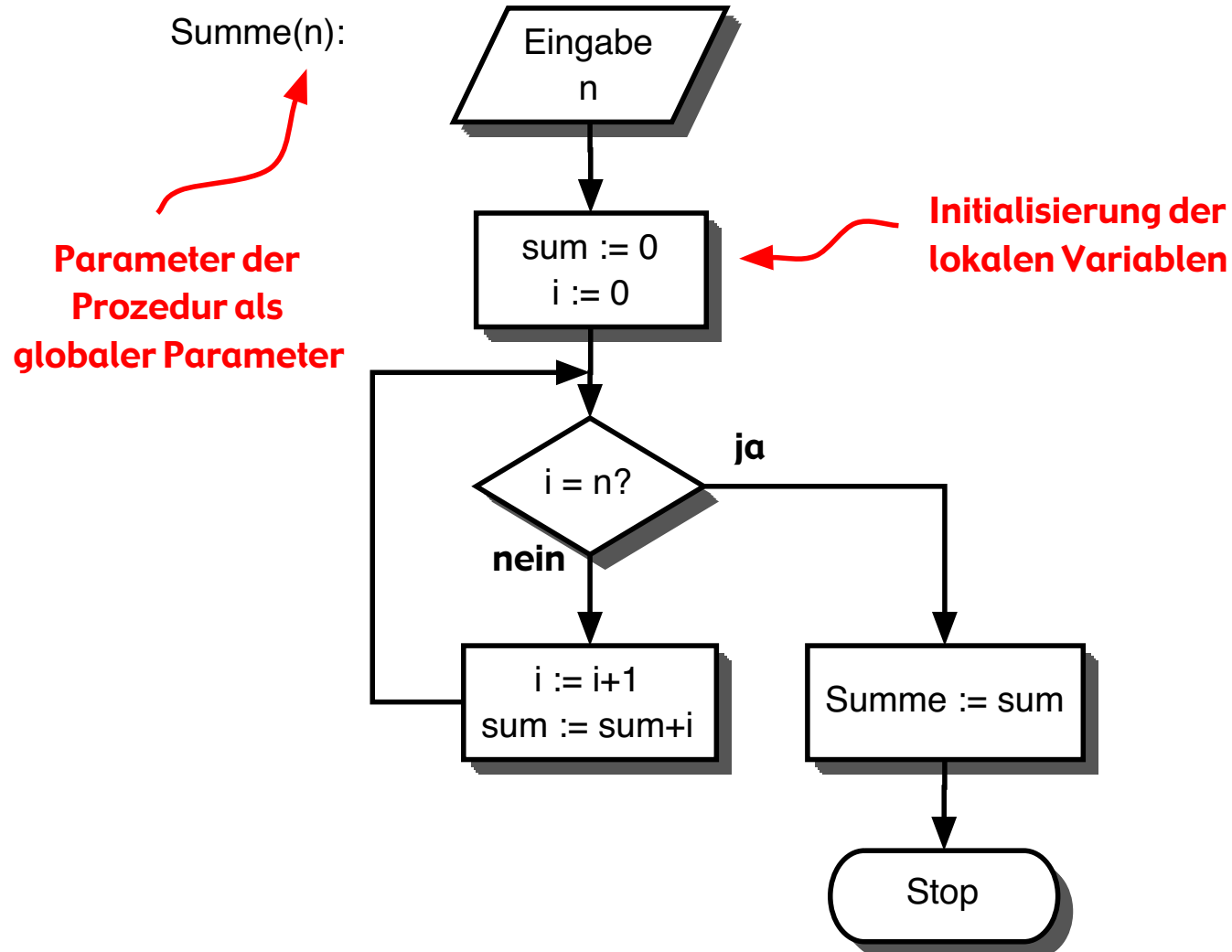
(b) programm-ähnliche Notation:

```
function Summe( $n$ )  
  {berechnet die Summe der natürlichen  
   Zahlen von 1 bis  $n$ }  
  sum  $\leftarrow$  0  
  for  $i \leftarrow 1$  to  $n$  do sum  $\leftarrow$  sum +  $i$   
  return sum
```



Beispiel (3)

(c) graphische Notation:



Die Gaußsche Formel

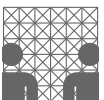
Theorem: Für beliebiges $n \in \mathbb{N}$ gilt:

$$1 + 2 + 3 + \dots + n = \binom{n+1}{2} = \frac{(n+1) \cdot n}{2}.$$

Algorithmus: Summe der ersten n natürlichen Zahlen ist $\binom{n+1}{2}$.

... das sieht viel einfacher aus, aber ist es auch korrekt?

→ Das muss erst bewiesen werden!



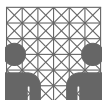
Beweis der Gaußschen Formel

(a) direkte Methode von Gauß:

$$\begin{aligned}2s &= 2 \cdot \sum_{i=1}^n i \\ &= 1 + 2 + \dots + (n-1) + n + \\ &\quad n + (n-1) + \dots + 2 + 1 \\ &= (n+1) \cdot n\end{aligned}$$

Also:

$$s = \frac{n(n+1)}{2}$$



Beweis der Gaußschen Formel (2)

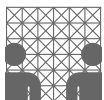
(b) mit vollständiger Induktion:

Verankerung:

$$n = 0 \longrightarrow \sum_{i=1}^0 i = 0 = \frac{0 \cdot (0 + 1)}{2}$$

Induktionsannahme:

Die Gaußsche Formel gilt für festes, aber beliebiges $m \in \mathbb{N}$.

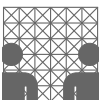


Beweis der Gaußschen Formel (2)

Induktionsschritt:

$$\begin{aligned}\sum_{i=0}^{m+1} i &= \sum_{i=0}^m i + (m + 1) \\ &= \frac{m \cdot (m+1)}{2} + (m + 1) \\ &= \frac{m \cdot (m+1) + 2 \cdot (m+1)}{2} \\ &= \frac{(m+1)(m+2)}{2}\end{aligned}$$

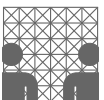
Somit ist die Annahme bewiesen!



Problemlösung

... besteht aus:

- Problem, vorhandene Eingaben und gewünschte Ausgaben korrekt erfassen
- Algorithmus finden, der das Problem löst
- Algorithmus als korrekt nachweisen
 - mittlere/schlechteste Laufzeit und Speicherbedarfe ermitteln
 - Algorithmus effizient implementieren



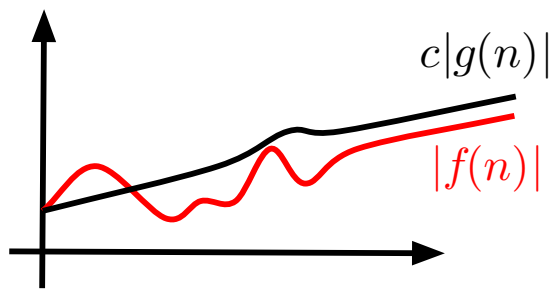
Die Landau- oder O-Notation



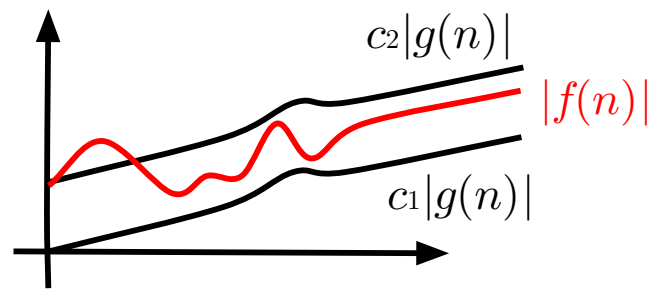
Anschaulich

Wir betrachten das Wachstum von Funktionen:

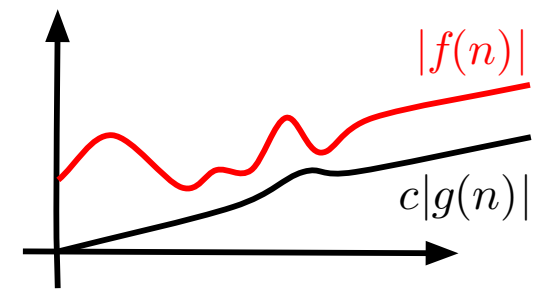
Auf *Paul Bachmann* (1894) geht die von E. Landau popularisierte Notation zurück.



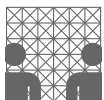
$$f(n) \in O(g(n))$$



$$f(n) \in \Theta(g(n))$$



$$f(n) \in \Omega(g(n))$$



O-Notation

Definition:

Eine Funktion $f : \mathbb{N} \rightarrow \mathbb{R}$

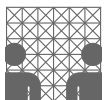
wächst mit der Ordnung $g(n)$ bzw. g , geschrieben

$$f(n) \in O(g(n)) \text{ bzw. } f \in O(g),$$

falls $g : \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion ist und eine Konstante $c \in \mathbb{R}^+ := \{x \in \mathbb{R} \mid x > 0\}$ existiert, so dass

$$|f(n)| \leq c \cdot |g(n)|$$

für alle, bis auf endlich viele $n \in \mathbb{N}$ gilt.



O-Notation (2)

Etwas knapper notiert liest sich das so:

$$O(g) =$$

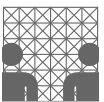
$$\{f : \mathbb{N} \rightarrow \mathbb{R} \mid (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0) \\ [|f(n)| \leq c |g(n)|]\}.$$

... die Menge aller Funktionen, für die g (mit einem konstanten Faktor) eine obere Schranke ist.

(„ f wächst nicht schneller als g “)

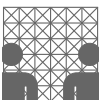
Mit $O(f)$ bezeichnen wir also eine Menge von Funktionen.

„ $f \in O(g)$ ist **von der Ordnung** $O(g)$.“



Zusammenfassung

- Die O-Notation ist asymptotische Notation. Sie spiegelt das Verhalten einer Funktion nur für große n korrekt wider. $f(n) \in O(g(n))$ bedeutet:
 - $g(n)$ und $f(n)$ haben für große n ein ähnliches Wachstumsverhalten.
 - Es ist $\frac{|f(n)|}{|g(n)|} \leq c$ für alle $n \geq n_0$ mit $g(n) \neq 0$.
 - Es gilt, falls $a_k \neq 0$:
$$O(a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0) = O(n^k)$$
 - $O(f(n) + g(n)) = O(\max(|f(n)|, |g(n)|))$.



Beispiele zur O-Notation

Beispiele:

Seien f und g definiert durch:

$$f(n) := 12n^4 - 11n^3 + 1993 \text{ und } g(n) := 7n^3 - n.$$

Dann ist $g \in O(f)$ aber nicht $f \in O(g)$.

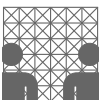
Auch gilt

$$f \in O(n^4)$$

und

$$f \in O(7028060n^4 - 1948).$$

$$1000x \in O(x^2 - x)$$



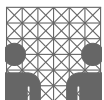
Die Funktionsklassen $\Omega(g)$

Definition:

$$\Omega(g) = \{ f : \mathbb{N} \rightarrow \mathbb{R} \mid (\exists c \in \mathbb{R}^+) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) [c |g(n)| \leq |f(n)|] \}$$

... die Menge aller Funktionen, für die g (mit einem konstanten Faktor) eine untere Schranke ist.

(„ f wächst nicht langsamer als g “)



Funktionsklassen $\Theta(g)$

Definition:

$$\Theta(g) =$$
$$\{f : \mathbb{N} \rightarrow \mathbb{R} \mid$$
$$(\exists c_1, c_2 \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})$$
$$(\forall n \geq n_0)[c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)|]\}.$$

„ f wächst etwa so schnell wie g “



Beispiel zu $\Theta(g)$

Beispiel: Um zu beweisen, dass

$$\frac{1}{2}n^2 - 3n \in \Theta(n^2) \quad n \rightarrow \infty$$

muss man solche $c_1 > 0, c_2, n_0$ finden, dass

$$c_1 n^2 \leq \left| \frac{1}{2}n^2 - 3n \right| \leq c_2 n^2 \quad \text{für } n \geq n_0$$

$$c_1 \leq \left| \frac{1}{2} - \frac{3}{n} \right| \leq c_2 \quad \text{für } n \geq n_0$$

Lösungen: $c_1 \leq \frac{1}{14}, \quad c_2 \geq \frac{1}{2}, \quad n \geq 7$



Weitere Funktionsklassen

Definition:

$$o(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid (\forall c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}) \\ (\forall n \geq n_0)[|f(n)| \leq c|g(n)|]\}$$

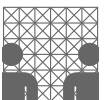
„ f wächst langsamer als g “

Beispiel: $\ln(x) \in o(\sqrt{x})$ und $\sqrt{x} \in o(x)$

Definition:

Es sei $f \in \omega(g)$ genau dann, wenn $g \in o(f)$

„ f wächst schneller als g “



Zusammenhänge

... grundlegende Beziehungen zwischen Θ -, O - und Ω -Notation:

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ und } f(n) \in \Omega(g(n)).$$

Eine Auswahl von Regeln zur Manipulation von O -Ausdrücken:

$$n^m \in O(n^{m'}) \text{ falls } m \leq m'$$

$$f(n) \in O(f(n))$$

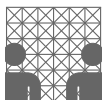
$$cO(f(n)) = O(f(n))$$

$$O(O(f(n))) = O(f(n))$$

$$O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$$

$$O(f(n))O(g(n)) = O(f(n)g(n))$$

$$O(f(n)g(n)) = f(n)O(g(n))$$



kanonische Erweiterung

Man schreibt $f(n) + O(g(n))$ für

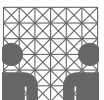
$$\{g' \mid g'(n) := f(n) + h(n) \text{ und } h \in O(g)\}$$

... zum Beispiel:

$$\sum_{k=0}^n (e + O(k)) \quad e, k \in \mathbb{R}$$

$$e + O(k) = \{f : \mathbb{N} \rightarrow \mathbb{N} \mid \exists c : |f(n, k)| \leq c \cdot k\}$$

Also: $\sum_{k=0}^n (e + O(k))$ enthält alle Funktionen der
Form: $\sum_{k=0}^n e + \sum_{k=0}^n f(n, k)$



Abschätzung

Wegen $|f(n, k)| \leq c \cdot k$ folgt:

$$\begin{aligned} & e \cdot (n + 1) + |f(n, 0)| + |f(n, 1)| + \cdots + |f(n, n)| \\ & \leq e \cdot (n + 1) + c \cdot 0 + c \cdot 1 + \cdots + c \cdot n \\ & = e \cdot (n + 1) + c \cdot \frac{n(n+1)}{2} \\ & = c \cdot \frac{n^2}{2} + \left(\frac{c}{2} + e\right) \cdot n + e \\ & \leq d \cdot n^2 \end{aligned}$$

Also: $\sum_{k=0}^n (e + O(k)) \subseteq O(n^2)$,

da die linke Seite eine Menge von Funktionen beschreibt!



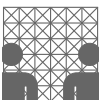
log-Funktion

Definition: Wir schreiben $\log(n)$ anstelle des oft üblichen $\log n$, außer in der Variante $(\log n)$ statt $(\log(n))$, und meinen damit:

$$\log(n) := \begin{cases} 1 & , \text{ falls } n \leq 1 \\ \lfloor \log_2(n) \rfloor + 1 & , \text{ sonst.} \end{cases}$$

... die Länge der Binärdarstellung einer natürlichen Zahl.

Anzahl der Speicherzellen/Schritte ist eine natürliche Zahl.



Effizienz

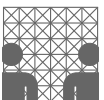
... jetzt haben wir die Grundlagen zum Einordnen von Funktionen (Schranken).

ABER: was wollen wir eigentlich abschätzen?

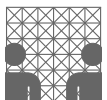
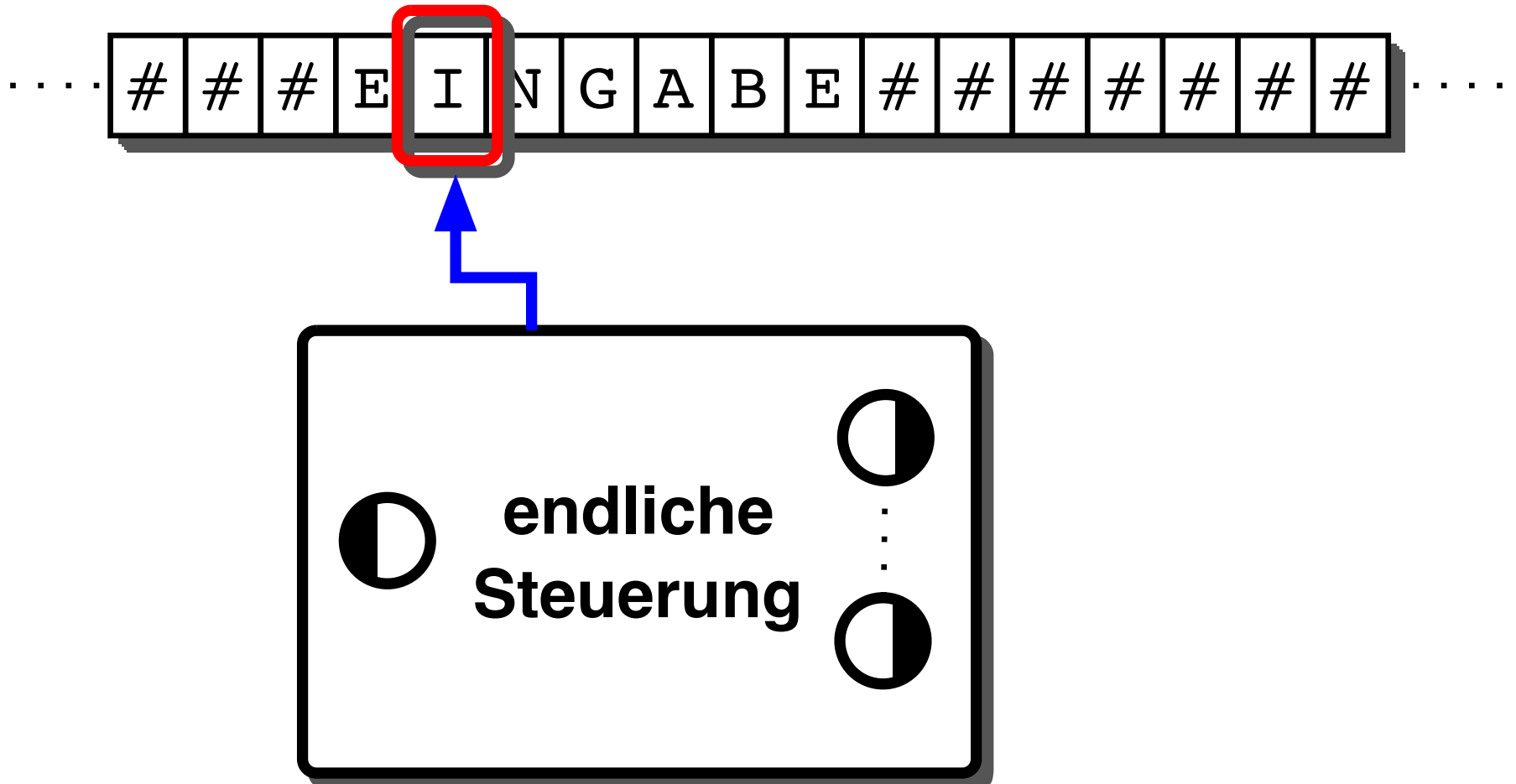
Zeit und Platz müssen formalisiert werden.

→ Turing-Maschine zur *Vereinheitlichung des Zeitbegriffes* (**Rechenschritte**, **Konfigurationsübergänge**) und des *Speicherbedarfes* (**Bandzellen**).

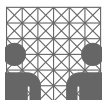
... im Skript: Beispiel ggt-Berechnung



Ausblick



Deterministische Turing-Maschinen (DTM)



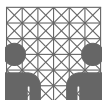
Turing-Machine

Wir suchen ein Modell zur formalen Definition

- der **Berechenbarkeit** von Funktionen und
- deren **Zeit-** und **Platzbedarf**.

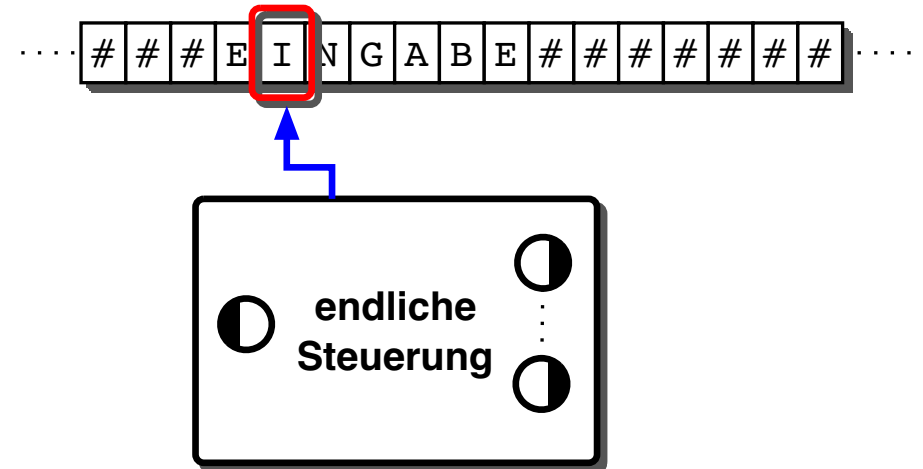
⇒ Verschiedene Modelle existieren:

- Turing-Maschine
- λ -Definierbarkeit
- μ -Rekursivität
- WHILE-Programme
- . . .

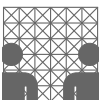


TM allgemein

... schematische Darstellung einer TM mit beidseitig unendlichem Band:



- Wie sehen die Kanten aus?
- Was kann mit dem Arbeitsband passieren?
- Was ist die Akzeptierbedingung?
- Wird etwas ausgegeben? (wichtig für die Definition der Berechenbarkeit!)

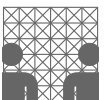


Definition: DTM

Eine **deterministische Turing-Maschine** (DTM) wird beschrieben durch $A := (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$, wobei:

- Z endliche Menge *(Zustände)*
- Γ endliche Menge *(Bandalphabet)*
- Σ endliche Menge *(Eingabealphabet)*
mit $\Sigma \subsetneq \Gamma$, und $\Gamma \cap Z = \emptyset$
- $\# \in \Gamma \setminus \Sigma$ *(Symbol für das unbeschriebene Feld)*

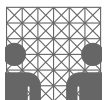
... **gemeint:** jedes „Nicht-Eingabe-Feld“ mit $\#$ initialisiert.



Definition: DTM (Forts.)

Eine **deterministische Turing-Maschine** (DTM) wird beschrieben durch $A := (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$, wobei:

- $q_0 \in Z$ (Startzustand)
- $Z_{\text{end}} \subseteq Z$ (Endzustände)
- $\delta : (Z \times \Gamma) \rightarrow (\Gamma \times \{L, R, H\} \times Z)$ (Übergangsfunktion)
 - ... bestimmt zum **aktuellen Zustand und Bandsymbol** den **Folgezustand** mit **neuer Bandinschrift/Kopfposition**.

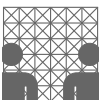


Konfigurationen einer DTM

Allgemein wird ein Systemzustand vollständig beschrieben durch:

- Spezifikation der Struktur des Systems
(die DTM)
- Angabe des Systemzustandes
(der Zustand der endlichen Steuerung und die Kopfposition)
- Angabe des Speicherinhaltes
(die Bandinschrift)

⇒ Konfiguration einer DTM: $(u, q, v) \in \Gamma^* \times Z \times \Gamma^*$

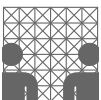


Eigenschaften der DTM

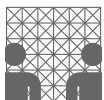
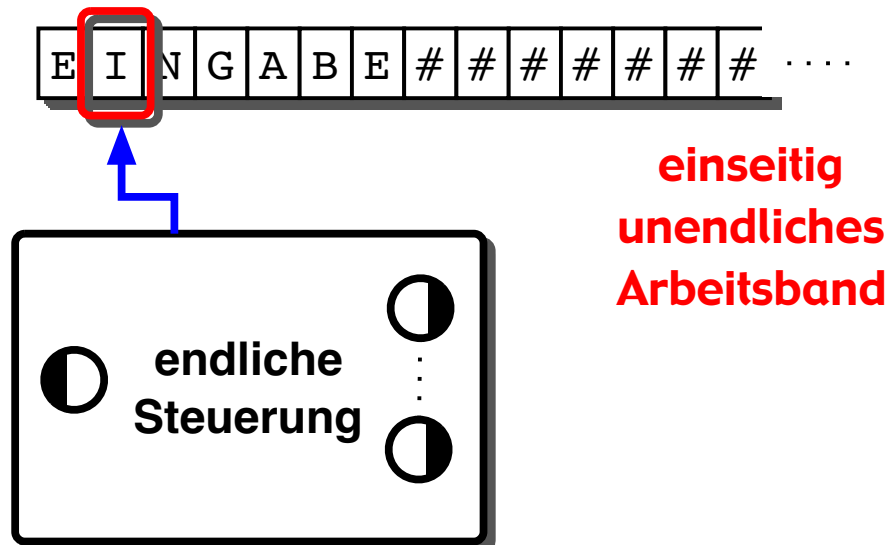
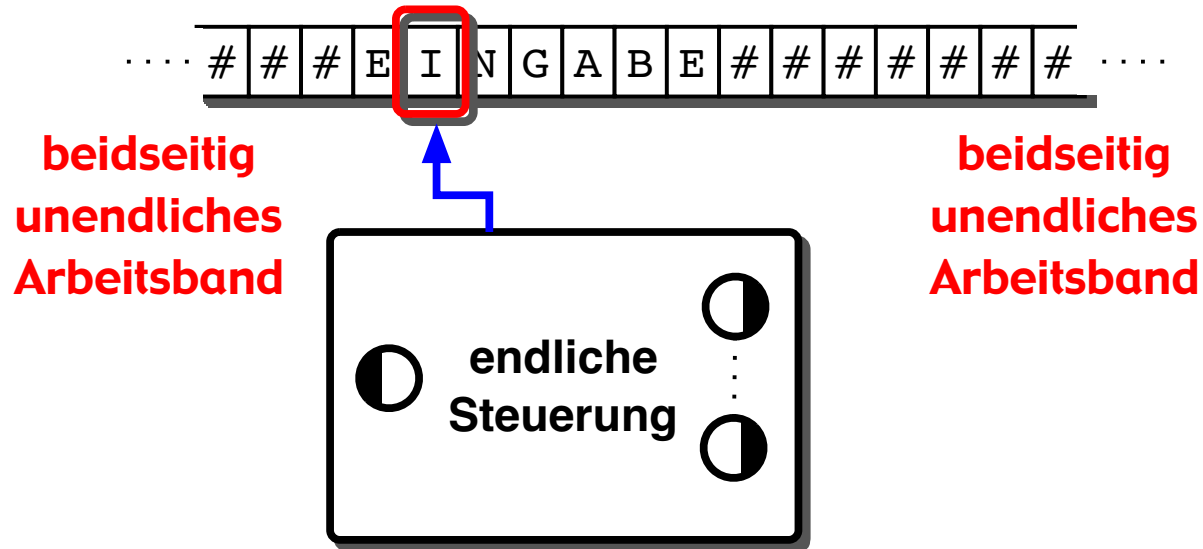
- Die Übergangsfunktion darf partiell sein (d.h. sie darf Definitionslücken enthalten)!
- Folgezustände/Folgekonfigurationen sind eindeutig bestimmt.
- Es gibt genau einen Startzustand.

Es gibt Varianten, die sich bzgl. der Berechenbarkeit als äquivalent erweisen:

- einseitig unendliches Band
- beidseitig unendliches Band
- mehrere Arbeitsbänder

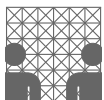
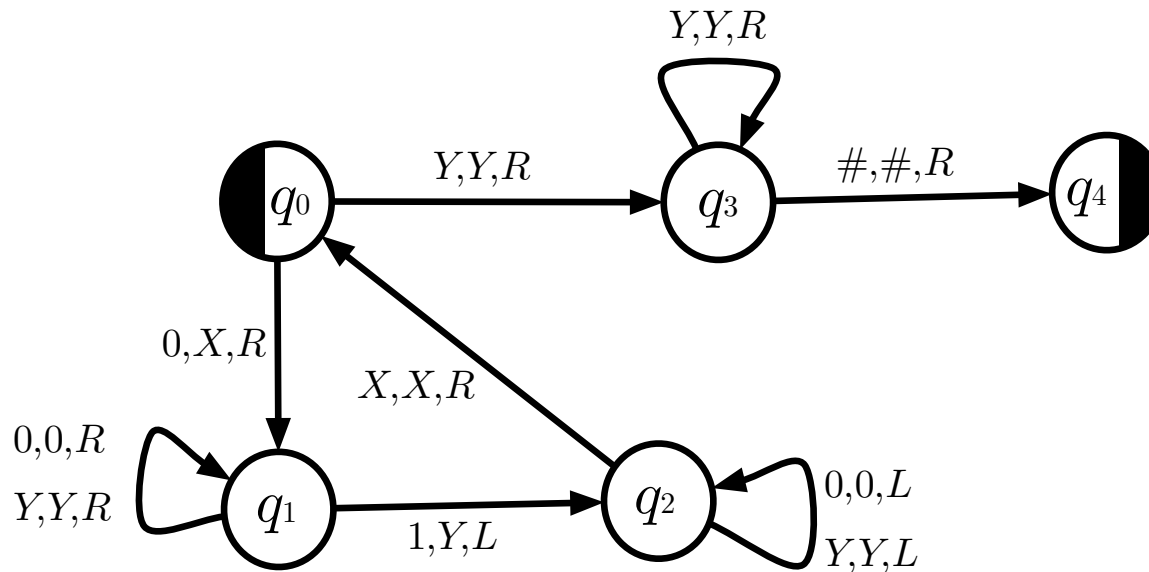


TM-Varianten

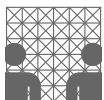
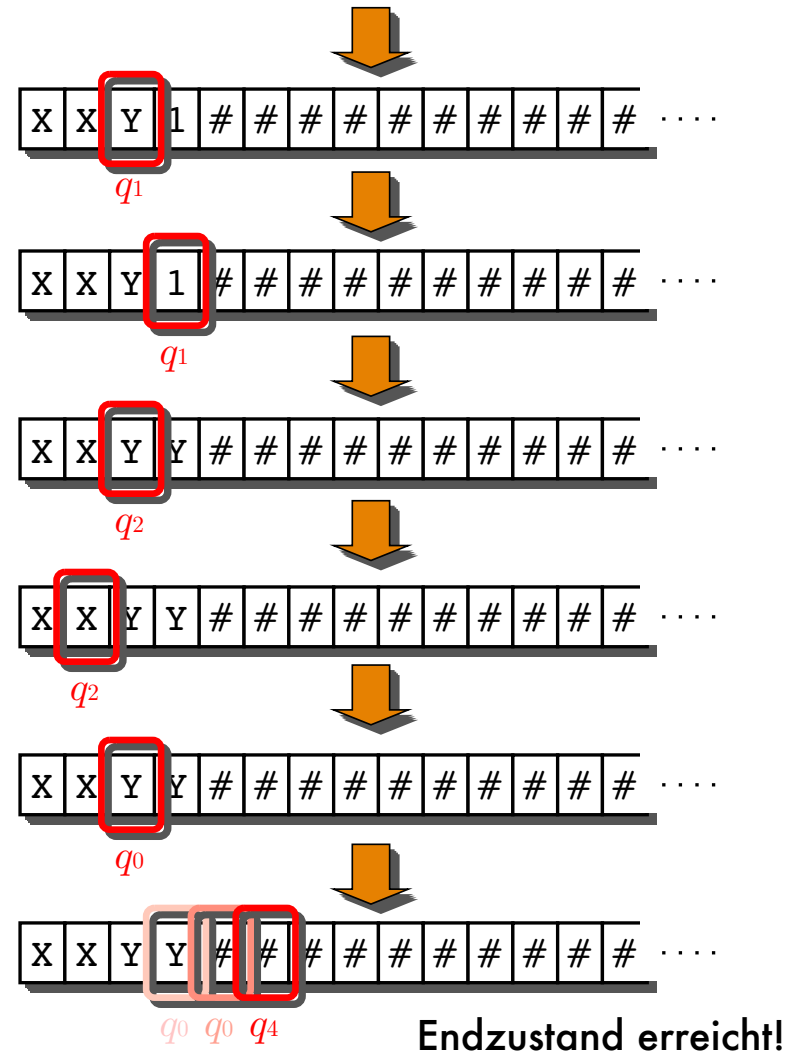
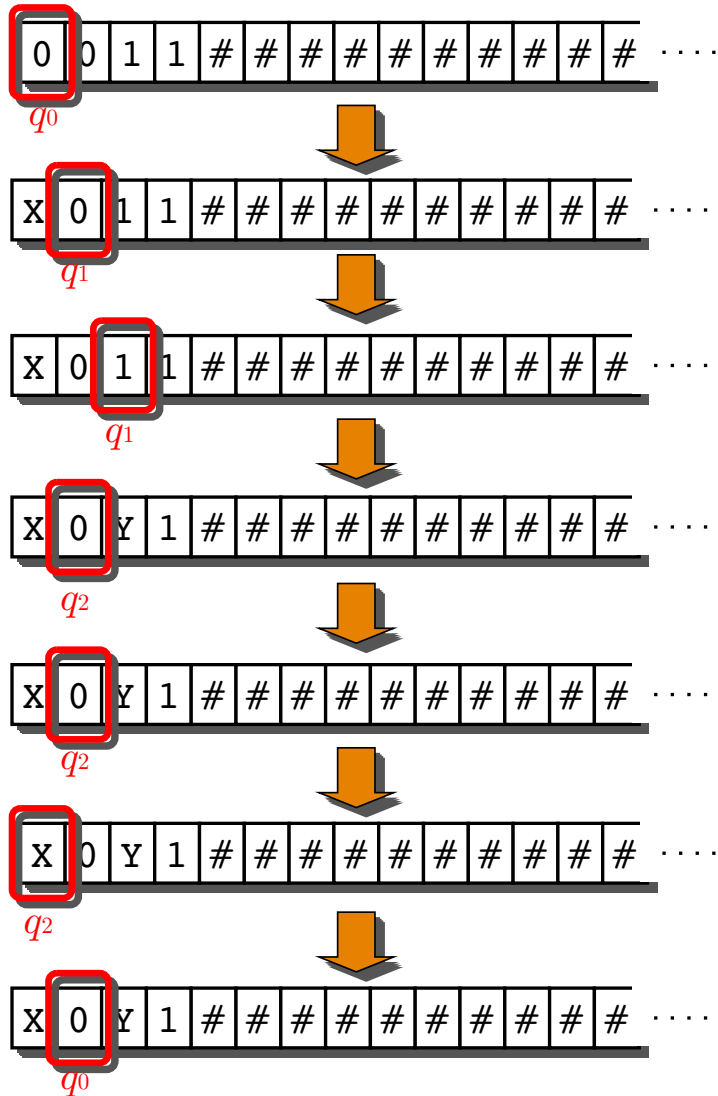


Beispiel: $0^n 1^n$

Zustand	0	1	X	Y	#
q_0	(X, R, q_1)	—	—	(Y, R, q_3)	—
q_1	$(0, R, q_1)$	(Y, L, q_2)	—	(Y, R, q_1)	—
q_2	$(0, L, q_2)$	—	(X, R, q_0)	(Y, L, q_2)	—
q_3	—	—	—	(Y, R, q_3)	$(\#, R, q_4)$
q_4	—	—	—	—	—



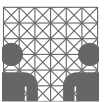
Konfigurationsübergänge



Definition: Konfiguration

$w \in \Gamma^* \cdot Z \cdot \Gamma^*$ heißt **Konfiguration** gdw.

- $w = upv$ mit $u, v \in \Gamma^*$ und $p \in Z$:
- A befindet sich im Zustand p , die Bandinschrift ist uv und der Kopf steht auf dem ersten Zeichen von v . Falls $v = \lambda$, so ist $\#$ unter dem Kopf.
- Falls $v \neq \lambda$, dann ist $v \in \Gamma^* \cdot (\Gamma \setminus \{\#\})$, d.h. ganz rechts in v steht nicht das Symbol $\#$ und rechts von v sind nur noch $\#$'s auf dem Band.
- Entsprechend für u : Falls $u \neq \lambda$, dann ist $u \in (\Gamma \setminus \{\#\})\Gamma^*$, d.h. ganz links von u steht nicht $\#$ und links von u sind nur noch $\#$'s.



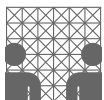
Konfigurationsübergang

■ $KONF_M$ bezeichnet die **Menge aller Konfigurationen** der Turing-Maschine M .

■ Für eine DTM A ist die **Schrittrelation** $\vdash_A \subseteq KONF_A \times KONF_A$ erklärt durch:

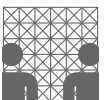
$w \vdash_A w'$ gilt genau dann, wenn für $w = uypxv$ mit $u, v \in \Gamma^*$, $x, y, z \in \Gamma$ und $p, q \in Z$, folgendes gilt:

$$w' = \begin{cases} uqyzv, & \text{falls } \delta(p, x) = (z, L, q) \\ uyzqv, & \text{falls } \delta(p, x) = (z, R, q) \\ uyqzv, & \text{falls } \delta(p, x) = (z, H, q) \end{cases}$$



Weitere Begriffe

- Mit \vdash_A^* wird die reflexive, transitive Hülle von \vdash_A bezeichnet.
- Folgen von Konfigurationen $k_1, k_2, k_3, \dots, k_i, k_{i+1}, \dots$, die in der Relation $k_1 \vdash k_2 \vdash \dots \vdash k_i \vdash k_{i+1} \vdash \dots$ stehen, heißen **Rechnungen**.
- Endliche Rechnungen $k_1 \vdash k_2 \vdash \dots \vdash k_t$ heißen **Erfolgsrechnungen**, wenn $k_1 \in \Gamma^* \cdot \{q_0\} \cdot \Gamma^*$ und $k_t \in \Gamma^* \cdot Z_{\text{end}} \cdot \Gamma^*$ ist.



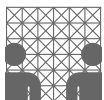
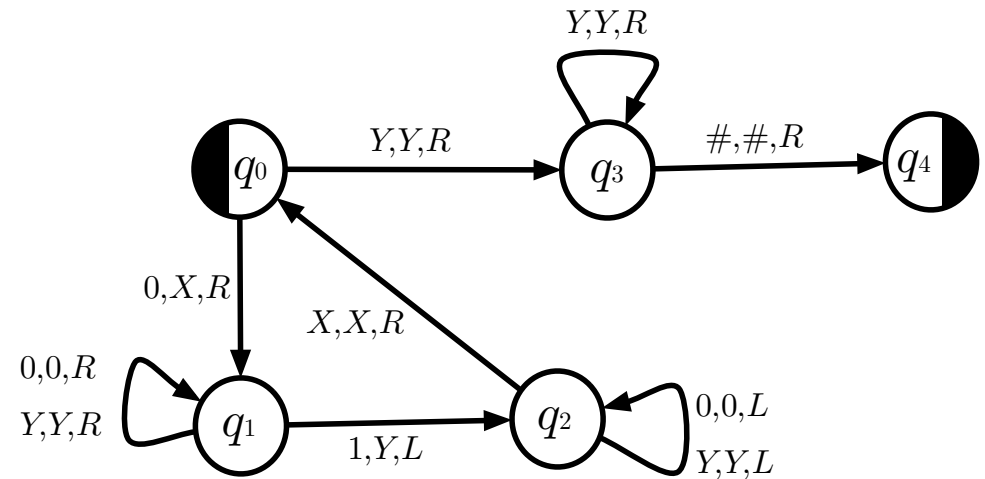
Definition: akzeptierte Sprache

Für die DTM $A = (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$ bezeichnet $L(A)$ die **von A akzeptierte Sprache**:

$$L(A) := \{w \in \Sigma^* \mid \exists u, v \in \Gamma^* \exists q \in Z_{\text{end}} : q_0 w \xrightarrow{*} uqv\}$$

Für das obige
Beispiel:

$$L(A) = \{0^n 1^n \mid n \in \mathbb{N}\}$$



Def.: Turing-Berechenbarkeit

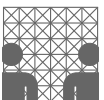
Sei Σ ein Alphabet.

Eine (Wort-)Funktion $f : \Sigma^* \xrightarrow[p]{\quad} \Sigma^*$ heißt

(Turing-)berechenbar oder **partiell rekursiv** gdw.
es eine DTM $A = (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$ gibt mit:

$q_0 w \xrightarrow[A]{*} q_e v$, für $q_e \in Z_{\text{end}}$ gdw. $f(w) = v$ ist.

... der Funktionswert steht *ab der Kopfposition als einziges* (abgesehen von #'s) auf dem Band!



Definition: Berechenbarkeit auf \mathbb{N}

Seien $r, s \in \mathbb{N}$ und $r, s \geq 1$.

Eine (partielle) Funktion $f : \mathbb{N}^r \xrightarrow{p} \mathbb{N}^s$ heißt

(Turing-)berechenbar oder **partiell rekursiv** gdw. eine DTM $A = (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$ existiert mit

$$q_0 0^{m_1+1} 1 \dots 10^{m_r+1} \xrightarrow[A]{*} p 0^{n_1+1} 10^{n_2+1} 1 \dots 10^{n_s+1}$$

und $p \in Z_{\text{end}}$, sowie

$$f(m_1, m_2, \dots, m_r) = (n_1, n_2, \dots, n_s)$$

gdw. der Funktionswert definiert ist.



einige berechenbare Funktionen

■ $f : \mathbb{N} \rightarrow \mathbb{N}$ mit $f(x) := x + 1$ (Nachfolger)

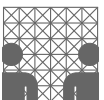
■ $f : \{1\}^* \rightarrow \{0, 1\}^*$ mit $f(w) := v$ mit $[w]_1 = [v]_2$
(unär→binär Konversion)

■ $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ mit $f(x, y) := x + y$ (Summe)

■ $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ mit $f(x, y) := x \cdot y$ (Produkt)

■ $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ mit $f(x, y) := x^y$
(Exponentiation)

... aber nicht alle Funktionen sind berechenbar!!!



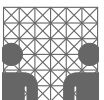
Existenz nicht berechenbarer Fkt.

- DTM als endliche Zeichenkette darstellbar
⇒ Menge aller DTM abzählbar.
- Ist die Menge aller Funktionen $f : \mathbb{N} \rightarrow \{0, 1\}$ abzählbar? (**Annahme: JA!** ⇒ f_1, f_2, f_3, \dots)
- Definiere $g : \mathbb{N} \rightarrow \{0, 1\}$ durch

$$g(x) := \begin{cases} 0 & \text{falls } f_x(x) = 1 \\ 1 & \text{sonst} \end{cases}$$

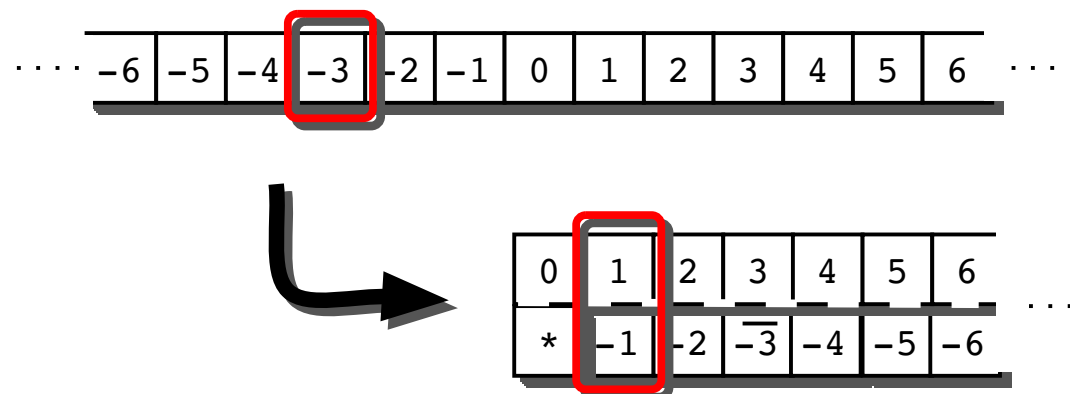
- Dann gilt: $\forall n \in \mathbb{N} : g \neq f_n$ **Widerspruch!!!**

Also muss es nicht berechenbare Funktion geben!

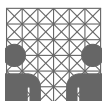


beidseitig / einseitig unendl. Band

- Zwei Turingmaschinen A und B heißen genau dann **äquivalent**, wenn sie die gleiche Sprache akzeptieren, d.h. $L(A) = L(B)$ gilt.
- Zu jeder DTM A mit *beidseitig* unendlichem Arbeitsband gibt es eine äquivalente DTM B mit *einseitig* unendlichem Arbeitsband und umgekehrt.



Beweisidee:
„Spuren-
bildung“

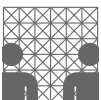


Darstellungsvarianten für TM

- graphisch als Zustandsübergangdiagramm
- als Tabelle
- als Turingtafel (δ als Relation
 $\subseteq Z \times Y \times Y \times \{L, H, R\} \times Z$ geschrieben)

... sind alles Möglichkeiten zur Darstellung ein und desselben mathematische Modells!

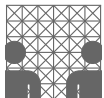
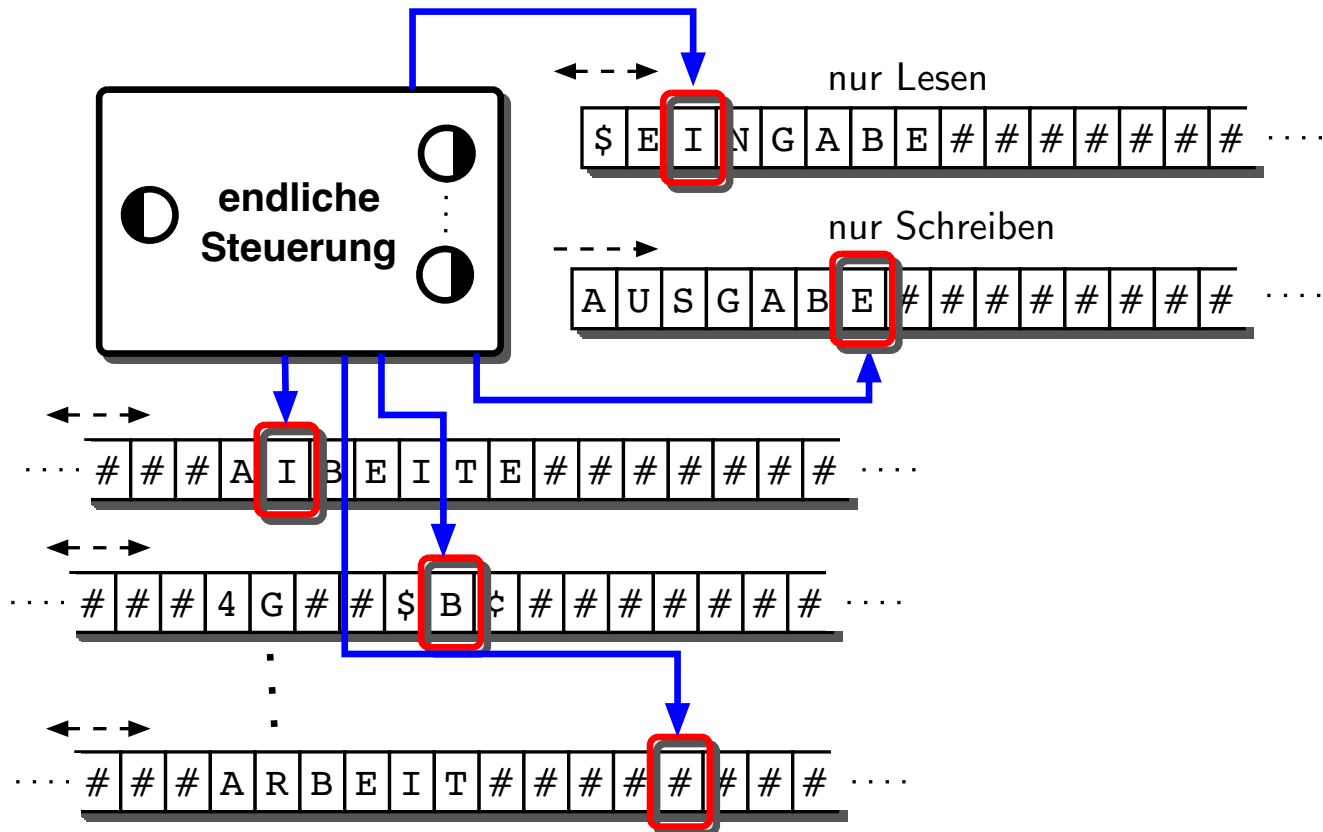
... und dann gibt es noch die „höhersprachliche“ Variante mit mehreren Bändern ...



Mehr-Band = 1-Band

... dieselbe Idee führt zum Ergebnis:

Zu jeder deterministischen k -Band off-line Turing-Maschine A mit $k \geq 1$ gibt es eine äquivalente DTM B mit nur einem Band.

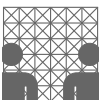


nichtdeterministische TM

$M = (Z, \Sigma, \Gamma, K, q_0, Z_{\text{end}})$ heißt
nichtdeterministische Turingmaschine (NTM),
wenn zu jedem Paar $(p, y) \in Z \times \Gamma$ eine endliche
Zahl von Übergängen möglich ist:

$$\delta : Z \times \Gamma \longrightarrow 2^{\Gamma \times \{L,R,H\} \times Z}$$

Alles andere, wie bei der DTM.



NTM-Übergangsrelation

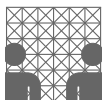
Alternativ zu δ : Kanten als Relation K

$$K \subseteq Z \times \Gamma \times \Gamma \times \{L, R, H\} \times Z$$

...zu einem Zustand q und einem Symbol x unter dem Kopf kann es mehrere verschiedene Möglichkeiten für die Folgekonfiguration geben.

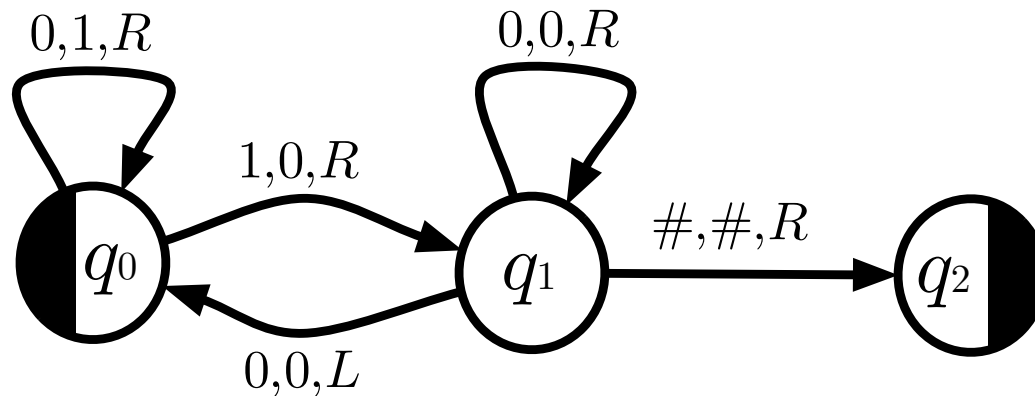
... zum Beispiel:

(q, A, A, R, q') und $(q, A, \#, H, q'')$ könnten beide in K sein!



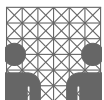
Beispiel: NTM

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, \#\}, \delta, q_0, \#, \{q_2\})$$



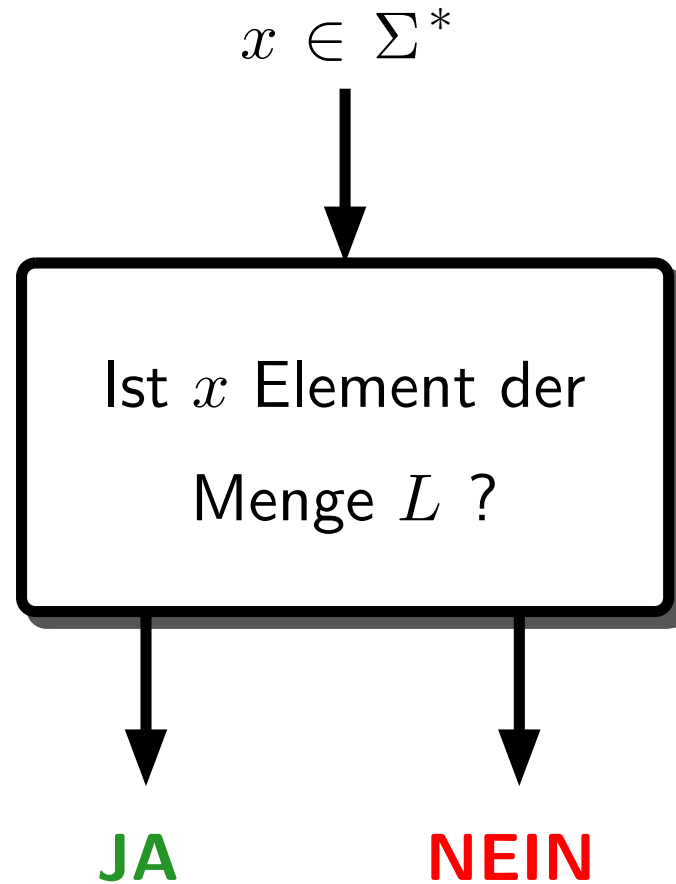
Zustand	0	1	#
q_0	$\{(1, R, q_0)\}$	$\{(0, R, q_1)\}$	\emptyset
q_1	$\{(0, R, q_1), (0, L, q_0)\}$	$\{(1, R, q_1), (1, L, q_0)\}$	$\{(\#, R, q_2)\}$
q_2	\emptyset	\emptyset	\emptyset

Aufgabe: Welche Konfigurationen sind bei Eingabe von 01 (bzw. 100) erreichbar?



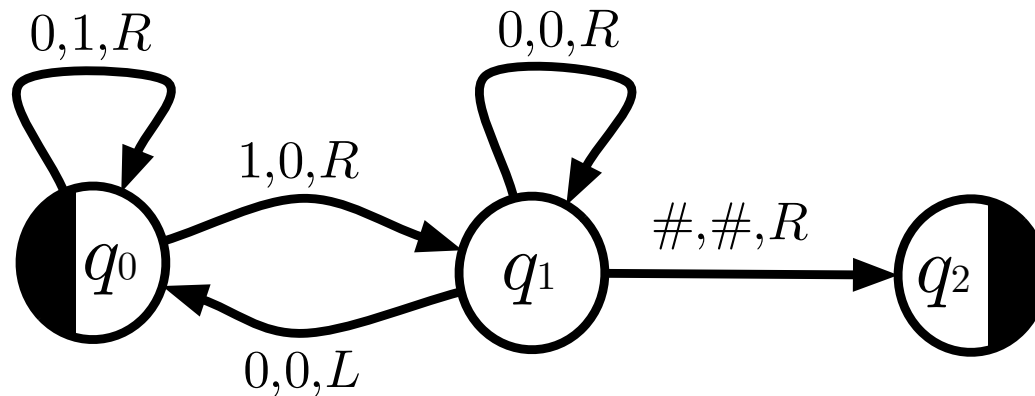
Ausblick

Entscheidbarkeit / Beweisbarkeit



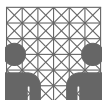
Beispiel: NTM

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, \#\}, \delta, q_0, \#, \{q_2\})$$

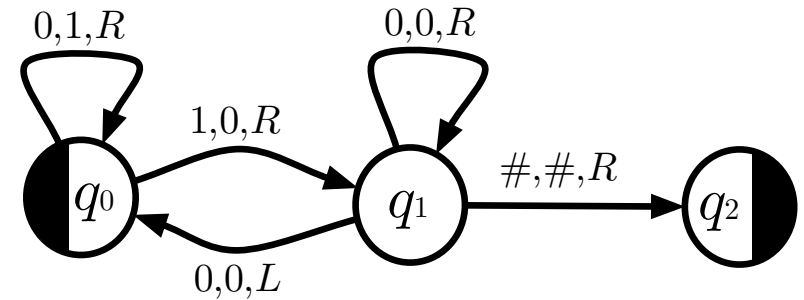


Zustand	0	1	#
q_0	$\{(1, R, q_0)\}$	$\{(0, R, q_1)\}$	\emptyset
q_1	$\{(0, R, q_1), (0, L, q_0)\}$	$\{(1, R, q_1), (1, L, q_0)\}$	$\{(\#, R, q_2)\}$
q_2	\emptyset	\emptyset	\emptyset

Aufgabe: Welche Konfigurationen sind bei Eingabe von 01 (bzw. 100) erreichbar?



Beispiel: NTM-Konfigurationen



Eingabe: 01 (bzw. 100)

.....

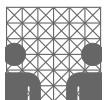
$$[q_0]01 \vdash_M 1 [q_0]1 \vdash_M 10 [q_1] \vdash_M 10\# [q_2]$$

$$[q_0]100 \vdash_M 0 [q_1]00 \vdash_M 00 [q_1]0 \vdash_M 000 [q_1] \vdash_M 000\# [q_2]$$

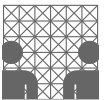
$$[q_0]100 \vdash_M 0 [q_1]00 \vdash_M [q_0]000 \vdash_M 1 [q_0]00 \vdash_M 11 [q_0]0 \vdash_M 111 [q_0]$$

$$[q_0]100 \vdash_M 0 [q_1]00 \vdash_M 00 [q_1]0 \vdash_M 0 [q_0]00 \vdash_M 01 [q_0]0 \vdash_M 011 [q_0]$$

$$[q_0]100 \vdash_M 0 [q_1]00 \vdash_M 00 [q_1]0 \vdash_M 000 [q_1] \vdash_M 00 [q_0]0 \vdash_M 001 [q_0]$$



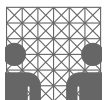
**Entscheidbare
vs.
rekursiv aufzählbare
Mengen**



Akzeptierung/Berechnung

... ein kurzer Rückblick:

- Sprache L wird durch eine TM M akzeptiert gdw. **nur** Wörter aus L eine Erfolgsrechnung auf M haben.
- Funktion f wird von DTM M berechnet, gdw. M für **jede** Eingabe w in einer Endkonfiguration $q_e f(w)$ hält bzw. nicht hält, falls $w \notin \text{Def}(f)$.

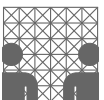
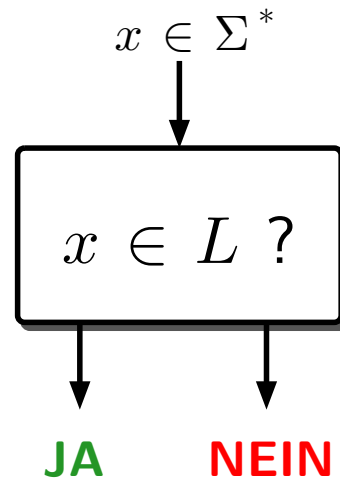


Definition: rekursiv

Eine Menge $L \subseteq \Sigma^*$ heißt (relativ zu Σ^*) **entscheidbar** oder **rekursiv** gdw. ihre charakteristische Funktion $\chi_L : \Sigma^* \rightarrow \{0, 1\}$ berechenbar ist.

Die *Klasse aller entscheidbaren Mengen* wird mit \mathcal{REC} (*recursive sets*) bezeichnet.

Entscheidbarkeit

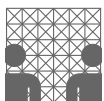
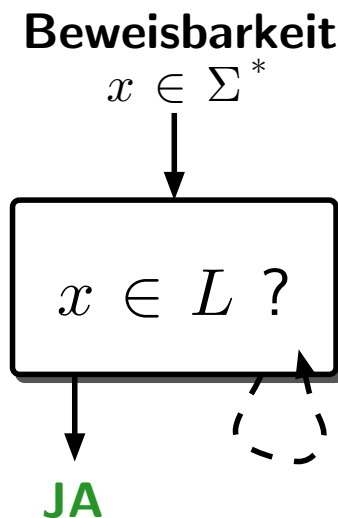


Definition: rekursiv aufzählbar

Eine Menge $L \subseteq \Sigma^*$ heißt **beweisbar** oder **rekursiv aufzählbar** gdw. $L = \emptyset$ ist, oder eine totale Turing-berechenbare Funktion $g : \mathbb{N} \rightarrow \Sigma^*$ existiert, für die $g(\mathbb{N}) = L$ ist.

Die *Klasse aller aufzählbaren Mengen* wird mit \mathcal{RE} (*recursively enumerable sets*) bezeichnet.

... oder
anders dar-
gestellt:



Eigenschaften

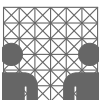
Theorem: L ist rekursiv aufzählbar gdw. eine TM M mit $L = L(M)$ existiert. **Beweisidee:** $L = L(M) \Rightarrow \exists$ TM A , die alle Wörter aus L auf die Ausgabe schreibt:

- Generiere sukzessive Paare aus $(i, j) \in \mathbb{N}^2$.
- Simuliere M auf dem i -ten Wort w_i .
- Falls M nach j Schritten hält, schreibe w_i und ein Trennsymbol.

\exists TM A , die alle Wörter aus L schreibt $\Rightarrow L = L(M)$

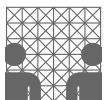
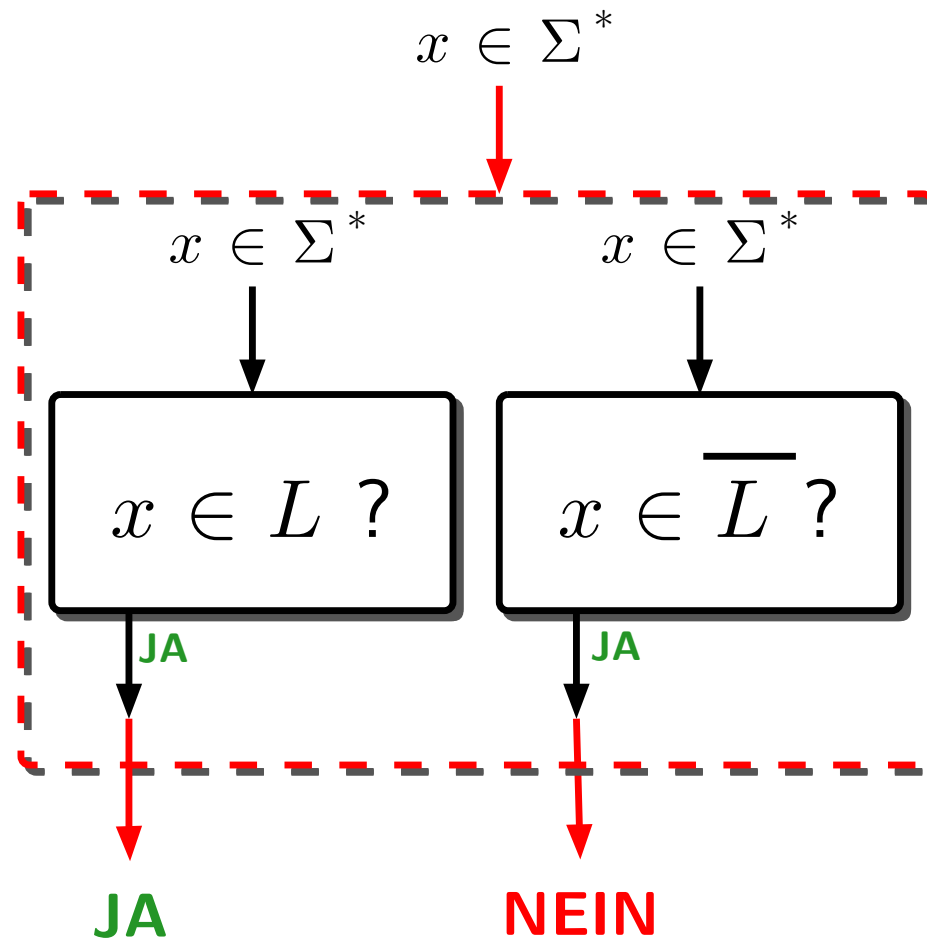
- Verwende Zähler i
- Kontrolliere, ob w_i (Ausgabe von A) das Eingabewort w ist? Ja: akzeptiere! Nein: inkrementiere i .

Analog: L rekursiv aufzählbar $\Rightarrow \exists$ TM $M : L = L(M)$



Eigenschaften

Theorem: Ist eine Menge L und ihr Komplement \bar{L} rekursiv aufzählbar, so ist L auch rekursiv.



Hierarchie

Folgende Beziehungen wollen wir zeigen:

Klasse der **regulären Mengen**

\subsetneq Klasse der **kontextfreien Mengen**

\subsetneq Klasse der **kontextsensitiven Mengen**

\subsetneq Klasse der **entscheidbaren Mengen**

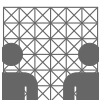
\subsetneq Klasse der **aufzählbaren Mengen**

\subsetneq Klasse der **abzählbaren Mengen**

\neq Klasse der **überabzählbaren Mengen**

Existenz überabzählbarer Mengen ist bekannt.

(z.B. aus Diagonalbeweis).



Abschlusseigenschaften

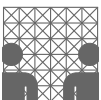
Theorem: Die Klasse der entscheidbaren Mengen bildet eine *boolesche Mengen-Algebra*.

Beweisidee: Z.z. ist der Abschluss gegen

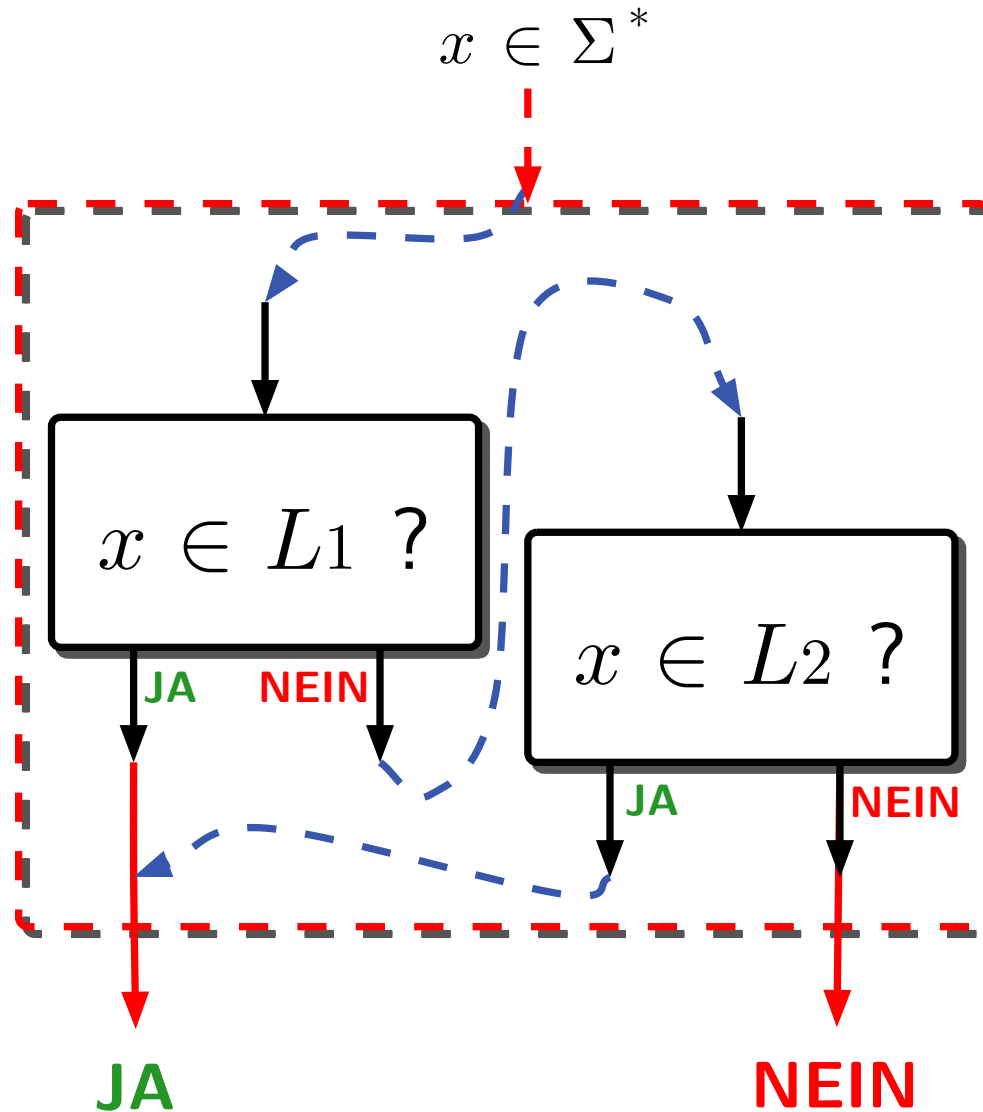
- Komplementbildung,
- Durchschnitt,
- Vereinigung.

Komplementabschluss:

... einfach JA und NEIN vertauschen.

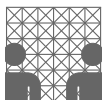
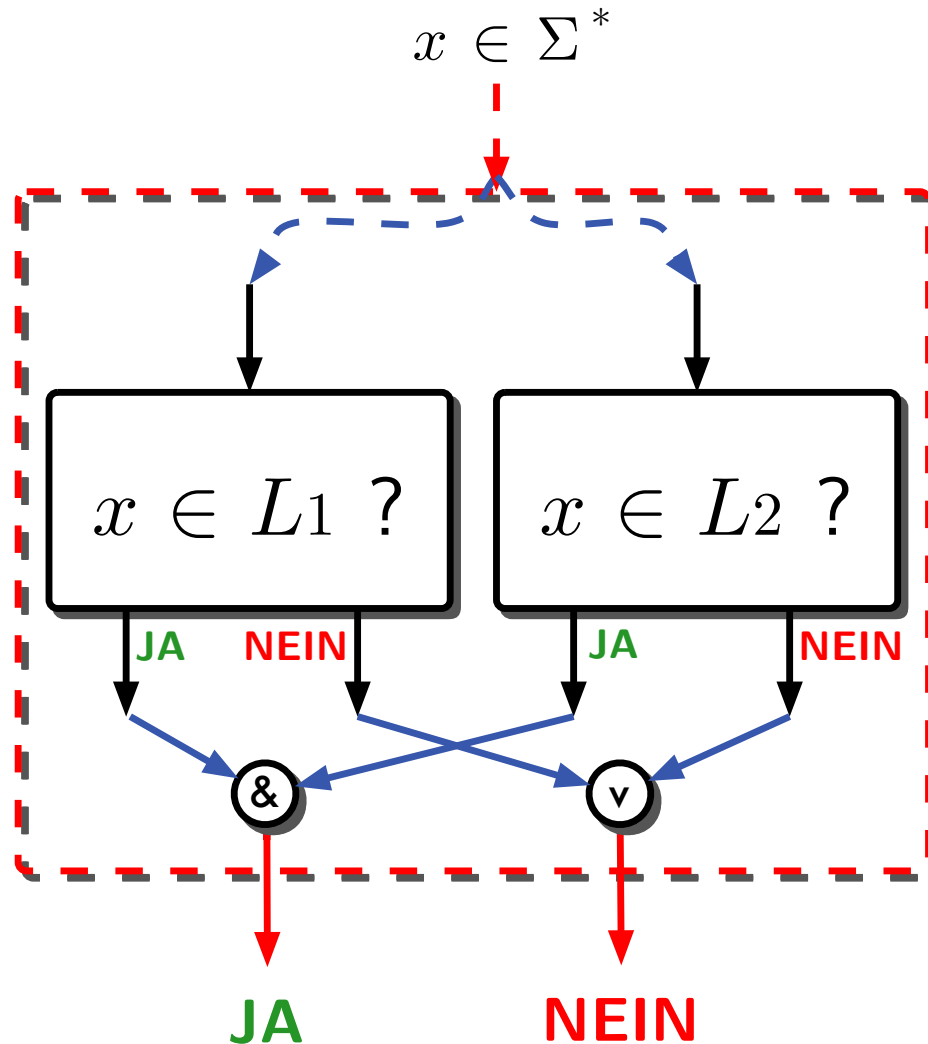


Vereinigung

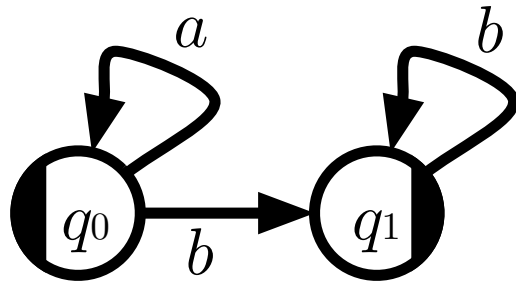


Durchschnitt

... folgt nach De Morgan, oder direkt:



Gödelisierung von FA's



... wird kodiert durch:

a	x
b	xx
q_0	z
q_1	zz

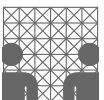
Kante (q_0, b, q_1) :

-ZXXZZ

Insgesamt:

-ZXZ-ZXXZZ+ZZXXZZ

... ähnlich auch für Turing-Maschinen!



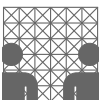
nicht aufzählbare Menge

- Wir zeigen, dass es Mengen gibt, die nicht rekursiv aufzählbar sind:

Theorem: Sei G ein Alphabet zur Kodierung von Turing-Maschinen bzw. Wörtern, sowie w_i das i -te Wort und M_i die i -te DTM in der lexikalischen Aufzählung der Wörter $\langle M_i \rangle \in G^*$.

Dann ist die Menge $L_d := \{w_i \mid w_i \notin L(M_i)\}$ nicht aufzählbar.

(Selbstanwendbarkeitsproblem)



Beweis: $L_d \notin \mathcal{RE}$

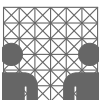
Charakteristische Funktion für $L(M_i)$ und L_d :

	w_0	w_1	w_2	\dots	w_n	\dots
M_0	0	1	1	\dots	0	\dots
M_1	1	1	0	\dots	1	\dots
M_2	1	0	0	\dots	1	\dots
\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
M_n	1	1	0	\dots	1	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots
L_d	1	0	1	\dots	0	\dots

Also: $\forall i \in \mathbb{N} : w_i \in L(M_i) \iff w_i \notin L_d$.

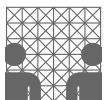
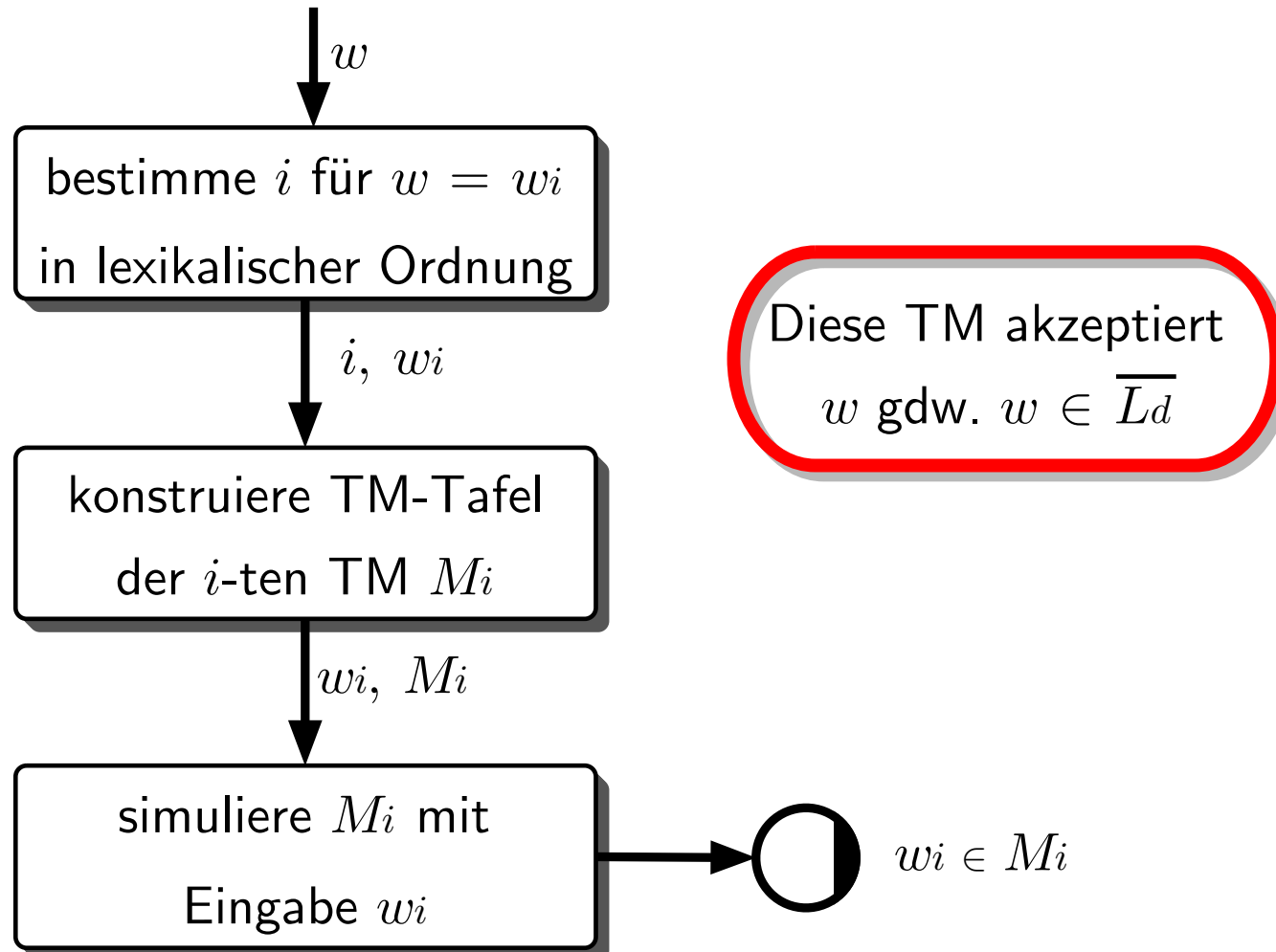
Somit $\forall i \in \mathbb{N} : L(M_i) \neq L_d$

und L_d nicht aufzählbar!



Das Komplement von L_d

... ist aufzählbar:

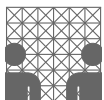


Definition: UTM

- Sei M eine DTM mit Anfangskonfiguration k_0 .
- Sei $\langle M \rangle$ die Kodierung über dem Alphabet $G := \{0, 1\}$ von M und $\langle k_0 \rangle$ jene von k_0 .

Eine DTM $U := (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$ heißt **universelle Turing-Maschine (UTM)**, wenn für die Anfangskonfiguration $q_0 \langle M \rangle \langle k_0 \rangle$ gilt:

- $k_i \xrightarrow{M} k_j \Rightarrow \langle A \rangle \langle k_i \rangle \xrightarrow{U}^* \langle A \rangle \langle k_j \rangle$, wobei hier nur die relevante Bandinschrift von U gemeint ist, ohne die Stellungen ihres LSK und der Zustände.

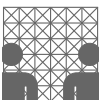


Existenz einer UTM

Definition der **universellen Turing-Maschine** macht noch keine Aussage darüber, ob so etwas auch tatsächlich existiert!

Lemma: Es gibt universelle Turing-Maschinen.

- Lesen der Kodierung einer TM und eines Eingabewortes,
- Simulation der Einzelschritte der kodierten TM.
- hier **kein Beweis** (durch Angabe einer UTM)



Das Halteproblem

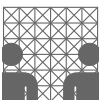
Das Halteproblem lautet:

Gegeben:	Ein Computerprogramm P und ein Eingabe x für P .
Gesucht:	Kommt P für x jemals in einen Stop- bzw. Endzustand?
Antwort:	?

- Darstellung als Menge:

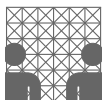
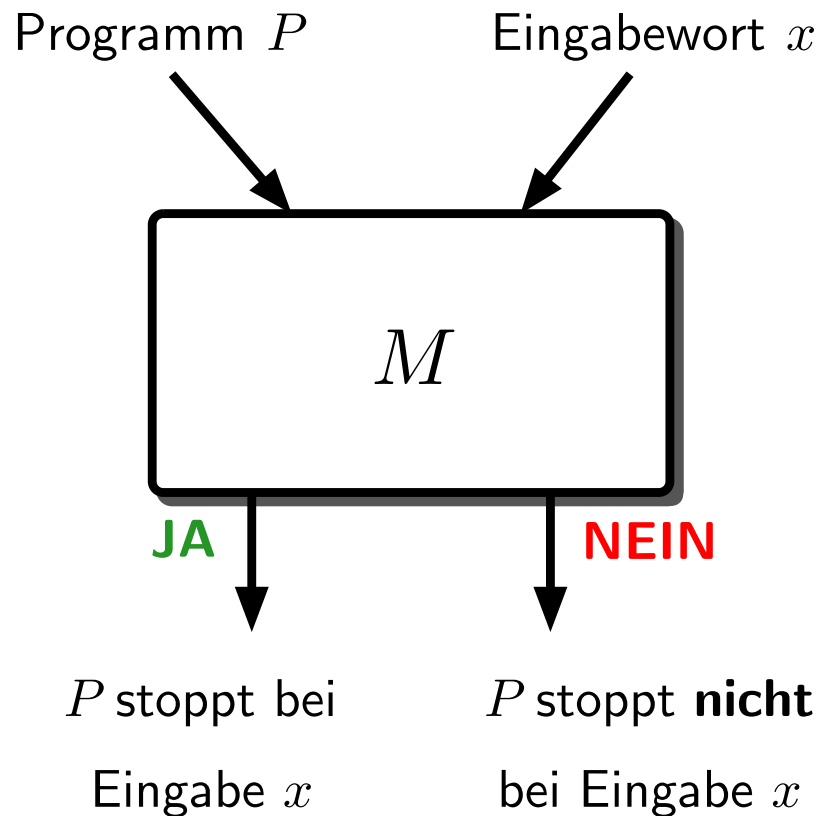
$$H = \{ \langle M \rangle \langle w \rangle \mid \text{TM } M \text{ hält auf } w \}$$

- Wichtig zur Erkennung von Endlosschleifen!
- **Aber:** H ist unentscheidbar.



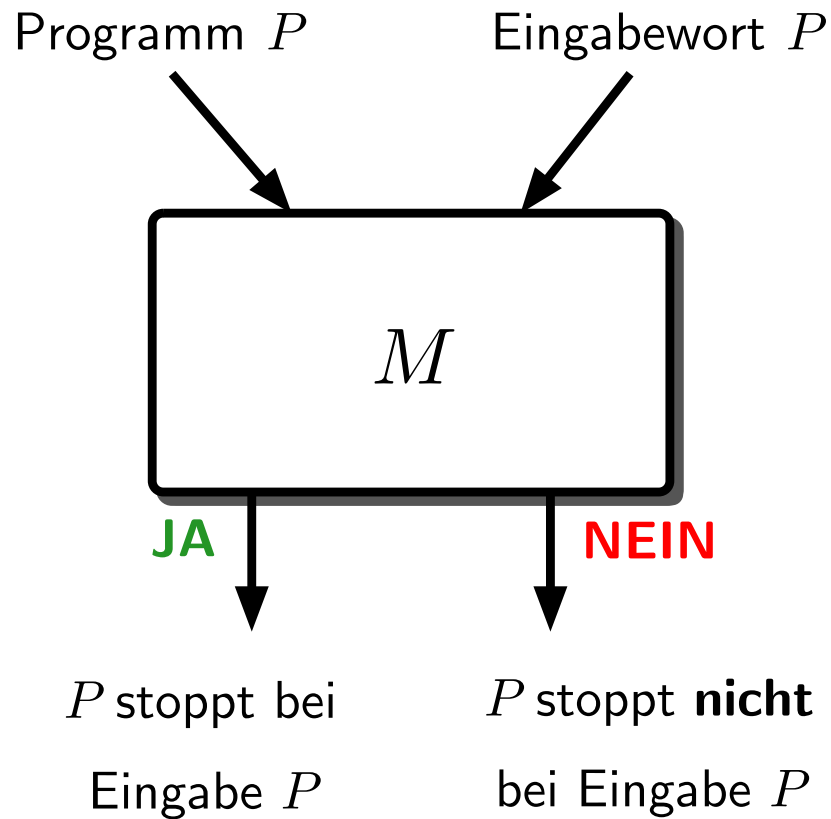
Beweis: Halteproblem $\notin \mathcal{REC}$

Erster Beweis: Konstruktion mit Widerspruch
Angenommen M löst das Halteproblem:



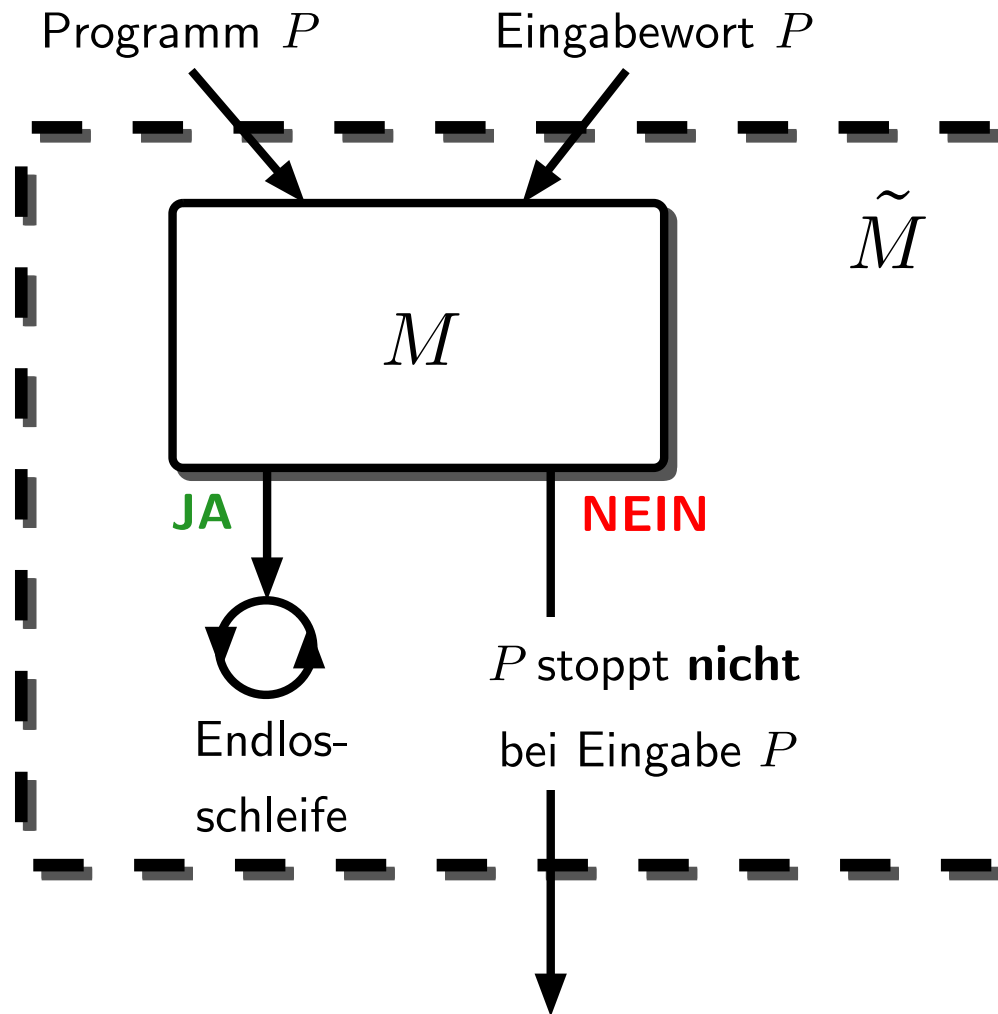
Halteproblem (2)

Eingabewort x ersetzt durch das Programm P :



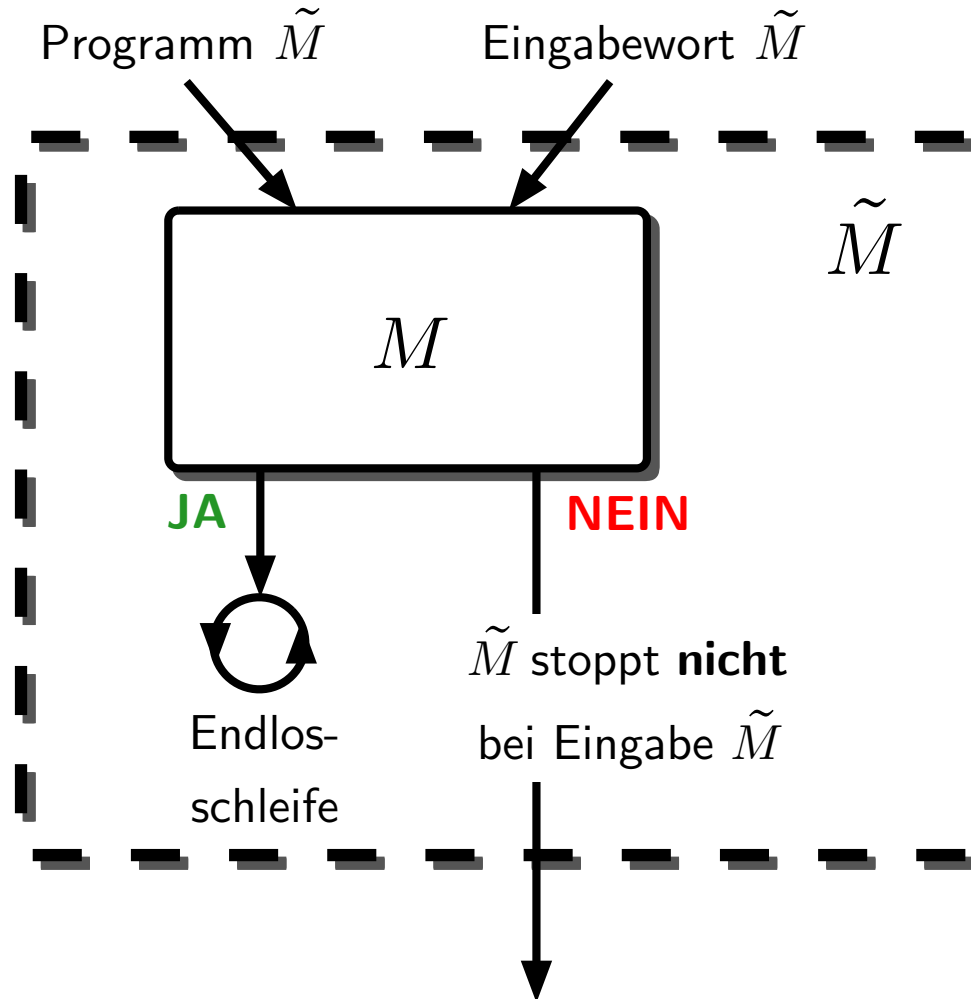
Halteproblem (3)

Neue Maschine \tilde{M} mit Endlosschleife bei JA:



Halteproblem (4)

Als Programm P verwenden wir jetzt \tilde{M} :

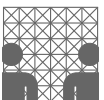


Halteproblem und L_d

- Kodieren wir L_d und H mit dem Alphabet $\{0, 1\}$.
- Die Unentscheidbarkeit des Halteproblems H kann dann durch Reduktion von $L_d^{\{0,1\}}$ auf H gezeigt werden (bei geeigneter Kodierung):

$$w \in L_d^{\{0,1\}} \iff \langle w \rangle \langle w \rangle \notin H$$

ALSO: Wenn H entscheidbar, dann auch $L_d^{\{0,1\}}$.

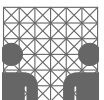


Halteproblem ist aufzählbar

$H = \{ \langle M \rangle \langle w \rangle \mid \text{TM } M \text{ hält auf } w \}$ ist rekursiv aufzählbar.

- Wir nehmen eine UTM.
- Diese simuliert M auf der Eingabe w
 - solange, bis M hält,
 - oder unendlich lange, falls M nicht auf w hält.

Mithin akzeptiert die UTM das Wort $\langle M \rangle \langle w \rangle$ gdw. M auf dem Wort w hält.



Ausblick

- Weitere wichtige Unentscheidbarkeitsresultate
- Weitere Berechenbarkeitsmodelle
- Realistische Maschinenmodelle

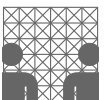


Turingsche These

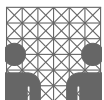
Die Turing-berechenbaren Funktionen sind genau die im intuitiven Sinne berechenbaren Funktionen.

Analog für den λ -Kalkül:

Die Churchsche These: Die λ -definierbaren Funktionen sind genau die im intuitiven Sinne berechenbaren Funktionen.

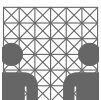


Einige unentscheidbare Probleme



Unentscheidbare Probleme

- schon als unentscheidbar gezeigt:
 - Sprache L_d
 - Komplement von L_d
 - Halteproblem
- außerdem unentscheidbar:
 - einige Kachelprobleme bzw. (2D-)Dominoprobleme
 - Vorkommen einer 1 als Bild einer Funktion
 - Leerheitsproblem für TM
 - 10. Hilbertsches Problem
 - ...

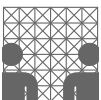


Kachel- / Dominoprobleme

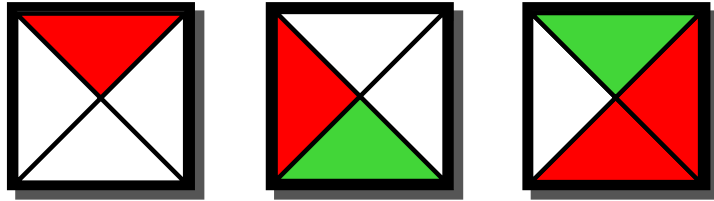
Gegeben: Eine Menge von r Kacheltypen $\mathcal{R} = \{K_1, K_2, \dots, K_r\}$, $n \in \mathbb{N}$ (beliebig viele Kacheln von jedem Typ)

Gesucht: (A) Kann man den ersten Quadranten der euklidischen Ebene korrekt kacheln?
(B) Kann man eine Fläche der Größe $n \times n$ korrekt kacheln?

Antwort: JA oder NEIN



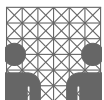
Beispiel: Kachelproblem



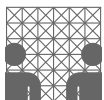
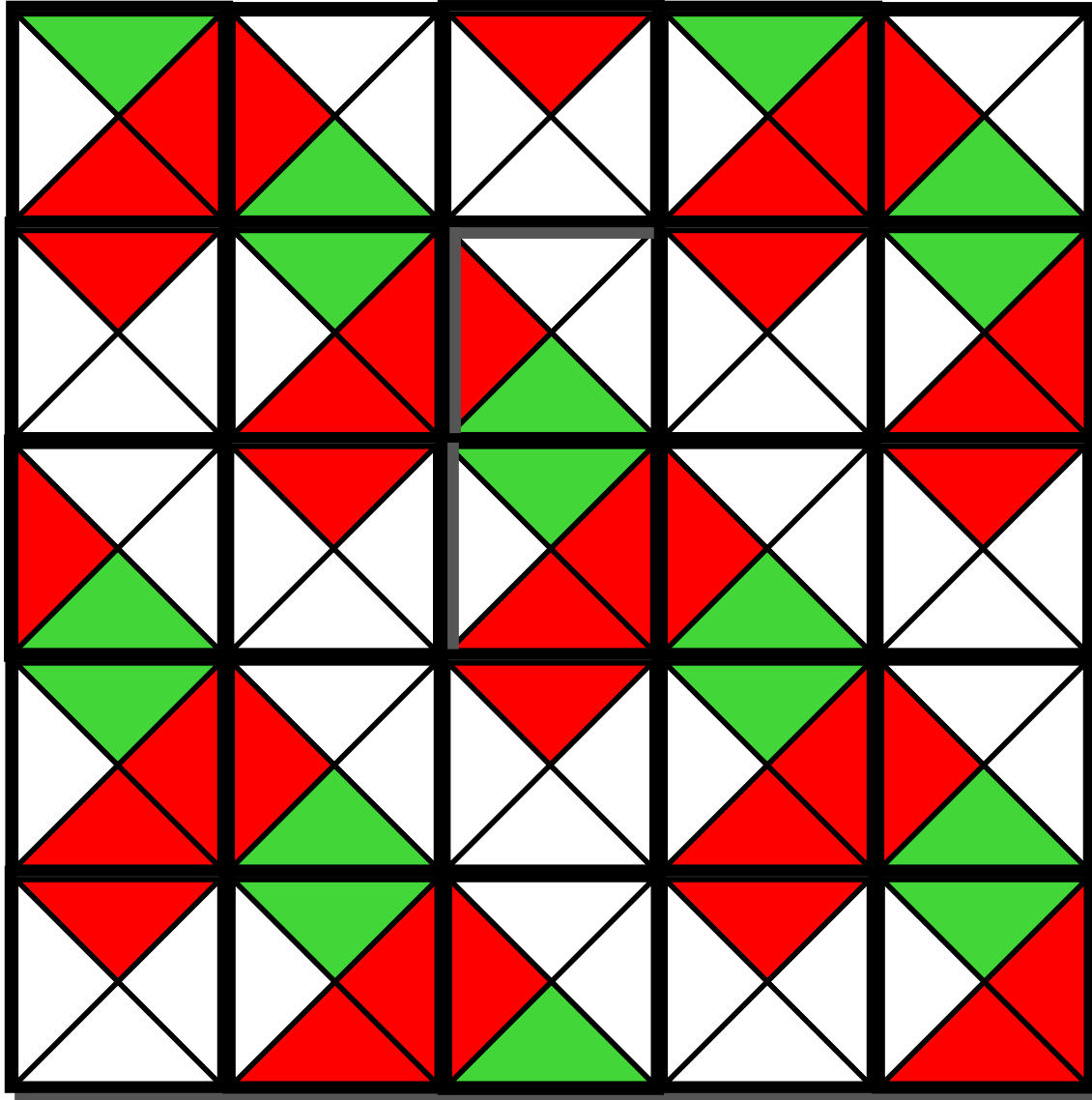
- nur gleichfarbige Seiten dürfen aneinandergelegt werden!
- Steine dürfen nicht gedreht werden!

Theorem: Das Kachelproblem **(A)** ist *unentscheidbar*.

Für **(B)** gibt es zur Zeit deterministische Verfahren nur mit exponentiellem Zeitaufwand, **denn das Problem ist \mathcal{NP} -vollständig**.



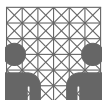
Kachelproblem (eine Lösung)



Kommt 1 als Funktionswert vor?

Definition: Mit \mathcal{M}_T sei die Menge aller (Kodierungen von) Turing-Maschinen M bezeichnet, die totale Funktionen $f_M : \Sigma^* \rightarrow \{0, 1\}$ berechnen.

Gegeben:	Eine beliebige stets haltende Turing-Maschine
Gesucht:	Wird bei allen Eingaben stets die Ausgabe 0 berechnet?
Antwort:	JA oder NEIN



Beweis

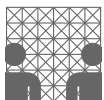
Annahme der Entscheidbarkeit von

$$\mathcal{P} := \{M \in \mathcal{M}_T \mid \exists w \in \Sigma^* : f_M(w) = 1\}$$

\Rightarrow TM $A_{\mathcal{P}}$ berechne $\chi_{\mathcal{P}}$ von \mathcal{P} .

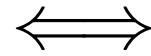
$$\text{Also: } \chi_{\mathcal{P}}(\langle M \rangle) = \begin{cases} 1, & \text{falls } M \in \mathcal{P} \\ 0, & \text{sonst} \end{cases}$$

- Betrachte Klasse $M_{B,w}$ von TM, mit:
 - Eingabe $n \in \mathbb{N} \rightarrow$ simuliere genau n Schritte von B bei Eingabe von w ,
 - halte an und gib 1 aus, sofern B nach genau n Schritten auf w hält,
 - sonst gib eine 0 aus.

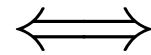


Beweis (Forts.)

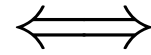
$$\chi_{\mathcal{P}}(\langle M_{B,w} \rangle) = 1$$



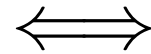
$$M_{B,w} \in \mathcal{P}$$



$M_{B,w}$ druckt bei Eingabe von n eine 1



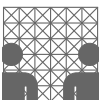
B hält auf w nach n Schritten an



$$w \in L(B)$$

Dann entscheidet $A_{\mathcal{P}}$ aber das Halteproblem!

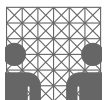
Widerspruch !!!



Unentscheidbarkeitsresultate

Theorem: Es gibt keinen Algorithmus, der für eine beliebige Funktion $f_M \in \mathcal{M}_T$ entscheidet, ob $f_M(w) = 1$ für mindestens ein $w \in \Sigma^*$ gilt.

Korollar: Es ist unentscheidbar, ob eine beliebige, durch eine TM definierte, entscheidbare Menge M leer ist.

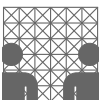


Alternativen zum durch Turing-Maschinen definierten Berechenbarkeitsbegriff



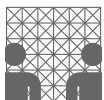
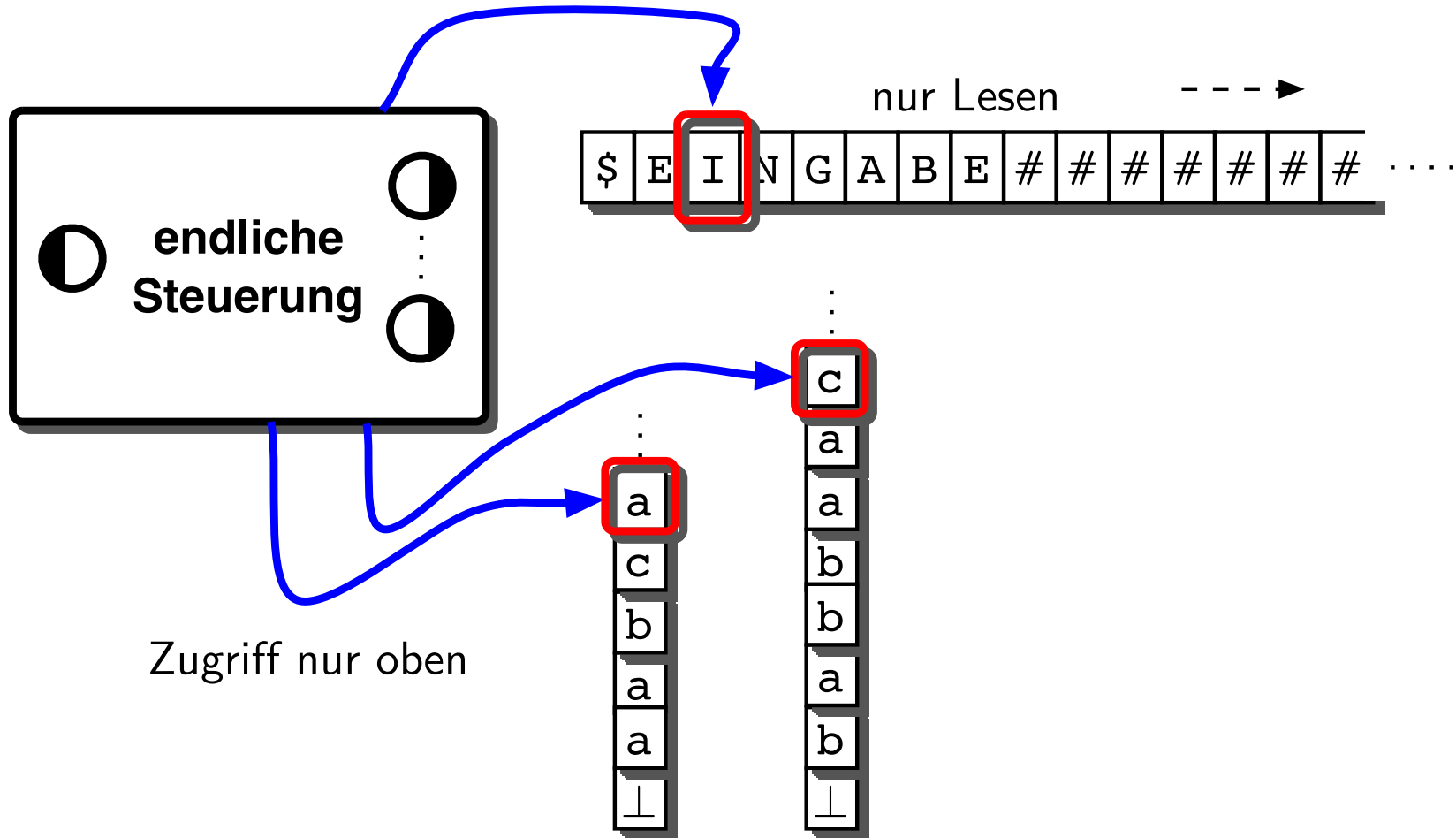
Übersicht

- Automatenmodelle:
 - 2-Kellerautomaten/ k -Kellerautomaten
 - Zählerautomaten
- mathematische Modelle:
 - μ -rekursive Definierbarkeit
- realistische Modelle:
 - RAM (Random Access Machine)

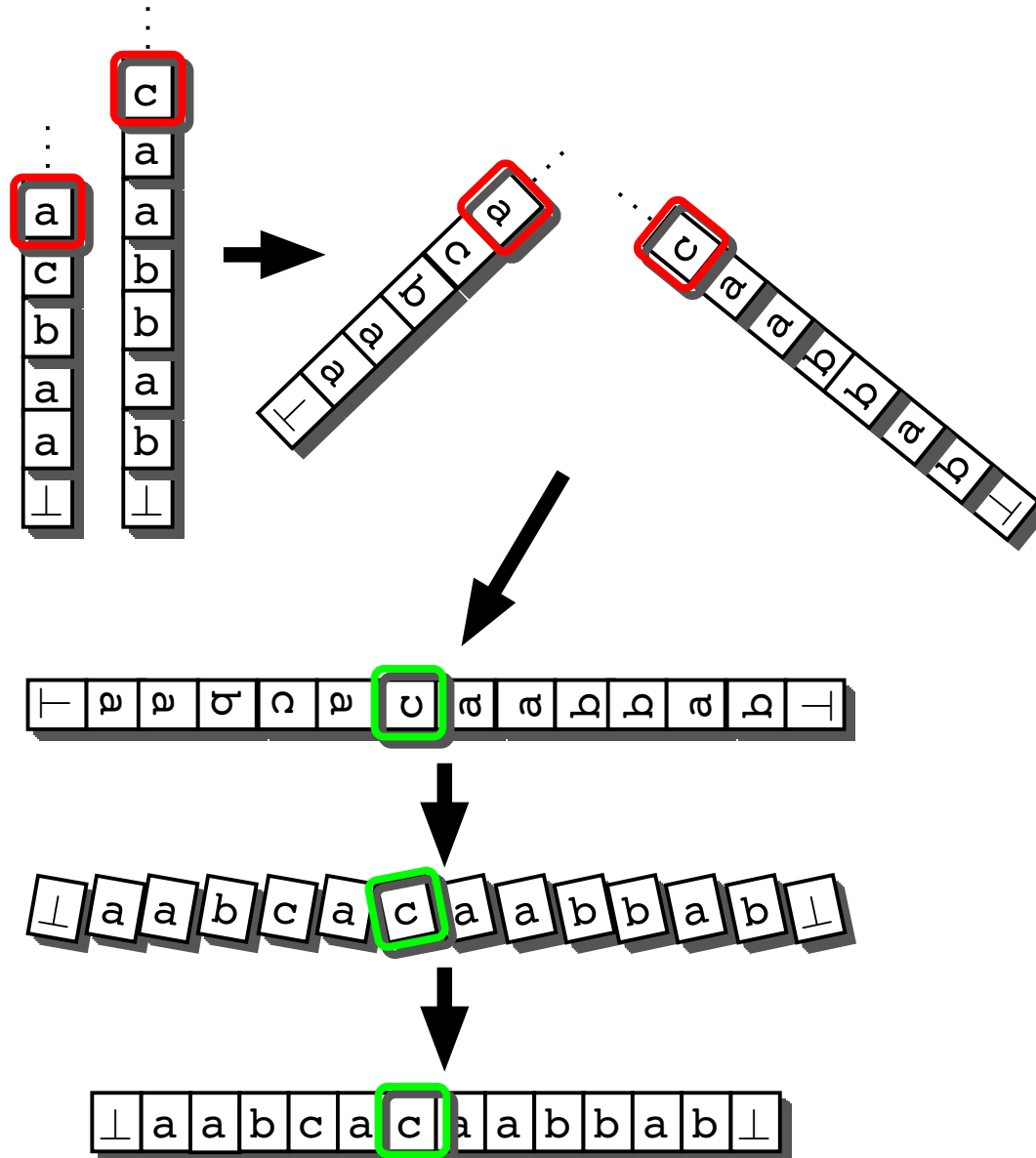


2-Kellerautomaten

... zur Akzeptierung von Sprachen:



2 Keller = Turing-Band



Zählerautomaten

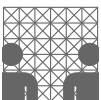
Ein k -**Zähler-Automat** ist ein k -Keller-Automat, bei dem das Kellularphabet für die Keller (zusätzlich zu dem benutzten **Kellerboden- symbol** \perp) nur aus einem einzigen Zeichen, z.B. $*$, besteht.

Für $\Gamma = \{y_1, y_2, \dots, y_{k-1}\}$ wird Kellerinhalt $y_{i_1}y_{i_2}y_{i_3} \dots y_{i_m}$ durch die Zahl $i_1 \cdot k^{m-1} + i_2 \cdot k^{m-2} + \dots + i_{m-1} \cdot k + i_m$ in eindeutiger Weise k -när kodiert.

push(y_i) entspricht der Änderung $z := z \cdot k + i$

pop entspricht der Änderung $z := \lfloor \frac{z}{k} \rfloor$

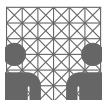
top= y_i entspricht dem Test $i = z \bmod k$



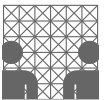
Äquivalenzen

Eine Menge M ist genau dann aufzählbar, wenn sie von einem Automaten der folgenden Art erkannt werden kann:

1. Einer deterministischen Turing-Maschine (DTM).
2. Einer nichtdeterministischen Turing-Maschine (NTM).
3. Einem 2-Keller-Automaten.
4. Einem 2-Zähler-Automaten.



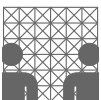
Mathematische Definition der Berechenbarkeit: primitive- und μ -Rekursion



Prinzip der primitiven Rekursion

Funktionen werden zusammengesetzt

- aus elementaren **Basisfunktionen**
 - konstante Funktionen
 - Nachfolgerfunktion
 - Projektionen (Zugriff auf eine Komponente eines Tupels)
- durch Anwendung von elementaren **Operationen**
 - Substitution (Schachtelung)
 - primitive Rekursion



Basisfunktionen

■ **Nachfolgerfunktion:** $S : \mathbb{N} \rightarrow \mathbb{N} = \{0, 1, 2, \dots\}$ mit
 $S(n) := n + 1$.

■ **konstante Funktionen:** $C_q^r : \mathbb{N}^r \rightarrow \mathbb{N}$ für $r, q \in \mathbb{N}$ mit:

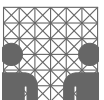
$$C_q^r(n_1, n_2, \dots, n_r) := q$$

für alle r -Tupel $(n_1, n_2, \dots, n_r) \in \mathbb{N}^r$. Die nullstellige
Konstante $q \in \mathbb{N}$ ist somit: $C_q^0 := q$.

■ **Projektionsfunktionen:** $U_i^r : \mathbb{N}^r \rightarrow \mathbb{N}$ mit:

$$U_i^r(n_1, n_2, \dots, n_r) := n_i$$

für alle r -Tupel $(n_1, n_2, \dots, n_r) \in \mathbb{N}^r$.



Operationen (1)

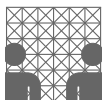
Substitution: Sind $f : \mathbb{N}^r \rightarrow \mathbb{N}$ und $g_1, g_2, \dots, g_r : \mathbb{N}^m \rightarrow \mathbb{N}$ in \mathcal{PR} , so ist auch die Funktion $h : \mathbb{N}^m \rightarrow \mathbb{N}$ mit

$$h(n_1, n_2, \dots, n_m) := f(g_1(n_1, \dots, n_m), \dots, g_r(n_1, \dots, n_m))$$

in \mathcal{PR} .

... zum Beispiel: $h : \mathbb{N}^2 \rightarrow \mathbb{N}$ mit $h(x, y) := (x^2 \cdot y) + 1$ wird zusammengesetzt aus $S : \mathbb{N} \rightarrow \mathbb{N}$ und $mult : \mathbb{N}^2 \rightarrow \mathbb{N}$ mit $mult(x, y) := x \cdot y$ als:

$$h(a, b) = S(mult(mult(U_1^2, U_1^2), U_2^2))(a, b) = S(mult(mult(a, a), b))$$



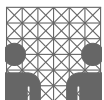
Operationen (2)

primitive Rekursion: Sind $f : \mathbb{N}^r \rightarrow \mathbb{N}$ und $g : \mathbb{N}^{r+2} \rightarrow \mathbb{N}$ in \mathcal{PR} , so ist auch die folgende Funktion $h : \mathbb{N}^{r+1} \rightarrow \mathbb{N}$ in \mathcal{PR} , die den folgenden *Rekursionsgleichungen* genügt:

$$a) \quad h(0, n_1, n_2, \dots, n_r) := f(n_1, \dots, n_r),$$

$$b) \quad h(S(n), n_1, \dots, n_r) := g(n, h(n, n_1, \dots, n_r), n_1, \dots, n_r),$$

h entsteht durch primitive Rekursion aus f und g .

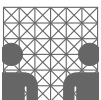


Definition \mathcal{PR}

Die Menge \mathcal{PR} der **primitiv rekursiven Funktionen** besteht aus den *Basisfunktionen* und den sich in endlich vielen Schritten durch *Substitution* und *primitive Rekursion* ergebenden Funktionen.

Andere Funktionen gehören nicht zu \mathcal{PR} .

Jede primitiv-rekursive Funktion $f \in \mathcal{PR}$ ist eine **totale** Funktion, d.h. sie ist überall definiert. Außerdem ist $f(x)$ stets eindeutig bestimmt.



Beispiel: Addition in \mathbb{N}

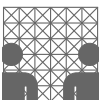
Die Addition in \mathbb{N} ist durch die folgenden Rekursionsgleichungen definiert:

$add : \mathbb{N}^2 \rightarrow \mathbb{N}$ mit

$$\begin{aligned} add(0, n) &:= U_1^1(n), \\ add(S(m), n) &:= g(m, add(m, n), n), \\ g(x, y, z) &:= S(U_2^3(x, y, z)). \end{aligned}$$

Übliche Kurzform:

$$\begin{aligned} 0 + n &:= n, \\ (m + 1) + n &:= (m + n) + 1. \end{aligned}$$



Beispiel: Multiplikation in \mathbb{N}

Die Multiplikation in \mathbb{N} , $mult : \mathbb{N}^2 \rightarrow \mathbb{N}$, ist definiert durch:

$$mult(0, n) := C_0^1(n),$$

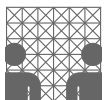
$$mult(S(m), n) := g(m, mult(m, n), n),$$

$$g(x, y, z) := add(U_2^3(x, y, z), U_3^3(x, y, z)).$$

Kurzform:

$$0 \cdot n := 0,$$

$$(m + 1) \cdot n := m \cdot n + n.$$



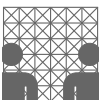
Beispiel: Fakultät

$$h(0) := C_1^0$$

$$h(S(n)) := \text{mult}(S(U_1^2(n, h(n))), U_2^2(n, h(n)))$$

Da *mult* primitiv rekursiv ist, ist somit die Fakultätsfunktion $_! : \mathbb{N} \rightarrow \mathbb{N}$ als primitiv rekursiv nachgewiesen durch:

$$h(n) = n!$$



Eigenschaften

Theorem: Jede primitiv-rekursive Funktion $f : \mathbb{N}^r \rightarrow \mathbb{N}$ ist Turing-berechenbar.

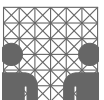
Die Umkehrung des Satzes ist jedoch falsch!

Denn es gilt:

Theorem: Nicht jede Turing-berechenbare Funktion ist total.

Frage: *Sind die totalen Turing-berechenbaren Funktionen genau die primitiv-rekursiven Funktionen?*

NEIN! Gegenbeispiel: Ackermann-Funktion.



Definition: Ackermann-Funktion

Die **Ackermann-Funktion** ist wie folgt definiert:

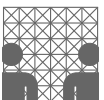
■ $f(0, n) := n + 1,$

■ $f(m + 1, 0) := f(m, 1),$

■ $f(m + 1, n + 1) := f(m, f(m + 1, n)).$

Die Ackermann-Funktion ist *Turing-berechenbar*, ihre Funktionswerte sind für *alle* Argumente eindeutig bestimmt.

Theorem: Die Ackermann-Funktion ist nicht primitiv-rekursiv.



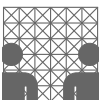
Lemma zu prim. rek. Funktionen

Sei $g : \mathbb{N}^r \rightarrow \mathbb{N}$ primitiv-rekursiv. Dann gibt es ein $c \in \mathbb{N}$, so daß

$$g(n_1, n_2, \dots, n_r) < f(c, n_1 + n_2 + \dots + n_r)$$

für alle $(n_1, n_2, \dots, n_r) \in \mathbb{N}^r$.

Zu jeder primitiv rekursiven Funktion gibt es eine Konstante, so daß die Ackermann-Funktion als obere Schranke dienen kann.



Beweis: Ackermann-Funktion

Beweisidee: Angenommen, die f sei primitiv-rekursiv. Dann ist auch die Funktion

$$g(n) := f(n, n)$$

primitiv-rekursiv, denn

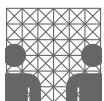
$$g(n) = f(U_1^2(n, m), U_1^2(n, m)) \in \mathcal{PR}.$$

Nach Lemma gibt es ein $c \in \mathbb{N}$ mit

$$g(n) < f(c, n) \text{ für alle } n \in \mathbb{N}.$$

Für den Spezialfall $n = c$ gilt dann aber:

$$g(c) < f(c, c) = g(c).$$



Widerspruch!!!

mächtiger durch μ -Rekursion

Sei $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ eine beliebige partielle Funktion und (x_1, x_2, \dots, x_n) ein beliebiges, festes n -Tupel aus \mathbb{N}^n . Dann betrachten wir die Folge von existierenden (!) Funktionswerten:

$$y_0 := f(x_1, x_2, \dots, x_n, 0),$$

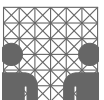
$$y_1 := f(x_1, x_2, \dots, x_n, 1),$$

$$\vdots$$

$$y_k := f(x_1, x_2, \dots, x_n, k),$$

$$\vdots$$

Minimiere k , so daß $y_k = 0$.



Definition: μ -Operator

Sei $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ eine partielle Funktion.

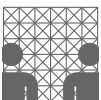
Sei ferner $M := \{i \mid f(x_1, x_2, \dots, x_n, i) = 0\}$.

Dann ist

$$\mu y [f(x_1, x_2, \dots, x_n, y) = 0]$$

$$:= \begin{cases} \min(M) & \text{falls } \min(M) \text{ existiert} \\ & \text{und } \forall j < \min(M). (x_1, x_2, \dots, x_n, j) \in \text{Def}(f) \\ \text{div} & \text{sonst} \end{cases}$$

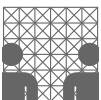
... allgemein: $\mu y [Q(\bar{x}, y)]$ für eine Eigenschaft Q .



Def.: μ -rekursive Funktionen

Eine Funktion f heißt **μ -rekursiv** (**partiell-rekursiv**) genau dann, wenn f entweder

1. eine primitiv-rekursive Grundfunktion ist oder
2. durch Substitution aus μ -rekursiven Funktionen entsteht oder
3. durch primitive Rekursion aus μ -rekursiven Funktionen entsteht oder
4. durch Anwendung der Minimalisierung (des μ -Operators) aus μ -rekursiven Funktionen entsteht.



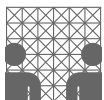
Die partiell rekursiven Funktionen

Theorem: Die Ackermann-Funktion ist μ -rekursiv.

Theorem: Für eine n -stellige Funktion $f : \mathbb{N}^n \rightarrow \mathbb{N}$ sind folgende Aussagen äquivalent:

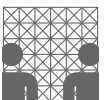
1. f ist durch eine 1-DTM berechenbar,
2. f ist durch eine k -DTM berechenbar,
3. f ist μ -rekursiv (partiell-rekursiv),
4. f ist durch eine RAM berechenbar.

⋮



Ausblick

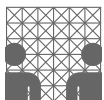
- Ein realistisches Maschinenmodell: die RAM;
- erweiterte Grammatiken:
 - kontextsensitive Grammatiken,
 - Phrasenstrukturgrammatiken;



Äquivalenzen

Eine Menge M ist genau dann aufzählbar, wenn sie von einem Automaten der folgenden Art erkannt werden kann:

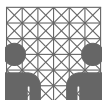
1. Einer deterministischen Turing-Maschine (DTM).
2. Einer nichtdeterministischen Turing-Maschine (NTM).
3. Einem 2-Keller-Automaten.
4. Einem 2-Zähler-Automaten.
5. M ist durch eine RAM akzeptierbar.



Äquivalenz DTM/NTM

1. Jede DTM ist auch NTM.
2. Simulation einer NTM auf einer DTM.
 - DTM hat keine Platzbeschränkung;
 - Merken des Konfigurationsbaumes auf dem Band;
 - systematische Suche nach Erfolgsrechnung.

Details im Skript und in der Literatur!

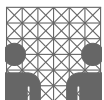


Die partiell rekursiven Funktionen

Theorem: Für eine n -stellige Funktion $f : \mathbb{N}^n \rightarrow \mathbb{N}$ sind folgende Aussagen äquivalent:

1. f ist durch eine 1-DTM berechenbar,
2. f ist durch eine k -DTM berechenbar,
3. f ist μ -rekursiv (partiell-rekursiv),
4. f ist durch eine RAM berechenbar.

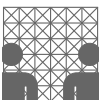
⋮



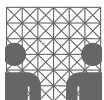
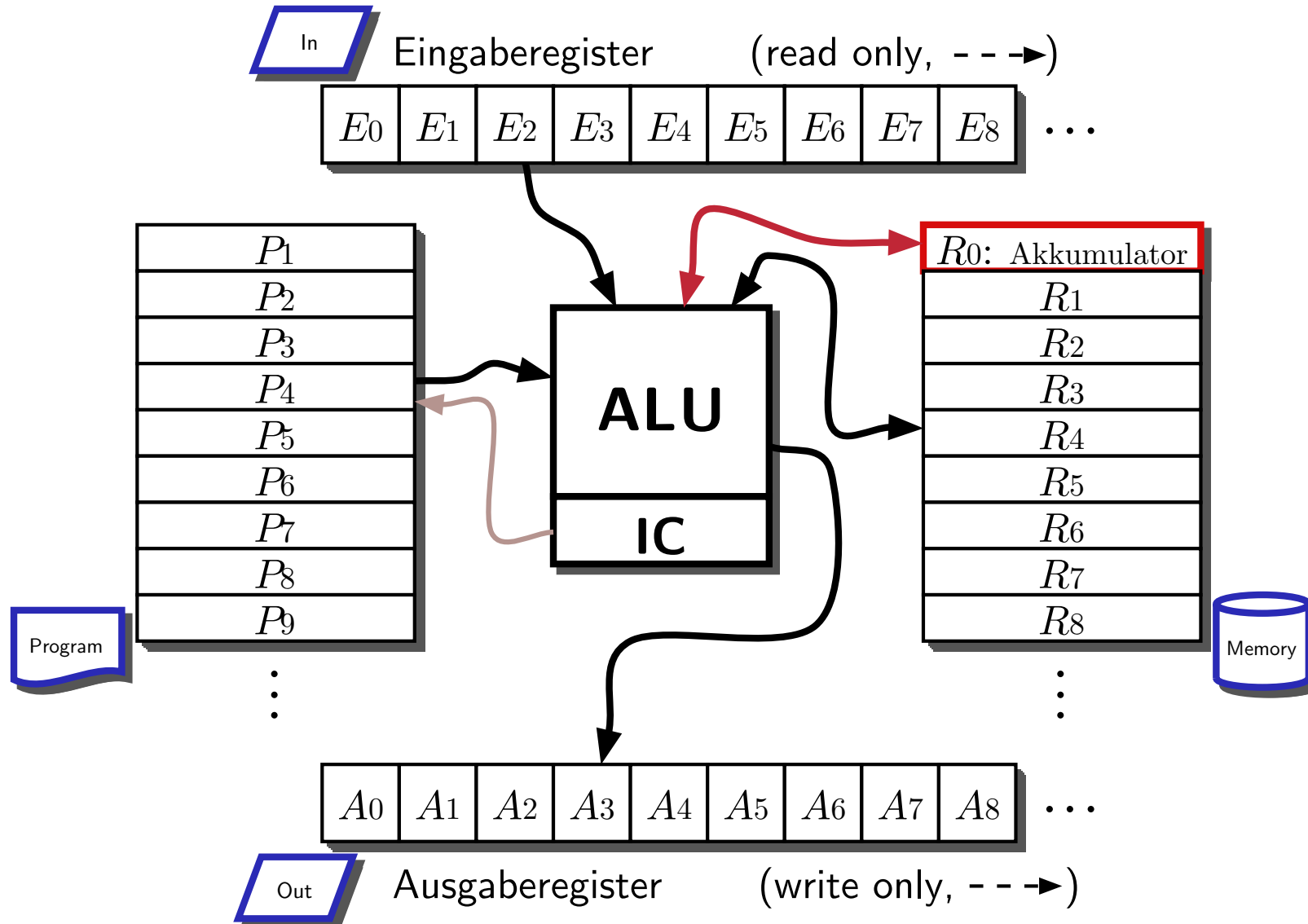
Random Access Machine (RAM)

Was unterscheidet die RAM von der TM?

- Rechnet auf natürlichen Zahlen.
- Speichermodell: Register
 - nehmen beliebige natürliche Zahlen auf
 - direkter Zugriff auf Register über Adressen möglich
 - indirekte Adressierung möglich
- Programmmodell: festes Programm



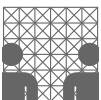
RAM-Schaubild



RAM-Varianten

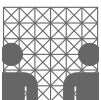
zwei verschiedene Varianten der RAM werden unterscheiden:

- Die *Bit-RAM*, bei der die Register beliebige natürliche (ganze) Zahlen enthalten dürfen
- Die *arithmetische RAM*, bei der die Speicherinhalte aus einem beliebigen Körper, wie z.B. \mathbb{Z} oder \mathbb{Q} , stammen können.
- Weitere Modellvarianten entstehen bei Einschränkung des erlaubten Befehlssatzes.



RAM-Befehle

- Operationen: READ, WRITE, LOAD, STORE, ADD, SUB, JUMP, JZERO, JGTZ
- Operanden:
 - $= i$ [die Zahl i]
 - i [Inhalt des Registers R_i]
 - $*i$ [indirekte Adresse: Inhalt des Registers R_j , wenn j Inhalt des Registers R_i ist]
- Besonderheiten: zusätzliche Operationen MULT und DIV bei RAM_{*}



RAM-Befehle (2)

1. LOAD a $c(0) \leftarrow v(a)$
2. STORE i $c(i) \leftarrow c(0)$
 STORE $*i$ $c(c(i)) \leftarrow c(0)$
3. ADD a $c(0) \leftarrow c(0) + v(a)$
4. SUB a $c(0) \leftarrow c(0) - v(a)$
5. MULT a $c(0) \leftarrow c(0) \times v(a)$
6. DIV a $c(0) \leftarrow \lfloor c(0) \div v(a) \rfloor$
7. READ i $c(i) \leftarrow$ aktuelles Eingabesymbol
 READ $*i$ $c(c(i)) \leftarrow$ aktuelles Eingabesymbol.
8. WRITE a $v(a)$ wird an der aktuellen Position auf die Ausgabe geschrieben



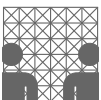
RAM-Befehle (3)

9. JUMP b Der *location counter* (Befehlsregister) wird auf die mit b markierte Anweisung gesetzt.
10. JGTZ b Der *location counter* wird auf die mit b markierte Anweisung gesetzt, wenn $c(0) > 0$ ist, sonst auf die nachfolgende Anweisung.
11. JZERO b Der *location counter* wird auf die mit b markierte Anweisung gesetzt, wenn $c(0) = 0$ ist, sonst auf die nachfolgende Anweisung.
12. HALT Programmausführung beenden.



Besonderheiten

- Felder auf dem Eingabe- und Ausgabeband enthalten Zahlen!
- Der Eingabekopf bewegt sich bei jedem READ um ein Feld nach rechts.
- Nach jedem WRITE bewegt sich der Kopf auf der Ausgabe um ein Feld nach rechts.
- Eine RAM ist immer **deterministisch!**

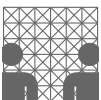


RAM-Berechenbarkeit

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}^l$ heißt *RAM-berechenbar* gdw. eine RAM existiert, die für die Eingabe (x_1, \dots, x_k) immer die Ausgabe $(y_1, \dots, y_l) = f(x_1, \dots, x_k)$ liefert, sofern $(x_1, \dots, x_k) \in \text{Def}(f)$.

Zum Beispiel ist die Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ mit $f(n) := n^2$ RAM-berechenbar (siehe Skript).

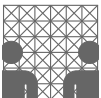
RAM-Programme für Bit-RAM's berechnen genau die partiell rekursiven Funktionen!



RAM-Sprachakzeptierung

Sei $\Sigma = \{a_1, \dots, a_m\}$ ein Alphabet. Das Symbol a_i wird durch die natürliche Zahl i kodiert (Schreibweise: $a_i^{(k)} = i$).

- Das Eingabewort $w = w_1 \dots w_k$ wird in den Feldern des Eingabebandes gespeichert.
 - w_i als die Zahl $w_i^{(k)}$ im i -ten Feld.
- im $(k + 1)$ -ten Feld wird 0 als Endmarkierung gespeichert.
- Ein RAM-Programm P akzeptiert w , gdw. P am Ende der Berechnung 1 in das erste Feld der Ausgabe geschrieben wurde.



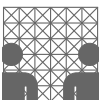
RAM-Sprachakzeptierung (2)

Die Sprache, welche von Programm P akzeptiert wird, ist die Menge der Wörter, die P akzeptiert.

RAM-Programme für Bit-RAM's akzeptieren genau die rekursiv aufzählbaren Sprachen.

Problem: Wie kann Komplexität definiert werden?

- Bei TM: Einfach durch Zählen der Schritte und Zählen der max. in einer Erfolgsrechnung genutzten Bandzellen.
- Bei der RAM: Zwei unterschiedliche Maße!



Komplexitätsmaße

- uniformes Maß
- logarithmisches Maß

Uniformes Zeitmaß:

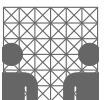
1 Schritt = 1 Zeiteinheit

Uniformes Platzmaß:

1 Register = 1 Platzeinheit

Logarithmisches Maß ist geeigneter, da es die Größe der Zahlen in Registern und Adressen berücksichtigt. Dazu definieren wir:

$$l(i) = \begin{cases} \lfloor \log i \rfloor + 1 & i \neq 0 \\ 1 & i = 0 \end{cases}$$

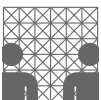


Logarithmisches Kostenmaß

- Kosten hängen von den verwendeten Operanden ab:

Operand a	Kosten $t(a)$
$= i$	$l(i)$
i	$l(i) + l(c(i))$
$*i$	$l(i) + l(c(i)) + l(c(c(i)))$

- Diese Kosten werden für die Berechnung der Komplexität des *Akzeptierens* bzw. der *Funktionsberechnung* auf die einzelnen Befehle angewandt.



konkrete Kosten

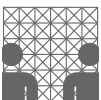
1. LOAD a $t(a)$
2. STORE i $l(c(0)) + l(i)$
STORE $*i$ $l(c(0)) + l(i) + l(c(i))$
3. ADD a $l(c(0)) + t(a)$
4. SUB a $l(c(0)) + t(a)$
5. MULT a $l(c(0)) + t(a)$
6. DIV a $l(c(0)) + t(a)$
7. READ i $l(\text{input}) + l(i)$
READ $*i$ $l(\text{input}) + l(i) + l(c(i))$
8. WRITE a $t(a)$



konkrete Kosten (2)

9.	JUMP b	1
10.	JGTZ b	$l(c(0))$
11.	JZERO b	$l(c(0))$
12.	HALT	1

-
- Das *logarithmische Platzmaß* eines Registers ist die maximale Länge $l(i)$ aller Zahlen i , die im Register gespeichert wurden.
 - Beim logarithmischen Kostenmaß für den Platz muss noch eine Korrekturgröße addiert werden!

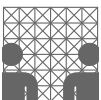


Simulation einer TM durch RAM

Wir skizzieren, wie TM und RAM sich gegenseitig simulieren können.

Eine $T(n)$ -zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$ -zeitbeschränkt ist und die im logarithmischen Maß $O(T(n) \cdot \log T(n))$ -zeitbeschränkt ist.

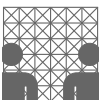
- pro Bandzelle ein Register
- log-Faktor wg. Speicherung der Kopfposition!



Simulation einer RAM durch TM

Eine im logarithmischen Maß $T(n)$ -zeitbeschränkte RAM_+ kann durch eine $O(T^2(n))$ -zeitbeschränkte 5-Band DTM M simuliert werden.

- Register hintereinander auf dem Band
- quadratischer Faktor wg. Speicherung der Register auf dem Band! $\Rightarrow S(n) \cdot T(n)$
abschätzen durch $T(n) \cdot T(n)$!

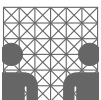


Komplexitätsklasse \mathcal{P}

\mathcal{P} ist die Klasse der Sprachen (algorithmischen Probleme), die man in Polynomialzeit erkennen (lösen) kann.

$\mathcal{P}Space$ ist die Klasse der Sprachen (algorithmischen Probleme), die man mit polynomiell viel Platz erkennen (lösen) kann.

- Obige Klassen sind unabhängig vom verwendeten Modell (RAM oder TM).

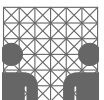


Literatur-Tip

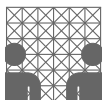


... eine unterhaltsame Exposition vieler interessanter Zusammenhänge zum Thema *Grenzen allen Wissens* (insbesondere aus der theoretischen Informatik und der Logik).

David Harel
Das Affenpuzzle
Springer-Verlag, 2001



Phrasenstrukturgrammatiken und kontextsensitive Grammatiken

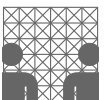


Sprachfamilie

Eine Menge \mathcal{L} heißt **Sprachfamilie** genau dann, wenn folgendes gilt:

1. Es existiert eine Sprache $L \in \mathcal{L}$ mit $L \neq \emptyset$, d.h. folglich gilt auch $\mathcal{L} \neq \emptyset$.
2. Für jede Sprache $L \in \mathcal{L}$ gibt es ein endliches Alphabet Σ_L mit $L \subseteq \Sigma_L^*$.

Bisher bekannt: $Cf \cap Reg = Reg$, d.h. jede reguläre Sprache ist auch kontextfrei.



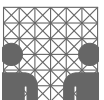
semi-Thue System (STS)

Ein **semi-Thue System** (STS) über dem Alphabet Σ ist eine (endliche oder unendliche) Teilmenge $S \subseteq \Sigma^* \times \Sigma^*$ ist ein Tupel (Σ, S) .

- Produktionen $(u, v) \in S$ werden oft als $u \longrightarrow v$ geschrieben.
- Die **Einschrittige Ableitungsrelation**

$\xRightarrow{(S)} \subseteq \Sigma^* \times \Sigma^*$, ist definiert durch: $w_1 \xRightarrow{(S)} w_2$,

wenn $w_1 = \alpha u \beta$, $w_2 = \alpha v \beta$ für $\alpha, \beta \in \Sigma^*$ und $u \longrightarrow v \in S$.



Ableitungsrelation

- Die reflexive, transitive Hülle von $\xRightarrow{(S)}$ wird mit

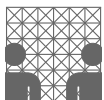
$\xRightarrow{*}_{(S)}$ bezeichnet, und ist die von S definierte

Ableitungsrelation.

- Weiterehin für $n \in \mathbb{N}$:

- $\xRightarrow{n+1}_{(S)} := \xRightarrow{n}_{(S)} \circ \xRightarrow{(S)}$

- $\xRightarrow{0}_{(S)} := Id_S$



Phrasenstrukturgrammatik

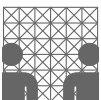
Eine **Typ-0** oder **Phrasenstruktur-Grammatik** wird durch ein Tupel $G := (V_N, V_T, P, S)$ spezifiziert. Hierbei gilt:

V_N ist ein endliches Alphabet von **Nonterminalen**,

V_T ist endliches Alphabet von **Terminalsymbolen** mit $V_N \cap V_T = \emptyset$,

$S \in V_N$ ist das **Startsymbol**.

$P \subseteq V^*V_NV^* \times V^*$ ist endliche Menge von **Produktionen** (oder **Regeln**), also ein spezielles STS über V .



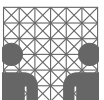
Generator: Typ-0-Grammatik

- Eine Produktion $(u, v) \in P$ wird auch hier meist als $u \longrightarrow v$ geschrieben.

- Die von G generierte oder erzeugte Sprache ist

$$L(G) := \{w \in V_T^* \mid S \xrightarrow[(P)]{*} w\}.$$

- Mit \mathcal{L}_0 wird die Familie aller von Typ-0 Grammatiken erzeugbaren Sprachen bezeichnet. D.h., $\mathcal{L}_0 := \{L \mid L = L(G) \text{ für eine Typ-0 Grammatik } G\}$



Typ-0-Sprachen

Beispiel:

Sei G gegeben durch die Produktionen:

$$S \longrightarrow abc \mid aRbc$$

$$Rb \longrightarrow bR$$

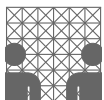
$$bL \longrightarrow Lb$$

$$Rc \longrightarrow Lbcc$$

$$aL \longrightarrow aaR$$

$$aL \longrightarrow aa$$

Dann gilt $L(G) = \{a^n b^n c^n \mid n \geq 1\}$.

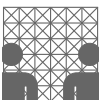


kontextsensitive Grammatik

Ein Spezialfall der Typ-0 Grammatik $G = (V_N, V_T, P, S)$ ist die **Typ-1** oder **kontextsensitive Grammatik**. Hierfür ist $u \longrightarrow v \in P$, wenn *entweder*

- $u = \alpha A \beta$, $v = \alpha w \beta$ mit $A \in V_N$, $w \in V^+$ und $\alpha, \beta \in V^*$ *oder*
- $u = S$, $v = \lambda$ und S kommt in keiner Produktion auf der rechten Seite vor.

Die Familie der **kontextsensitiven Sprachen** wird mit \mathcal{C}_s oder \mathcal{L}_1 abgekürzt, und es ist $\mathcal{L}_1 := \{L \mid L = L(G) \text{ für eine Typ-1 Grammatik } G\}$.



Vergleich der Regeln

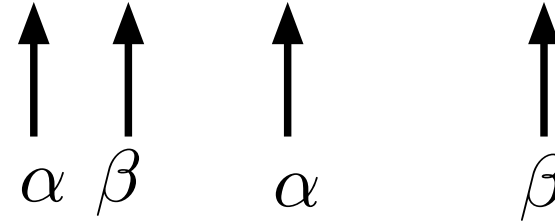
①
kontextfrei

$$A \rightarrow aBccA$$

②
kontextsensitiv

ⓐ $cA \rightarrow caBccD$

ⓑ $cAb \rightarrow caBccDb$

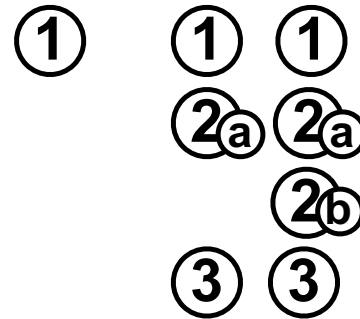


③
nicht
kontextsensitiv

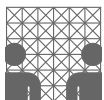
$$cA \rightarrow aBccD$$

Beispiel:

AbcAcAba

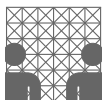


} Anwendbarkeit
der Regeln



Theorem Jede von einer Turing-Maschine akzeptierte Sprache kann auch von einer Typ-0 Grammatik generiert werden und umgekehrt, kurz:
 $\mathcal{L}_0 = \mathcal{RE}$.

- Was sind monotone Grammatiken?
- Gibt es eine Automatenmodell für kontextsensitive Sprachen?
- Einige Entscheidbarkeitsresultate.



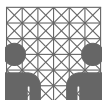
$$\mathcal{L}_0 = \mathcal{RE}$$

Theorem Jede von einer Turing-Maschine akzeptierte Sprache kann auch von einer Typ-0 Grammatik generiert werden und umgekehrt, kurz: $\mathcal{L}_0 = \mathcal{RE}$.

Der Beweis erfolgt in zwei Schritten:

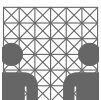
1. $\mathcal{L}_0 \subseteq \mathcal{RE}$
2. $\mathcal{RE} \subseteq \mathcal{L}_0$

Beweisidee: Simulation der Ableitungen einer Grammatik durch eine TM, bzw. Simulation der Rechnung einer TM durch eine Grammatik.



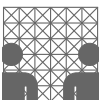
$$\mathcal{L}_0 \subseteq \mathcal{RE}$$

- konstruiere zu Typ-0 Grammatik G eine NTM:
 - kopiere das Eingabewort w auf eine Spur des Arbeitsbandes,
 - schreibe auf zweite Spur das Startsymbol S von G ;
- NTM führt die einzelnen Ableitungsschritte auf der zweiten Spur aus;
- NTM prüft nach jeder Ersetzung nach, ob das Ergebnis mit der Eingabe übereinstimmt und akzeptiert bei Gleichheit.



$$\mathcal{RE} \subseteq \mathcal{L}_0$$

- O.B.d.A. $L \in \mathcal{RE}$ ist $L(M)$ für eine DTM $M := (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$;
- Erzeuge aus Startsymbol S beliebige Anfangskonfiguration $q_0w \in Z\Sigma^*$ und kopiere w dahinter;
 - wie eine Grammatik für $\{ww \mid w \in \Sigma^*\}$;
 - Ableitbarkeit von $\{eQ_0w\mid w \in \Sigma^*\}$ aus S ;
- Gemäß δ zu $q_i \in Z$ das Nonterminal Q_i mit linkem und rechtem Nachbarsymbol ersetzen;
- Einer Endkonfiguration entsprechendes Anfangsstück löschen und in das Terminalwort w überführen.

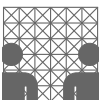


$\mathcal{RE} \subseteq \mathcal{L}_0$ (Forts.)

$$\begin{aligned}\forall y \in \Gamma : \quad & yQ_i x \longrightarrow Q_j y z, \text{ falls } \delta(q_i, x) = (q_j, z, L) \\ & eQ_i x \longrightarrow eQ_j \# z, \text{ falls } \delta(q_i, x) = (q_j, z, L) \\ & Q_i x \longrightarrow zQ_j, \text{ falls } \delta(q_i, x) = (q_j, z, R) \\ & \qquad \qquad \qquad \text{und } x \neq \emptyset \\ & Q_i \emptyset \longrightarrow zQ_j \emptyset, \text{ falls } \delta(q_i, \#) = (q_j, z, R)\end{aligned}$$

Löschen der Konfigurationsinformation:

$$\begin{aligned} & Q_i \longrightarrow F, \text{ falls } q_i \in Z_{\text{end}} \\ \forall y \in \Gamma : \quad & Fy \longrightarrow F, \\ \forall y \in \Gamma : \quad & yF \longrightarrow F, \\ & eF\emptyset \longrightarrow \lambda\end{aligned}$$



monotone Grammatik

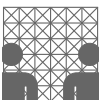
- Was sind monotone Grammatiken?

Eine Typ-0 Grammatik $G = (V_N, V_T, P, S)$ heißt **monoton**, falls $\forall u \longrightarrow v \in P : (|u| \leq |v|)$.
Einzige erlaubte Ausnahme: $S \longrightarrow \lambda$ und S kommt in keiner Produktion rechts vor.

- Kontextsensitive Grammatiken sind monoton!
- Wir können monotone Regel kontextsensitiv simulieren, mit neuen Nonterminalen. Damit Terminalsymbole nicht stören, ersetzen wir sie überall durch Nonterminale der Menge:

$$V'_T := \{A' \mid A \in V_T\}.$$

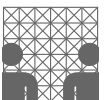
Am Schluss dann $A' \longrightarrow A$ für jedes $A \in V_T$!



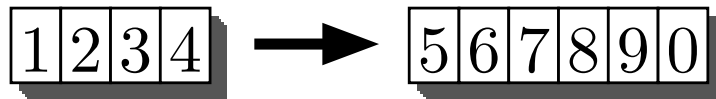
Simulation einer monotonen Regel

- Wir können monotone Regel kontextsensitiv simulieren, mit neuen Nonterminalen:

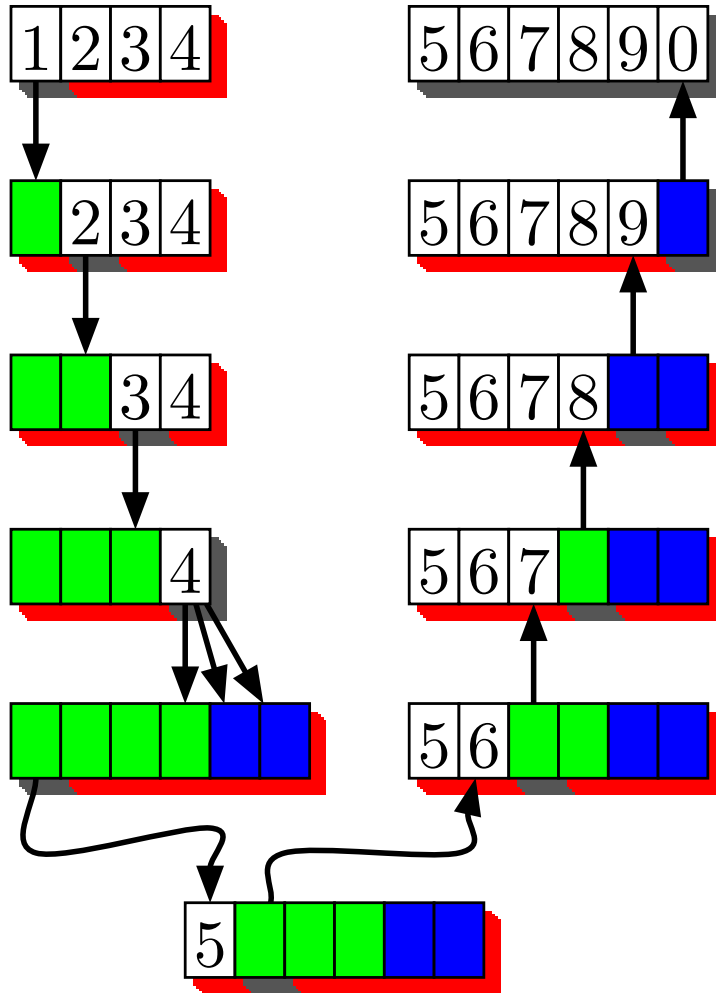
$$V'_N := \left\{ \binom{A}{k} \mid A \in V_N \cup V'_T \text{ und } 1 \leq k \leq |P| \right\}$$
$$\cup V_N \cup V'_T$$



monotone Regel

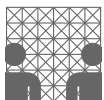


beliebige monotone Regel



grüne und blaue Felder
sind neue Nonterminale

rot unterlegt ist der
jeweilige, unverändert
bleibende Kontext.



Simulation monotoner Regel

Sei $A_1 A_2 \dots A_n \longrightarrow B_1 B_2 \dots B_m$ die k -te Regel aus P .
hier: $1 \leq i \leq n, 1 \leq j \leq m, n \leq m, A_i, B_j \in V'_N$

Simulation:

$$\begin{array}{l} A_1 A_2 \dots A_n \longrightarrow \binom{A_1}{k} A_2 \dots A_n \\ \binom{A_1}{k} A_2 \dots A_n \longrightarrow \binom{A_1}{k} \binom{A_2}{k} \dots A_n \\ \vdots \\ \binom{A_1}{k} \binom{A_2}{k} \dots \binom{A_{n-1}}{k} A_n \longrightarrow \binom{A_1}{k} \dots \binom{A_n}{k} \binom{B_{n+1}}{k} \dots \binom{B_m}{k} \\ \binom{A_1}{k} \dots \binom{A_n}{k} \binom{B_{n+1}}{k} \dots \binom{B_m}{k} \longrightarrow B_1 \binom{A_2}{k} \dots \binom{B_m}{k} \\ \vdots \\ B_1 B_2 \dots \binom{B_m}{k} \longrightarrow B_1 B_2 \dots B_m \end{array}$$

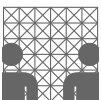


Simulation monotoner Regel (2)

... zusätzlich werden benötigt: $A' \longrightarrow A$
für alle $A \in V_T$.

- Die einmal begonnene Simulation einer Regel kann nur durch Abarbeiten der simulierenden Regeln beendet werden!
- Zu jeder monotonen Regel existiert ein Satz simulierender Regeln.

Damit haben wir gezeigt, dass $\mathcal{L}_1 = \mathcal{MON}$



Übung

Seien $G_i := (\{S_i, A, B, C\}, \{a, b, c\}, P_i, S_i)$
Grammatiken mit:

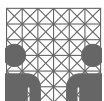
$P_1 :$

$$\begin{array}{l} S_1 \longrightarrow ABS_1 \quad | \quad a \\ aB \longrightarrow aC \\ A \longrightarrow a \\ C \longrightarrow c \end{array}$$

$P_2 :$

$$\begin{array}{l} S_2 \longrightarrow AS_2B \quad | \quad \lambda \\ A \longrightarrow a \\ B \longrightarrow b \end{array}$$

- Sind G_1 und G_2 kontextsensitiv?
- Was sind $L(G_1)$ und $L(G_2)$?



Entscheidbarkeitsresultate

... zur Wiederholung:

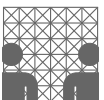
- Wann ist eine Menge entscheidbar?



Wortproblem

Jede kontextsensitive Sprache ist entscheidbar, d.h.
 $\mathcal{L}_1 \subseteq \mathcal{REC}$.

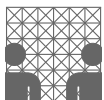
- Für jedes Wort w kann entschieden werden, ob $w \in L \in \mathcal{L}_1$.
 - Es gibt endlich viele Wörter v mit $|v| \leq |w|$.
 - Es gibt eine kontextsensitive Grammatik $G = (V_N, V_T, P, S)$ mit $L(G) = L$.
 - Keine Regel verkürzt die Satzform!
 - Ohne Satzformwiederholungen gibt es nur endlich viele Ableitungsschritte, bis die Satzform zu lang ist!!



$$\mathcal{L}_1 \neq \mathcal{REC} \subsetneq \mathcal{RE}$$

Diagonalbeweis: (ähnlich L_d)

- Die Menge der kontextsensitiven Grammatiken ist aufzählbar: G_1, G_2, \dots
- Sei $f : \{0, 1\}^* \rightarrow \mathbb{N}$ eine berechenbare Bijektion, mit: $f(w) = i$ gdw. w ist i -tes Wort in der lexikalischen Ordnung auf $\{0, 1\}^*$.
- $L_e := \{w \mid w \notin L(G_{f(w)})\}$ ist entscheidbar!
- ABER: L_e ist nicht kontextsensitiv!



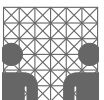
$\mathcal{L}_1 \neq \mathcal{REC}$ (Fortsetzung)

Annahme: $\exists n$ mit $L(G_n) = L_e$. Für das n -te Wort u der lexikalischen Aufzählung von $\{0, 1\}^*$ mit $f(u) = n$ ergibt sich ein Widerspruch für beide möglichen Fälle $u \in L_e$ und $u \notin L_e$:

$$u \in L_e \xrightarrow{\text{Def. } L_e} u \notin L(G_n) \xrightarrow{\text{Annahme}} u \notin L_e$$

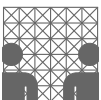
Andererseits ergibt sich auch:

$$u \notin L_e \xrightarrow{\text{Def. } L_e} u \in L(G_n) \xrightarrow{\text{Annahme}} u \in L_e$$



linear beschränkte Automaten

- Gibt es eine Automatenmodell für kontextsensitive Sprachen?
 - Ein **linear beschränkter Automat (LBA)** ist eine NTM, die bei beliebiger Eingabe w auf dem Arbeitsband höchstens $c \cdot |w|$ Felder bis zur Akzeptierung besucht.
 - $c \in \mathbb{R}^+$ ist eine Konstante, die nicht von w abhängt.
 - Arbeitet die TM bei gleicher Beschränkung ihres Arbeitsbandes deterministisch, so wird der linear beschränkte Automat mit **DLBA** abgekürzt.

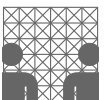


Äquivalenz: $\mathcal{LBA} = \mathcal{L}_1$

Die Familie der von LBA's bzw. DLBA's akzeptierten Sprachen wird mit \mathcal{LBA} bzw. \mathcal{DLBA} bezeichnet.

Theorem: Es gilt: $\mathcal{LBA} = \mathcal{MON} = \mathcal{L}_1$

Also: Die *kontextsensitiven Sprachen* sind genau die *durch LBA's akzeptierbaren Sprachen*!

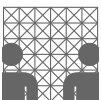


Beweis: $MON \subseteq LBA$

Diese Inklusion läßt sich auf ähnliche Weise zeigen, wie im Falle von Typ-0 Grammatiken und TM:

- Simuliere Ableitung in *monotoner* Grammatik auf einer NTM.
- Brich ab, sobald die abgeleitete Satzform auf Spur 2 länger ist als das Eingabewort.
- Akzeptiere, falls Satzform auf Spur 2 exakt mit dem Eingabewort übereinstimmt.

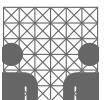
Diese Maschine benötigt nur $c \cdot |w|$ Platz.



Beweis: $LBA \subseteq MON$

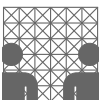
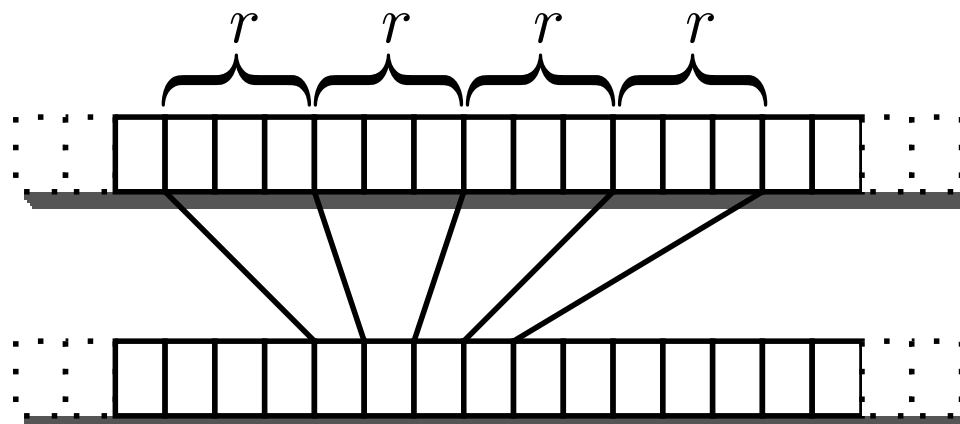
Gegeben ist ein LBA, d.h. eine TM, die mit $c \cdot |w|$ Platz auskommt.

- Die Konstruktion für TM und Typ-0-Grammatik liefert *keine* monotone Grammatik!
- Deshalb muss eine andere Simulation gewählt werden.
- Zunächst „normieren“ wir den LBA auf Platzbedarf $\max. |w|$.



Bandkompression

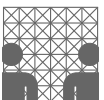
- Für $0 \leq c \leq 1$ ist nichts zu tun.
- Ansonsten. Vergrößerung der Bandalphabets und Blockbildung.
 - Blocklänge $r \in \mathbb{N}$;
 - Bandbedarf nur noch $\frac{c}{r} \cdot |w|$;
 - Für $r := \min(n \in \mathbb{N} \mid n \geq c)$ sind das höchstens $|w|$ Felder.



Symbole für die Simulation

Erklärung der verwendeten Nonterminale:

1. Spur	$\left[\begin{array}{c} x \\ y \\ \$ \\ q \end{array} \right]$	Eingabesymbol
2. Spur		Bandsymbol
3. Spur		Anfang-/Ende-Marker
4. Spur		Zustand/Kopfposition

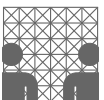


Simulation des LBA

Die *Anfangs-Konfiguration* q_0w für ein beliebiges Wort $w \in \Sigma^*$ wird aus S mit folgenden Regeln erzeugt:

$$S \longrightarrow A \begin{bmatrix} x \\ x \\ \epsilon \\ \# \end{bmatrix}, \quad A \longrightarrow A \begin{bmatrix} x \\ x \\ \# \\ \# \end{bmatrix} \quad \text{und} \quad A \longrightarrow \begin{bmatrix} x \\ x \\ e \\ q_0 \end{bmatrix},$$

für alle $x \in \Sigma$.

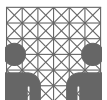


Simulation des LBA-Befehle

Linksschritt: Für $x_1, x_2, x_3 \in \Sigma$, $y_1, y_2 \in \Gamma$ und für $(p, y, z, L, q) \in K$, folgende Regeln in G :

$$\begin{array}{c} \left[\begin{array}{c} x_1 \\ y_1 \\ \# \\ \# \end{array} \right] \left[\begin{array}{c} x_2 \\ y \\ \# \\ p \end{array} \right] \end{array} \longrightarrow \begin{array}{c} \left[\begin{array}{c} x_1 \\ y_1 \\ \# \\ q \end{array} \right] \left[\begin{array}{c} x_2 \\ z \\ \# \\ \# \end{array} \right] , \begin{array}{c} \left[\begin{array}{c} x_1 \\ y_1 \\ e \\ \# \end{array} \right] \left[\begin{array}{c} x_2 \\ y \\ \# \\ p \end{array} \right] \end{array} \longrightarrow \begin{array}{c} \left[\begin{array}{c} x_1 \\ y_1 \\ e \\ q \end{array} \right] \left[\begin{array}{c} x_2 \\ z \\ \# \\ \# \end{array} \right] \end{array}$$

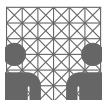
$$\begin{array}{c} \left[\begin{array}{c} x_1 \\ y_1 \\ e \\ \# \end{array} \right] \left[\begin{array}{c} x_2 \\ y \\ \# \\ p \end{array} \right] \end{array} \longrightarrow \begin{array}{c} \left[\begin{array}{c} x_1 \\ y_1 \\ e \\ q \end{array} \right] \left[\begin{array}{c} x_2 \\ z \\ \# \\ \# \end{array} \right] \end{array}$$



Simulation des LBA-Befehle (2)

Rechtsschritt: Für $x_1, x_2, x_3 \in \Sigma$, $y_1, y_2 \in \Gamma$ und für $(p, y, z, R, q) \in K$, folgende Regeln in G :

$$\begin{array}{cc}
 \begin{bmatrix} x_2 \\ y \\ \# \\ p \end{bmatrix} & \begin{bmatrix} x_3 \\ y_2 \\ \# \\ \# \end{bmatrix} & \longrightarrow & \begin{bmatrix} x_2 \\ z \\ \# \\ \# \end{bmatrix} & \begin{bmatrix} x_3 \\ y_2 \\ \# \\ q \end{bmatrix}, & \begin{bmatrix} x_2 \\ y \\ \# \\ p \end{bmatrix} & \begin{bmatrix} x_3 \\ y_2 \\ \emptyset \\ \# \end{bmatrix} & \longrightarrow & \begin{bmatrix} x_2 \\ z \\ \# \\ \# \end{bmatrix} & \begin{bmatrix} x_3 \\ y_2 \\ \emptyset \\ q \end{bmatrix} \\
 \\
 \begin{bmatrix} x_2 \\ y \\ e \\ p \end{bmatrix} & \begin{bmatrix} x_3 \\ y_2 \\ \emptyset \\ \# \end{bmatrix} & \longrightarrow & \begin{bmatrix} x_2 \\ z \\ e \\ \# \end{bmatrix} & \begin{bmatrix} x_3 \\ y_2 \\ \emptyset \\ q \end{bmatrix}
 \end{array}$$



Ende der Simulation

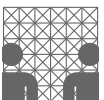
Bei Erreichen eines Endzustands des LBA müssen alle Nonterminale in entsprechende Terminalsymbol überführt werden.

Für alle $x, z \in \Sigma$, $y \in \Gamma$ und $\& \in \{\#, e, \phi\}$ folgende Regeln:

$$\begin{bmatrix} x \\ y \\ \& \\ q \end{bmatrix} \longrightarrow x \text{ falls } q \in Z_{end},$$

$$\begin{bmatrix} x \\ y \\ \& \\ \# \end{bmatrix} z \longrightarrow xz,$$

$$z \begin{bmatrix} x \\ y \\ \& \\ \# \end{bmatrix} \longrightarrow zx$$



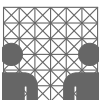
Problem/Fehler dieser Simulation

... leider steckt ein Fehler in diesem Satz von Regeln!

- Es können nur Wörter w mit $|w| \geq 2$ generiert werden!
- Warum? ϕ und e sind keine echten Begrenzer!
- Abhilfe: alle Wörter bis zur Länge 2 daraufhin untersuchen, ob sie vom LBA akzeptiert werden!

Zusätzliche Regeln $S \longrightarrow x$ für diejenigen $x \in \{\lambda\} \cup V_T$, die dazugehören müssen! Warum ist $x \in L(LBA)$ entscheidbar?

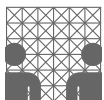
- Es gibt nur endlich viele Rechnungen **ohne Schleifen** für ein Wort!



Vorteile der LBA-Darstellung

Mit den Produktionen kann ein terminales Wort erzeugt werden, gdw. es eine Erfolgsrechnung für w im LBA mit $|w|$ Speicherplatz gibt.

- Mit Hilfe der Charakterisierungen von Sprachfamilien durch Automaten, lassen sich häufig Abschlusseigenschaften leichter beweisen als mit Grammatiken.
- Die Familie $\mathcal{L}_1 = \mathcal{LBA}$ ist gegen Durchschnittsbildung abgeschlossen.
- Beweisidee: Spurenbildung und Kombination der beiden LBA's



Kostenmaße

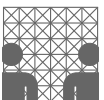
Bei der TM nur ein Kostenmaß:

- Ein *Schritt* (Konfigurationsübergang) kostet eine *Zeiteinheit*;
- eine *Bandzelle* kostet eine *Platzeinheit*.

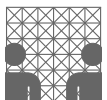
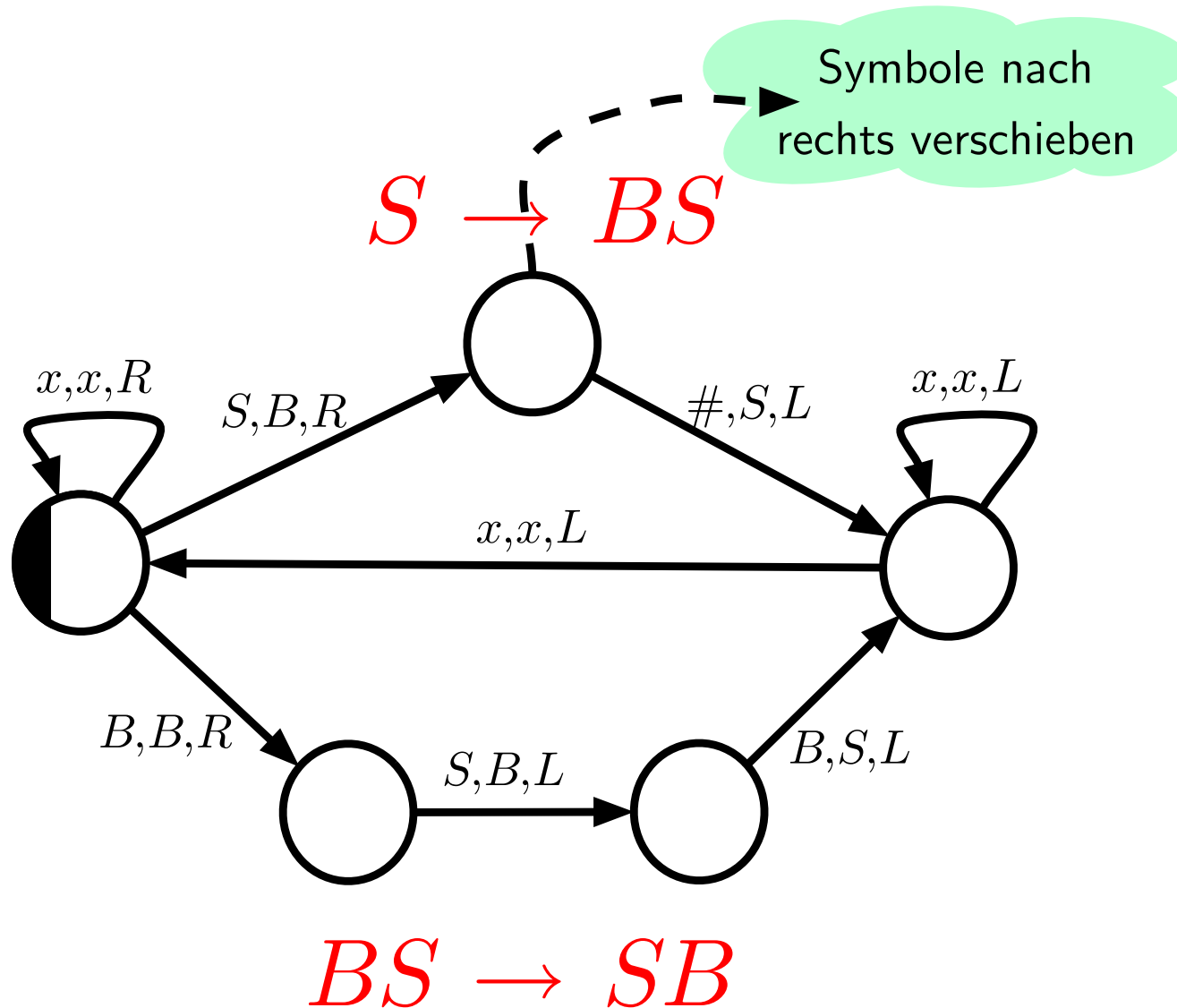
Bei der RAM zwei Kostenmaße:

- **uniformes Kostenmaß:** (wie oben);
- **logarithmisches Kostenmaß:** Die Länge der Binärdarstellung von Registeradressen und Inhalten wird berücksichtigt.

Im folgenden: „uniformes Kostenmaß“!



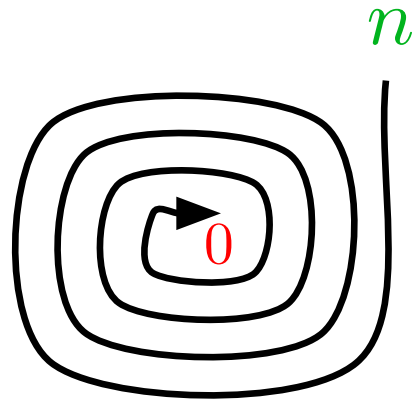
TM-Simulation von Grammatiken



Algorithmentechniken

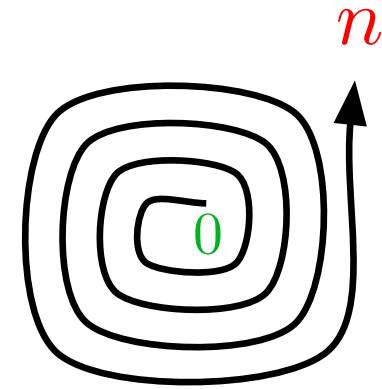


rekursiv vs. iterativ/induktiv



rekursiv

$$f(x) := g(\dots, f(x-1), \dots)$$
$$:= g(\dots, g(\dots, f(x-2), \dots), \dots)$$



iterativ

```
FOR i FROM 0 TO x DO  
  f := g(..., f, ...)
```

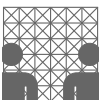


Beispiel: Fakultät rekursiv

■ $n! := \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$

■ Ein rekursives Programm:

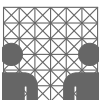
```
FUNCTION fac(n)
  BEGIN
    IF n = 0
      THEN
        RETURN 1
      ELSE
        RETURN n·fac(n-1)
    END.
```



Beispiel: Fakultät iterativ

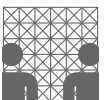
- Die übliche Definition $n! := \prod_{i=1}^n i$ suggeriert eine iterative Lösung.
- Ein iteratives Programm:

```
FUNCTION fac(n)
  BEGIN
    DECL prod INTEGER;
    prod ← 1;
    FOR i FROM 1 TO n DO
      prod ← prod·i
    RETURN prod
  END.
```



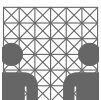
Die Qual der Wahl ...

- Welche der beiden Methoden soll nun verwendet werden?
 - Diese Frage kann immer nur mit Blick auf das Problem entschieden werden.
 - Manchmal lautet die Antwort:
„Keine von beiden!“
 - Warum? Z.B. ist bereits ab $13! = 6227020800$ das Ergebnis bei 32-Bit-Integer-Darstellung nicht mehr darstellbar!
- Was gibt es für Alternativen?



Die Suche nach Alternativen

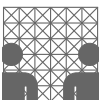
- **Im Fall der Fakultätsfunktion ist die Antwort auf die erste Frage einfach:**
 - Diese Funktion sollte gar nicht programmiert, sondern nur durch eine Tabelle implementiert werden!
- Häufig sucht man nach *geschlossenen Formen* (Berechnung ohne Schleifen).
- In jedem Fall sollte die Komplexität abgeschätzt werden, um das effizienteste Verfahren auszuwählen!



Effektivität vs. Effizienz

Effektivität: Ein Problem heißt *effektiv lösbar*, wenn ihm eine berechenbare Funktion zugrunde liegt, d.h. es ein ausführbares, endlich beschreibbares Verfahren gibt, das für alle Probleminstanzen in endlicher Zeit eine korrekte Lösung liefert. (*BF*)

Effizienz: Ein Algorithmus heißt *effizient*, wenn er ein vorgegebenes Problem in möglichst kurzer Zeit und/oder mit möglichst geringem Aufwand an Betriebsmitteln löst. (*Informatik-Duden*)



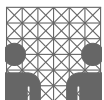
Das Problem der binären Suche

Gegeben: eine in aufsteigender Reihenfolge sortierte Liste `list` von Zahlen und ein Wert x

Gesucht: Kommt x in `list` vor?

Antwort: JA oder NEIN

- Allgemein kann auch von einer Menge mit vorgegebener linearer Ordnung $<$ ausgegangen werden.
- So formuliert man das Problem dann z.B. auch für Listen von Wörtern.



Schema: binäre Suche

1, 5, 26, 36, 38, 40, 55, 72, 77, 98

Enthält die Liste die 6?

1, 5, 26, 36, 38, 40, 55, 72, 77, 98

$6 \leq 38$ $6 \geq 40$

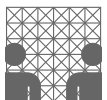
1, 5, 26, 36, 38

$6 \leq 26$ $6 \geq 36$

1, 5, 26

$6 \leq 5$ $6 \geq 26$

Nein!



Algorithmus für binäre Suche

Suche(x, A)

1. $\left[\begin{array}{l} |A| = 1 \quad A \neq \{x\} : \textit{x nicht in A} \\ A = \{x\} : \textit{x in A} \end{array} \right.$

2. Partitioniere den Suchraum in zwei Teile A_1 und A_2 , so dass $A = A_1 \uplus A_2$ mit $|A_1| = \lceil \frac{1}{2} |A| \rceil$ und $|A_2| = \lfloor \frac{1}{2} |A| \rfloor$, sowie $\max(A_1) < \min(A_2)$.

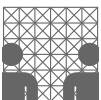
3. $\left[\begin{array}{ll} \max(A_1) < x < \min(A_2) : & \textit{x nicht in A} \\ x \leq \max(A_1) : & \text{Suche}(x, A_1) \\ x \geq \min(A_2) : & \text{Suche}(x, A_2) \end{array} \right.$



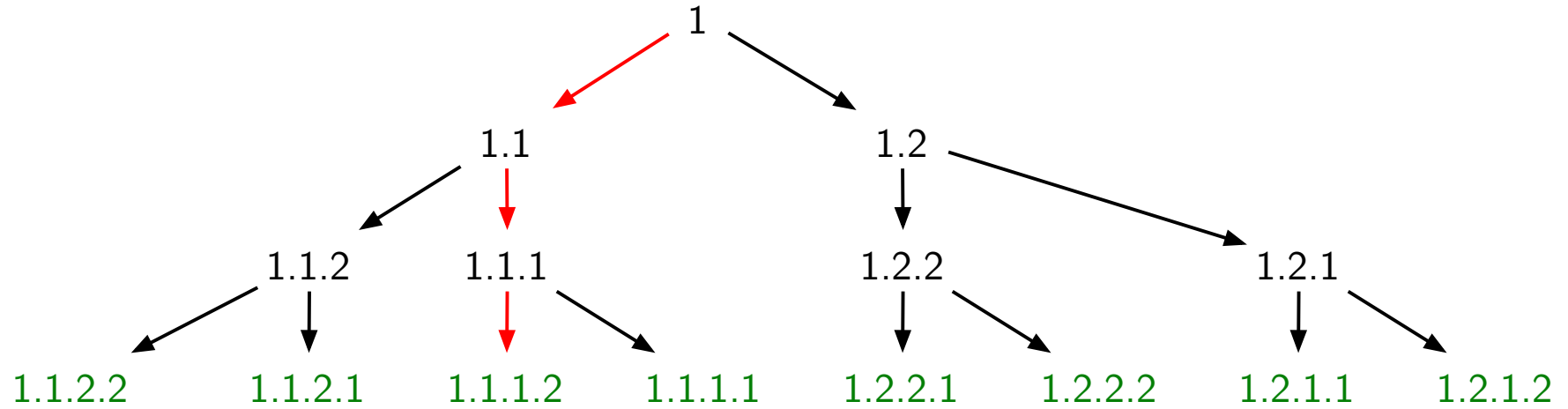
Kosten der binären Suche

- Es wird jeweils mit einem ungefähr halbiertem Suchraum fortgefahren bis das Element gefunden ist.
- Schlimmstenfalls werden $O(\log n)$ Partitionierungen durchgeführt.
 - Ein vollständiger ausgeglichener Binärbaum mit n Blättern hat eine Tiefe von $\log_2 n$.

In einem Feld mit einer Millionen Zahlen sind somit höchstens 20 Partitionierungsschritte nötig um ein bestimmtes Element zu finden!



Binärbäume

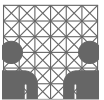


Tiefe ist $\log 8 = 3$

||

maximale Pfadlänge

8 Blätter

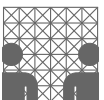


Beispiel: Sortieren

Gegeben: eine Folge von Elementen aus einer geordneten Menge: $a = (a_1, a_2, \dots, a_n)$

Gesucht: eine sortierte Folge

Antwort: eine Folge $\bar{a} = (a_{i_1}, a_{i_2}, \dots, a_{i_n})$ mit den gleichen Elementen und mit $a_{i_1} \leq a_{i_2} \leq a_{i_3} \leq \dots \leq a_{i_n}$



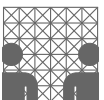
Beispiel: MERGESORT

1. Zerlegung von a in zwei (fast) gleichgroße Teilfolgen:

$$a' = (a_1, \dots, a_{\lfloor \frac{n}{2} \rfloor}),$$

$$a'' = (a_{\lfloor \frac{n}{2} \rfloor + 1}, \dots, a_n).$$

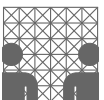
2. Sortieren von a' und a'' mit dem gleichen Verfahren zu den Ergebnissen: \bar{a}' , \bar{a}'' .
3. Mischen der sortierten Folgen \bar{a}' , \bar{a}'' zu einer sortierten Folge \bar{a} .



Komplexität von MERGESORT

... setzt sich zusammen aus der Anzahl der einzelnen Arbeitsschritte in 1.), 2.) und 3.):

1. **Zerlegung:** ein Schritt,
2. **Mischen:** $O(n)$ Schritte,
3. **Sortieren:** Sei $T_A(n)$ die Zeitkomplexität von MERGESORT. \Rightarrow Sortieren der Teilfolgen \bar{a}' und \bar{a}'' benötigt jeweils $T_A\left(\frac{n}{2}\right)$.



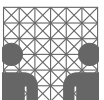
Kosten als Rekurrenzgleichung

Insgesamt ergibt sich für MERGESORT die Rekursionsgleichung:

$$T_A(n) = 2 \cdot T_A\left(\frac{n}{2}\right) + O(n),$$

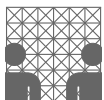
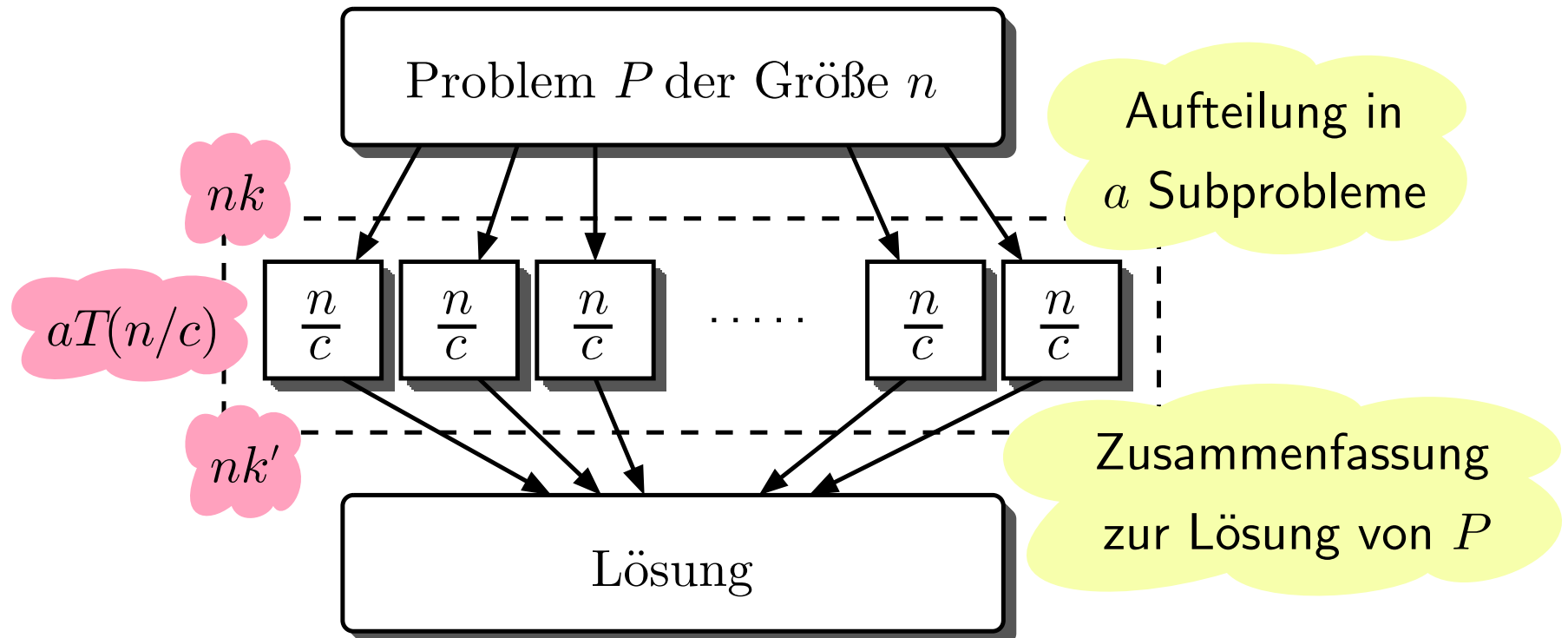
mit der Lösung:

$$T_A(n) = O(n \log n).$$



divide and conquer

MERGESORT arbeitet nach dem *divide and conquer*-Verfahren:



dichteste Punktepaare?

Gegeben: eine Menge P von n Punkten in der Ebene: $p_i = (x_i, y_i)$, $i = 1, 2, \dots, n$

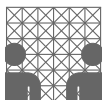
Gesucht: zwei Punkte minimaler Distanz

Antwort: Koordinaten der Punkte

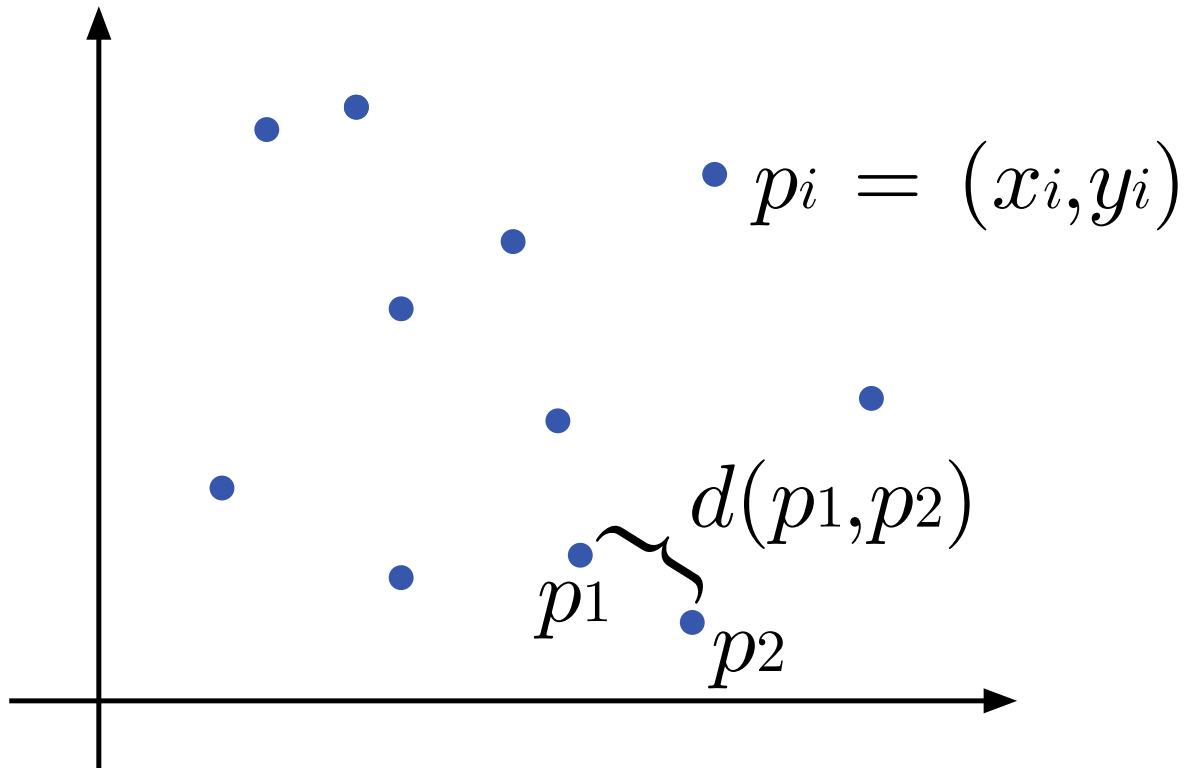
- Die **Distanz (Entfernung)** zweier Punkte ist wie folgt definiert:

$$d(p_i, p_k) := \sqrt{(x_i - x_k)^2 + (y_i - y_k)^2}$$

- Die Distanz wird in der euklidischen Metrik angegeben.



Distanz von Punkten in der Ebene



Die Punkte
liegen nicht
sortiert vor!



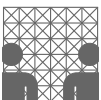
Eindimensionaler Fall

Gegeben: Eine Menge P von n Punkten auf einer Geraden:
 $P = \{p_1, p_2, \dots, p_n\} = \{x_1, x_2, \dots, x_n\}$, wobei x_i die Abszisse von p_i ist.

Gesucht: zwei Punkte minimaler Distanz

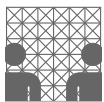
Antwort: Punktkoordinaten

Wir betrachten jetzt einen Algorithmus zur Bestimmung des dichtestes Punktepaars auf einer Linie.



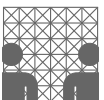
Algorithmus

1. Zerlegung von P in P_1 und P_2 mit Trennungspunkt $x = \alpha$ ($\alpha \notin P$), so dass $x_i \in P_1 \Leftrightarrow x_i < \alpha$ und $x_j \in P_2 \Leftrightarrow x_j > \alpha$.
2. Bestimmung des jeweils dichtesten Paares in P_1 und P_2 :
Rekursion: $\delta :=$ minimale Distanz der beiden Paare, die in P_1 und P_2 gefunden wurden.
3. Sei x_{max}^1 größtes Element in P_1 und x_{min}^2 kleinstes Element in P_2 : $\delta' := |x_{min}^2 - x_{max}^1|$
4. $\delta^* := \min\{\delta, \delta'\}$ ist der dichteste Abstand von Punkten in P .

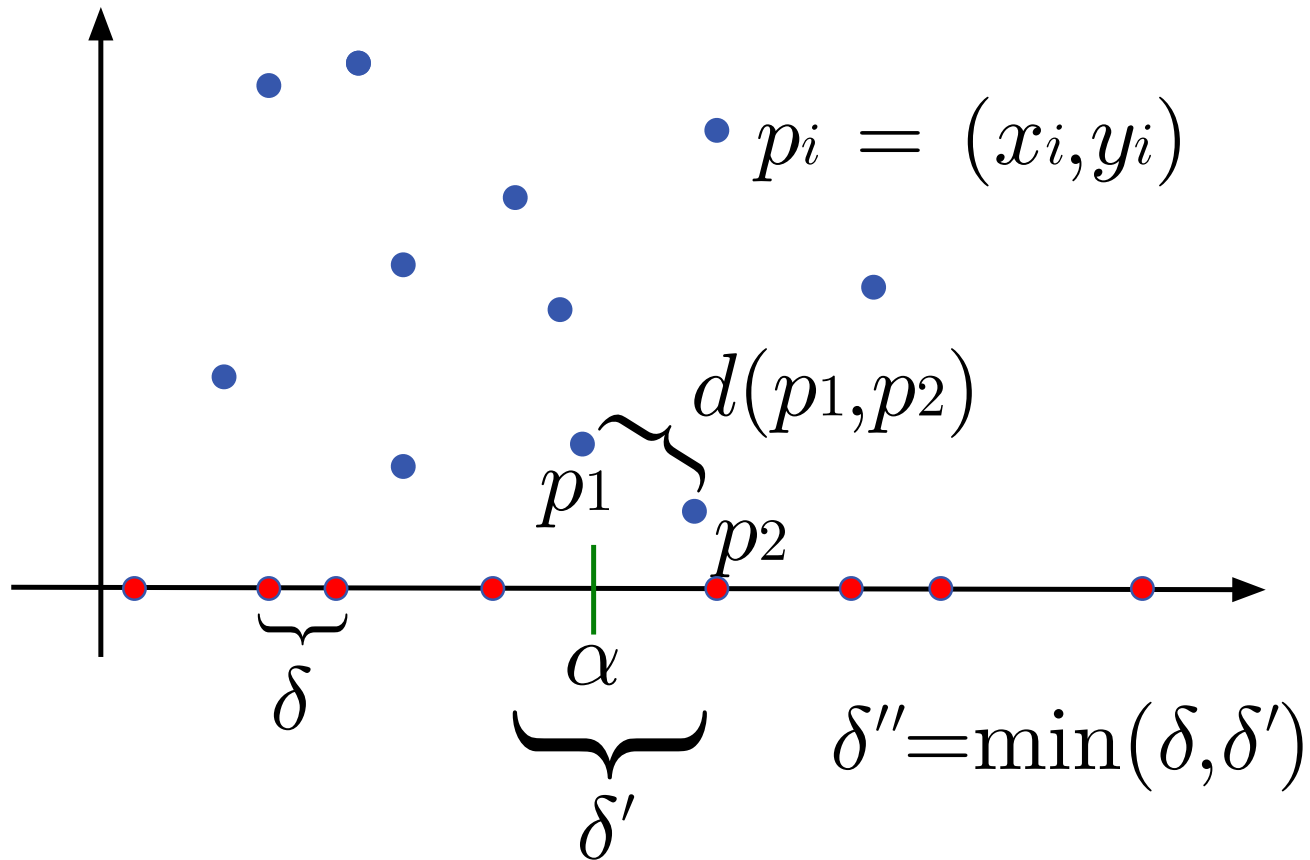


Ausblick

- Greedy-Algorithmen
- dynamische Programmierung
- Backtracking
- branch and bound
- Heuristiken und Approximation

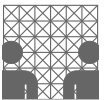


Rückblick: divide and conquer



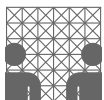
weitere Algorithmentechniken

- Greedy-Algorithmen
- dynamische Programmierung
- Backtracking
- branch and bound
- Heuristiken und Approximation



Greedy-Algorithmen

- Einsatz bei Auswahl einer Teilmenge von Objekten mit bestimmten Eigenschaften;
- Greedy-Algorithmen wählen aus der Eingabemenge sukzessive Elemente aus und entscheiden *ad hoc*, ob diese zu der Lösungsmenge gehören;
- Entscheidungen sind endgültig und können nicht wieder rückgängig gemacht werden!
- Greedy-Algorithmen suchen in der Regel nur lokal optimale Lösungen, die aber bisweilen auch global die besten sind.



Arbeitsweise des Greedy-Alg.

```
      ⋮  
SOLUTION ← ∅  
WHILE  $M \neq \emptyset$  DO  
  BEGIN  
    CHOOSE ARBITRARY  $x \in M$   
     $M \leftarrow M \setminus \{x\}$   
    IF  $x$  MATCHES CRITERION THEN  
      SOLUTION ← SOLUTION  $\cup \{x\}$   
    END  
  RETURN SOLUTION  
      ⋮
```



Beispiel: Minimales Gerüst

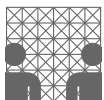
Gegeben: endlicher Graph $G = (V, E)$, ungerichtet, zusammenhängend; Kantenbewertung $l : E \rightarrow \mathbb{N}$

Gesucht: Ein Gerüst T von G mit minimaler Länge

Antwort: Graph des minimalen Gerüstes

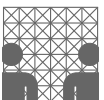
■ $l(e)$ bezeichne die "Länge" der Kante $e \in E$).

■ O.b.d.A. seien $V = \{v_1, v_2, \dots, v_n\}$,
 $E = \{e_1, e_1, \dots, e_m\}$.

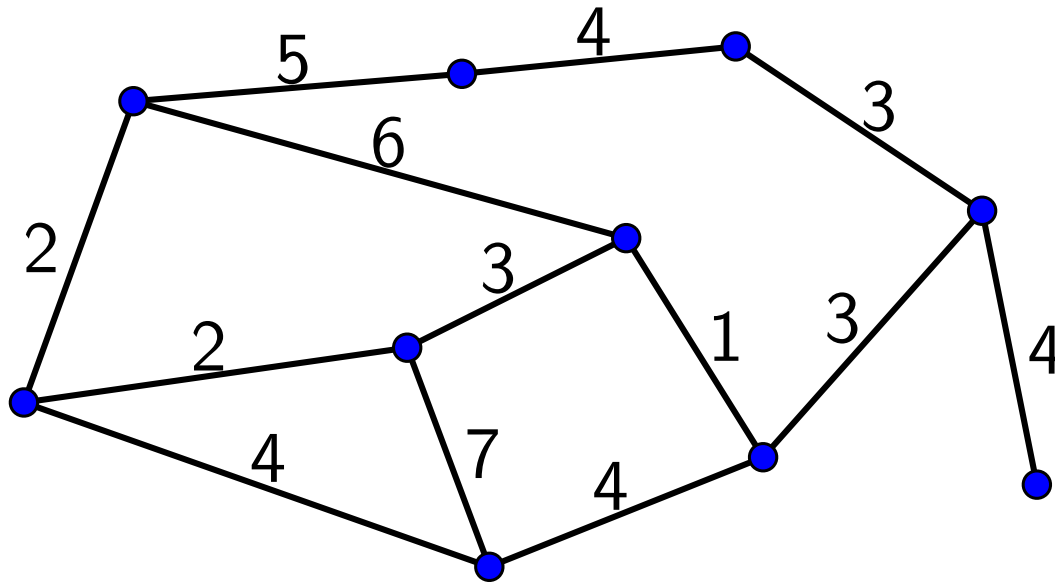


Definition: minimales Gerüst

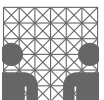
- Ein **Gerüst** von $G = (V, E)$ ist ein Teilgraph $T = (V, F)$ von G , der
 - alle Knotenpunkte von G enthält,
 - zusammenhängend ist und
 - eine minimale Anzahl von Kanten hat.
- T ist ein **Baum** und wird auch als **(auf-)spannender Baum** (*spanning tree*) bezeichnet.
- Die **Länge** von T ist: $l(T) = \sum_{e \in F} l(e)$. Dies ist die *Zielfunktion* für dieses Problem, wobei ein minimales $l(T)$ gesucht wird.
(Stichwort: Optimierung)



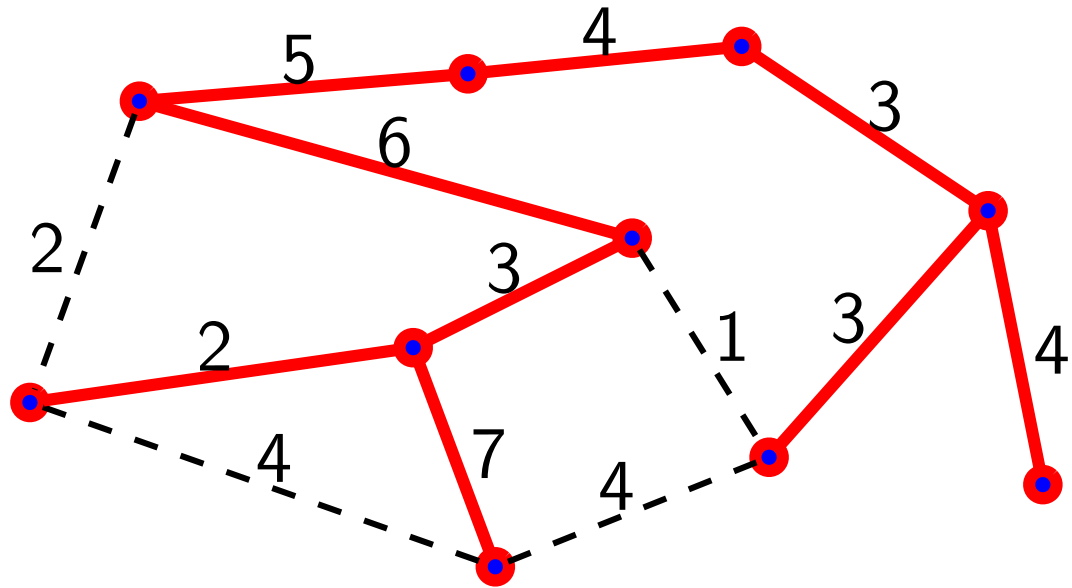
Beispielgraph G



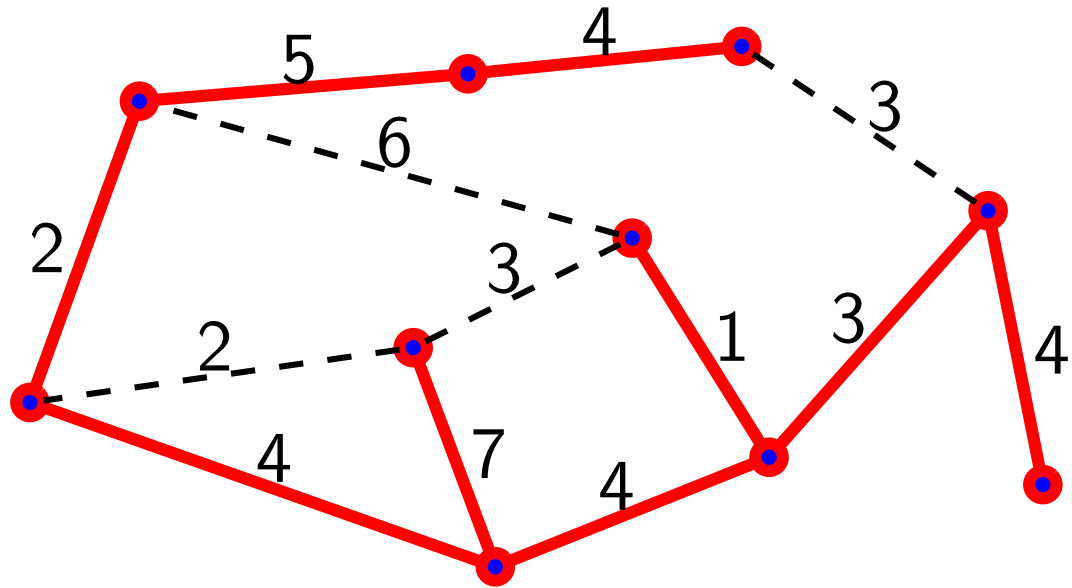
... dieser Graph ist zusammenhängend,
ungerichtet und kantenbewertet.



Ein Gerüst von G



... noch ein Gerüst von G



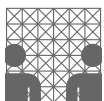
Lösungsansatz

- **Lösungskandidaten** sind alle möglichen Gerüste T von G .
- Gesucht wird dasjenige mit kleinster Länge.
- Die Lösung ist im allgemeinen nicht eindeutig!

Lösungsraum: $\mathcal{L}_G = \{T_G \mid T_G \text{ ist Gerüst von } G\}$.

Wir suchen eine Lösung für das Problem-Beispiel (G, l) :

$T_G^* \in \mathcal{L}_G$ ist *optimale* Lösung aus $\mathcal{L}_G \Leftrightarrow T_G^* \in \mathcal{L}_G$ mit $l(T_G^*) \leq l(T_G)$ für alle $T_G \in \mathcal{L}_G$.



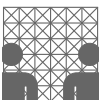
Eigenschaften

- Die *Anzahl der Lösungen* ist $|\mathcal{L}_G|$. Es gilt dabei:

$$1 \leq |\mathcal{L}_G| \leq |V|^{|V|-2} (= n^{n-2}) \quad \text{mit } |V| = n.$$

- Dabei ist $|V|^{|V|-2}$ die Anzahl der verschiedenen Gerüste des vollständigen Graphen K_n .

Sei im folgenden $G = (V, E)$ ein ungerichteter Graph mit der Kantenbewertung l .

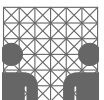


Algorithmus von PRIM/DIJKSTRA

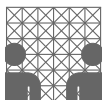
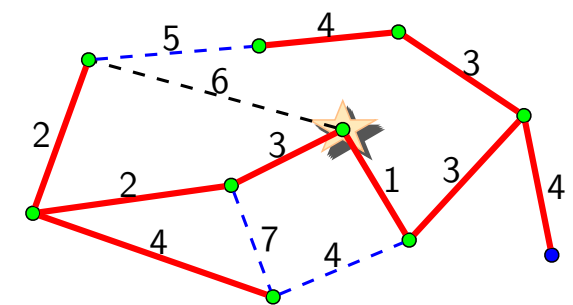
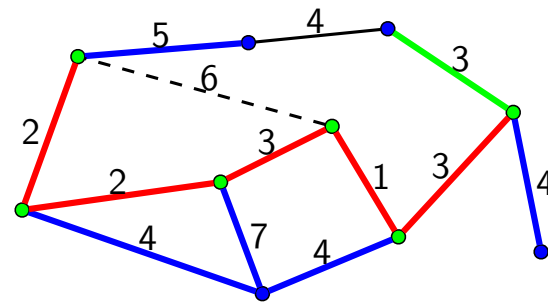
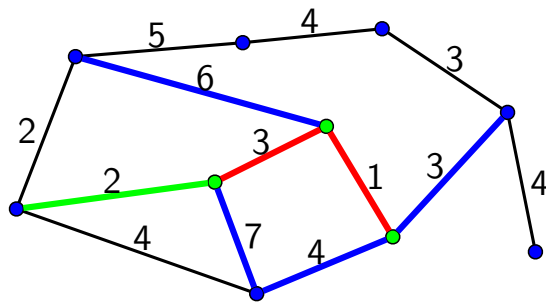
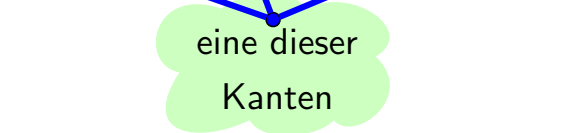
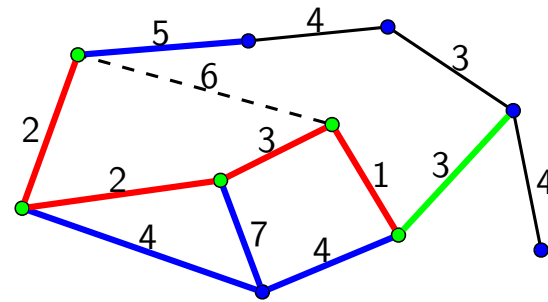
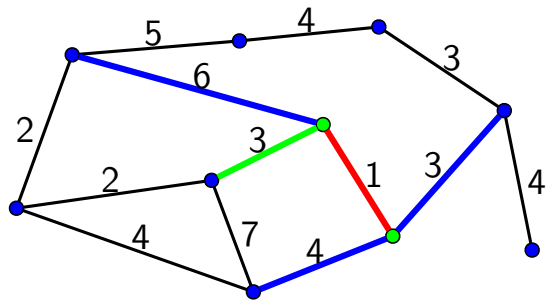
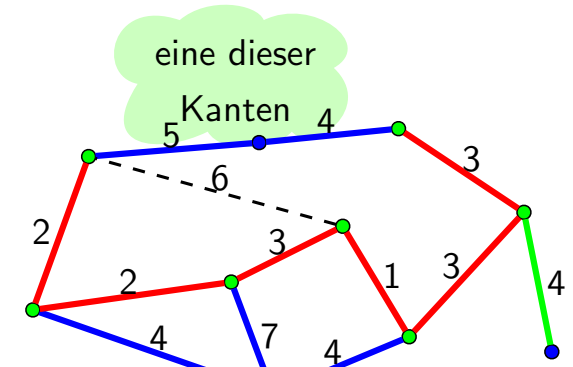
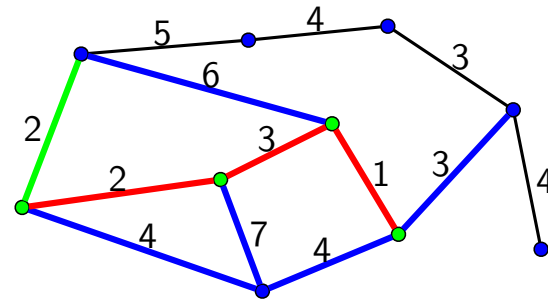
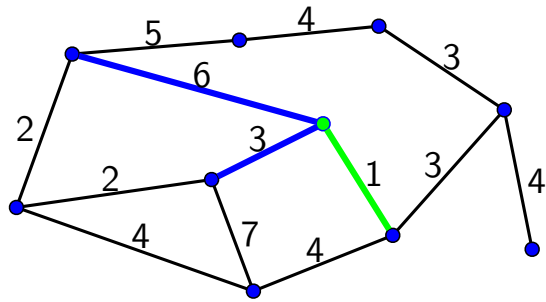
1. **Initialisierung:** Setze $X := \emptyset; U := \emptyset$.
2. **Wähle** beliebigen Knotenpunkt $v \in V$.
 $X \longleftarrow \{v\}; V \longleftarrow V \setminus \{v\}$.
3. Betrachte alle Kanten $(a, b) \in E \cap (X \times V)$ **Wähle** eine kürzeste Kante (a_1, b_1) . Dann gilt:

$$l(a_1, b_1) \leq l(a, b) \quad \text{für alle } (a, b) \in E \cap (X \times V)$$

4. $X \longleftarrow X \cup \{b_1\}; U \longleftarrow U \cup \{(a_1, b_1)\}; V \longleftarrow V \setminus \{b_1\}$.
5. **Falls** $V \neq \emptyset$ setze mit 3. fort;
6. **STOP.** *Output:* $T = (X, U)$.

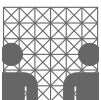


Beispiel: PRIM/DIJKSTRA-Alg.



Rechenzeit: PRIM/DIJKSTRA-Alg.

- Im Schritt 3. sind höchstens m Vergleiche auszuführen.
Schritt 3. wird $(n - 1)$ -mal aufgerufen.
 $\rightarrow T_A = O(m * n)$
- Die **untere Schranke** ist $\Omega(m)$, da jede Kante mindestens einmal geprüft werden muss.
- Etwas bessere Algorithmen existieren
($O(|E| + |V| \cdot \log |V|)$ durch Ausnutzung der verwendeten Datenstruktur).
- Die Suche nach dem besten Algorithmus wird durch die nachweisbar untere Schranke eingeschränkt!

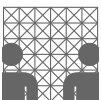


Korrektheit

Theorem: MINGERÜST liefert ein Minimal- Gerüst des zusammenhängenden Graphen G .

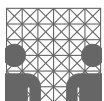
Beweis: Es werden sukzessive Teil-Unter-Graphen (*TU-Graphen*) von G erzeugt: $H_1, H_2, \dots, H_i, H_{i+1}, \dots, H_n$ mit $H_1 = (\{v\}, \emptyset)$ und $H_n = (V, U)$.

- Die TU-Graphen sind Bäume:
 - H_1 ist ein Baum.
 - H_i ist Baum $\Rightarrow H_{i+1}$ ist Baum
- Für $i = 1, 2, \dots, n - 1$: Ist H_i TU-Graph eines Minimalgerüsts von G , so gilt dies auch für H_{i+1} .



Korrektheit (2)

- T ist Gerüst von G
 $\Rightarrow \exists e_1 = (a_1, b_1) \in U. a_1 \in V_i, b_1 \in V \setminus V_i.$
- Ist \tilde{e} die Kante, die bei Erzeugung von H_{i+1} aus H_i ausgewählt wird:
 $\tilde{e} = e_1 \Rightarrow$ fertig.
 $\tilde{e} \neq e_1 \Rightarrow \tilde{e} = (\tilde{a}, \tilde{b}) \in E$ mit $\tilde{a} \in V_i$ und $\tilde{b} \in V - V_i.$
- \tilde{e} zu T hinzufügen. Es entsteht ein Kreis C .
- Wegen Schritt 3. muss $l(\tilde{e}) \leq l(e_1)$ gelten.
- Bilde $\tilde{T} = (T \setminus \{e_1\}) \cup \{\tilde{e}\} \Rightarrow l(\tilde{T}) \leq l(T)$. Minimalität von $T \Rightarrow \tilde{T}$ ist Minimalgerüst $\Rightarrow H_{i+1}$ ist TU-Graph eines Minimalgerüsts \tilde{T} von G . \square



Beispiel: Münzrückgabe

Mit dem Euro-Münzsatz:



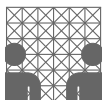
34¢ Wechselgeld:



Mit einem anderen Münzsatz:

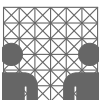


GREEDY



Fazit

- Es gibt Probleme, die sich mit *Greedy-Algorithmen* optimal lösen lassen.
- **Aber:** Es gibt auch Probleme, für die *Greedy-Algorithmen* ungeeignet sind!
- **Frage:** Lassen sich die Probleme, die durch *Greedy-Algorithmen* optimal gelöst werden, charakterisieren?
 - Ein *Greedy-Algorithmus* liefert immer eine optimale Lösung, wenn das Problem eine **matroidale Struktur** hat.
 - **Matroide** sind Verallgemeinerungen von Matrizen.

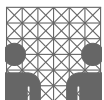


Vereinfachtes Rucksackproblem

Gegeben: Rucksack mit Maximalgewicht m ; Multimenge von Objekten, jeweils mit Gewicht und Wert $o : O \rightarrow \mathbb{N}$ mit Objektmenge $O = \{(t_1, g_1, w_1), \dots, (t_n, g_n, w_n)\}$

Gesucht: Füllung mit maximalem Wert

Antwort: Liste der Objekte im Rucksack

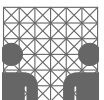


Beispiel: v. Rucksackproblem

i	Objektyp	g_i	w_i
1	Nägel	5,0 kg	8,50 €/kg
2	Heftpflaster	0,1 kg	50,00 €/kg
3	Whisky	1,5 kg	38,00 €/kg
4	Toilettenpapier	0,5 kg	4,50 €/kg
5	Käse	9,0 kg	14,50 €/kg
6	Brot	1,0 kg	2,20 €/kg

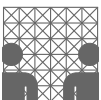
Maximales Rucksackgewicht: $m = 15kg$

Annahme: Alles ist beliebig portionierbar!



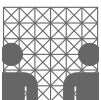
dynamische Programmierung

- Problemlösung durch Zerlegung in Teilprobleme und Zusammenfügen der Lösungen. (**Ähnlichkeit zu *divide and conquer***)
- Dynamische Programmierung eignet sich zur Lösung solcher Probleme, bei denen es sehr *viel weniger* Teilprobleme als mögliche Zerlegungen gibt.
- Häufig ist es günstig, **gefundene Teillösungen in einer Tabelle zu speichern**, so dass jedes Teilproblem nur einmal gelöst wird.
- Anwendung: z.B. *Vergleich des genetische Codes unterschiedlicher Arten von Lebewesen*



dynam. Prog.: Vorgehensweise

1. Bestimme die zu einer optimalen Gesamtlösung führenden Zerlegungen der Eingabe.
2. Definiere rekursiv den Wert der Zielfunktion für
 - die Gesamtlösung und
 - die in Betracht kommenden Teilaufgaben.
3. Berechne diese Werte für alle in Betracht kommenden Teilaufgaben und speichere sie ab.
4. Bestimme eine optimale Gesamtlösung unter Rückgriff auf die vorher berechneten Werte für benutzte Teilaufgaben.

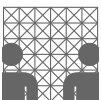


Matrizenmultiplikation

Eine optimale Folge von Entscheidungen erfordert optimale Teilfolgen!

Betrachten wir z.B. n Matrizen der Größe $d_{i-1} \times d_i$ mit $1 \leq i \leq n$.

- Gesucht wird $M = M_1 \cdot \dots \cdot M_n$ mit **minimaler Anzahl von Multiplikationen**.
- Die Matrizenmultiplikation ist assoziativ, also kann das Produkt auf verschiedene Arten berechnet werden, und man kann **durch geschickte Klammerung die Anzahl der skalaren Multiplikationen senken**.

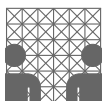


Beispiel: Matrizenmultiplikation

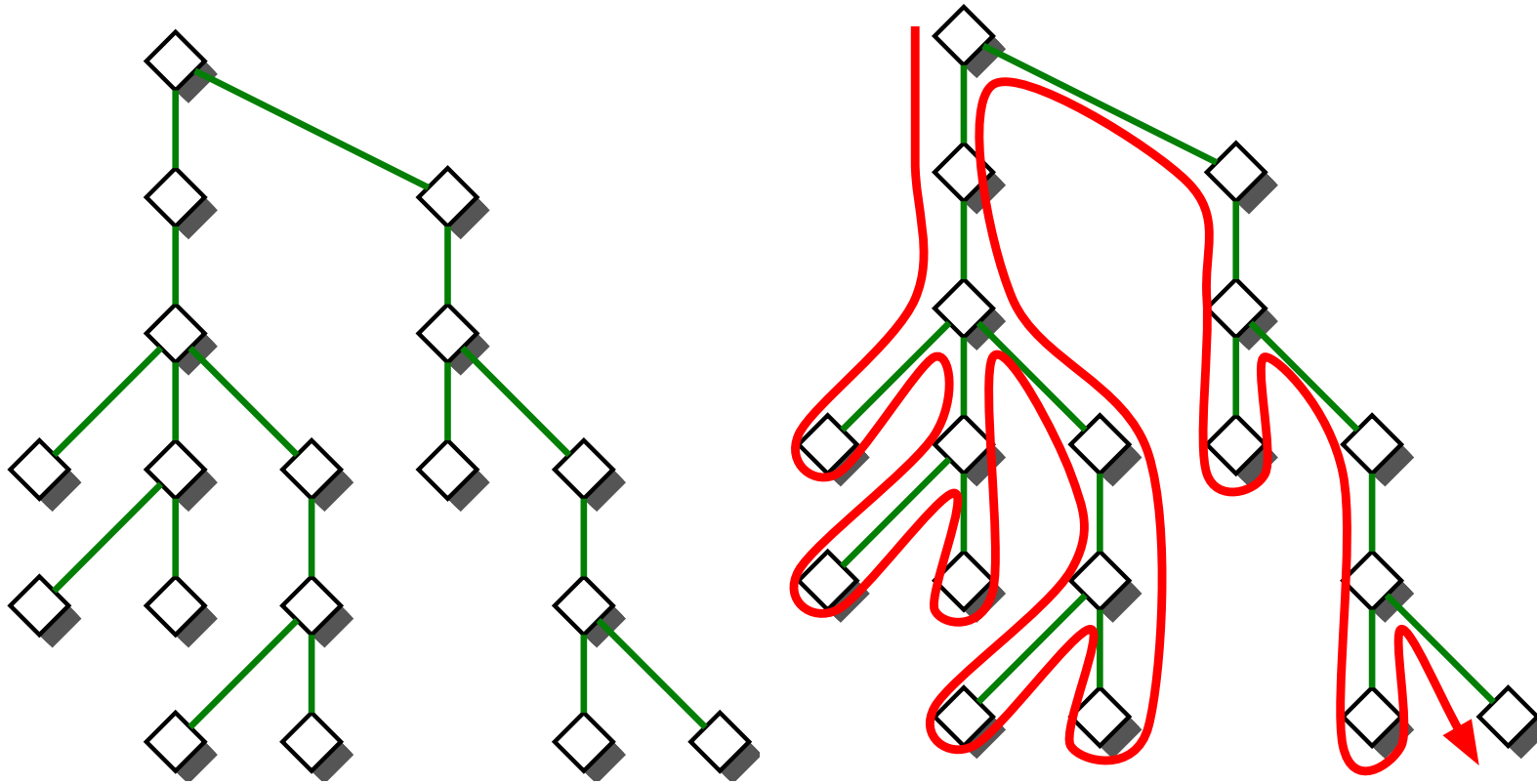
- 4 Matrizen A:[13,5], B:[5,89], C:[89,3] und D:[3,34]:

Klammerung	Anzahl der Multiplikationen
$A(B(CD))$	$9078+15130+2210 = 26418$
$A((BC)D)$	$1335+510+2210 = 4055$
$(AB)(CD)$	$5785+9078+39338 = 54201$
$(A(BC))D$	$1335+195+1326 = 2856$
$((AB)C)D$	$5785+3471+1326 = 10582$

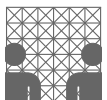
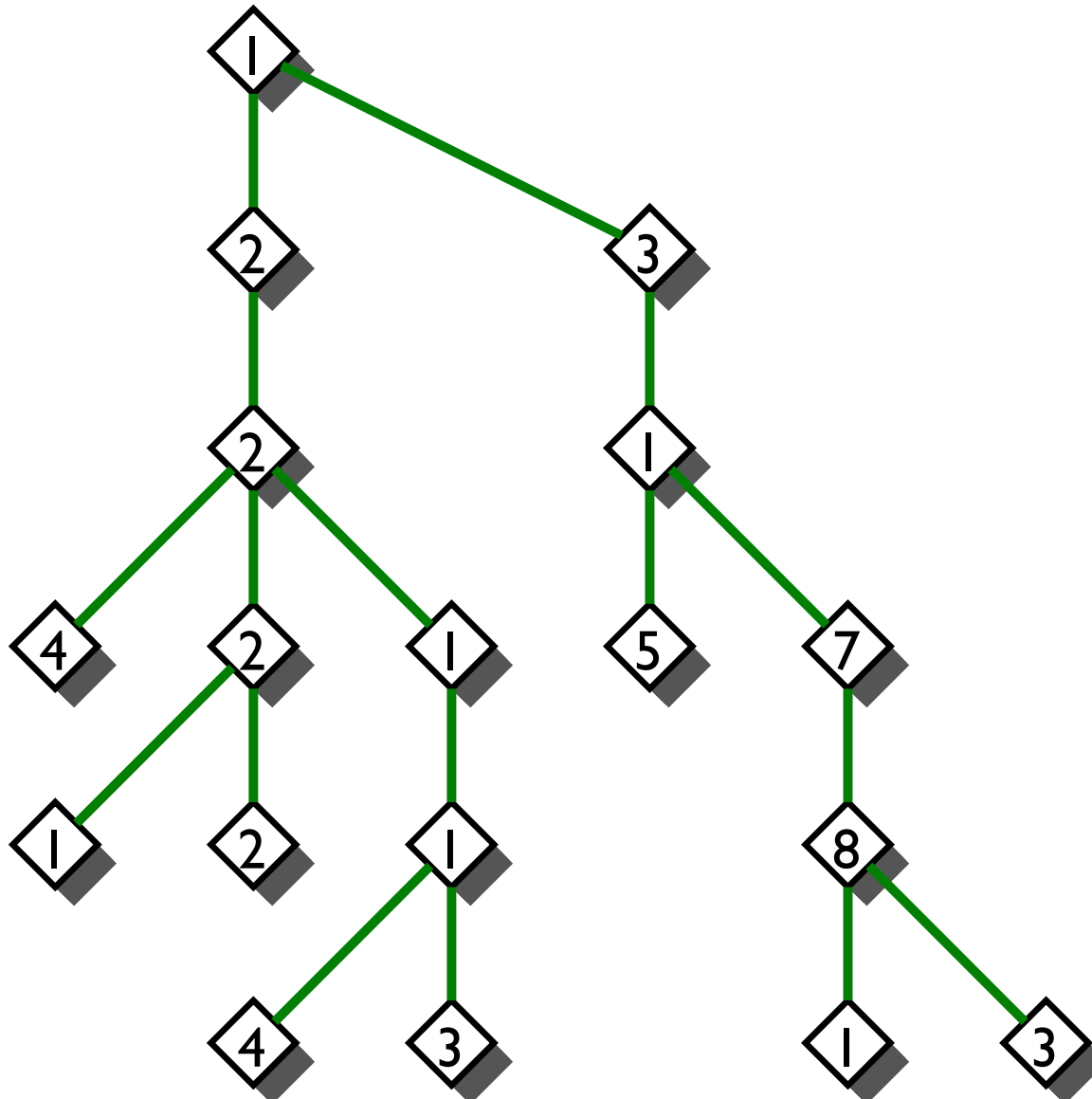
- beste Klammerung ist ca. 19mal schneller
- Lösungen gemeinsamer Teile werden in Tabelle gespeichert



Backtracking

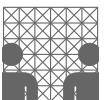


branch and bound



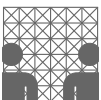
Ausblick

- noch einmal dynamische Programmierung:
 - Eine Lösung für das *Wechselgeldproblem*.
 - Das allgemeine *Rucksackproblem*.
 - Das *Travelling-Salesman-Problem*.
- weitere Algorithmentechniken



lineare Programmierung

- Viele Probleme sind durch **lineare Gleichungssysteme** charakterisiert
⇒ lineare Programmiermethoden
- Der **Lösungsraum** ist häufig auf **ganze Zahlen** oder gar **natürliche Zahlen** eingeschränkt!
- Das Auffinden einer Lösung wird schwieriger!
- **Konsequenz:** Betrachtung von linearer Programmierung mit **Ganzzahllösungen**.
- **Problem:** Effizientes Auffinden der gesuchten Lösung(en).

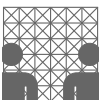


Beispiel: Investitionsplanung

Problem: Der Betrag von € 14000 soll möglichst gewinnbringend investiert werden.

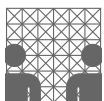
#	Investitionssumme	Wert
(1)	€ 5000	€ 8000
(2)	€ 7000	€ 11000
(3)	€ 4000	€ 6000
(4)	€ 3000	€ 4000

Maximiere $8x_1 + 11x_2 + 6x_3 + 4x_4$ mit den **Einschränkungen**
 $5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14$ und $x_i \in \{0, 1\}$ für alle
 $i \in \{1, 2, 3, 4\}$.



Lösung

- Standardverfahren liefern als Lösung: $x_1 = x_2 = 1$,
 $x_3 = 0,5$ und $x_4 = 0$ mit dem erreichten Wert € 22000.
- Keine ganzzahlige Lösung!
- Abrunden ergibt nur einen Wert von € 19000.
- Optimale ganzzahlige Lösung ist € 21000.
($x_2 = x_3 = x_4 = 1$ und $x_1 = 0$)
- Weitere Einschränkungen formalisierbar:
 - höchstens zwei Investitionen: $\sum x_i \leq 2$
 - wird in (2) investiert, so auch in (4): $x_2 - x_4 \leq 0$
 - wird in (1) investiert, dann nicht in (3): $x_1 - x_3 \leq 1$



allgemeines Rucksackproblem

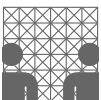
Gegeben: Menge von Paaren $(l_i, d_i) \in \mathbb{N} \times \mathbb{N}, 1 \leq i \leq m$ sowie **Kapazität** $L \in \mathbb{N}$ des Rucksacks. Hierbei ist l_i die **Größe** und d_i der **Wert** des i -ten Objekts.

Gesucht: Lösung des Optimierungsproblems

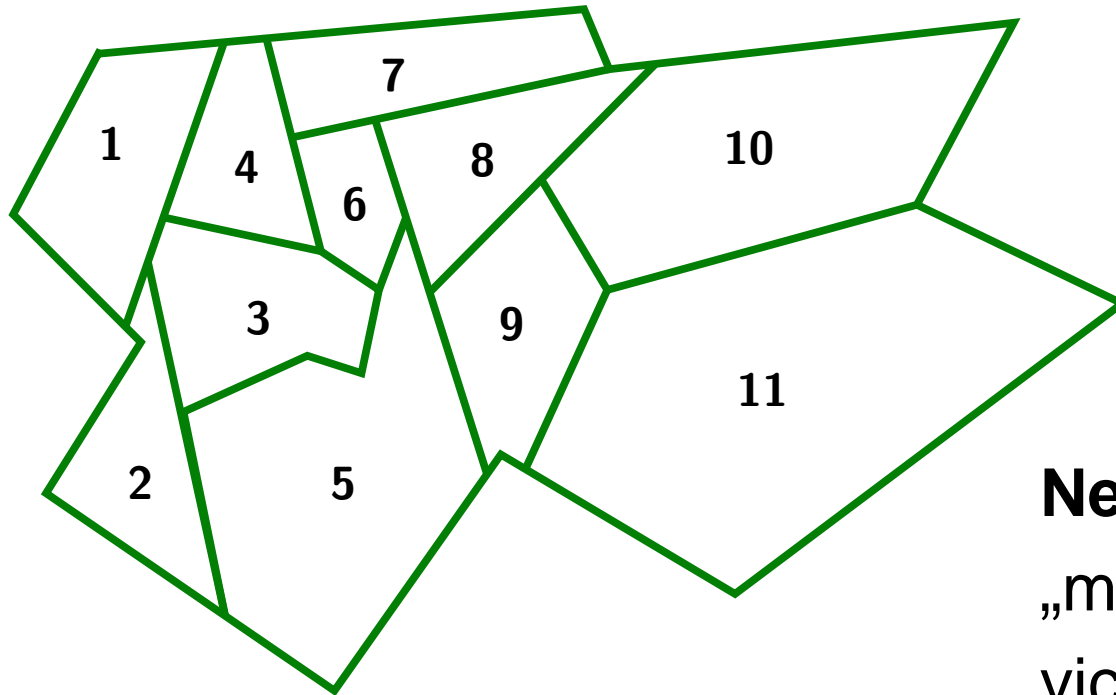
1. $w = \sum_{i=1}^m d_i x_i$ ist maximal.

2. Nebenbedingung: $\sum_{i=1}^m l_i x_i \leq L.$

Antwort: Vektor $\mathbf{x} = (x_1, x_2, \dots, x_m) \in \mathbb{N}^m$



Servicepoints



Minimiere

$$\begin{aligned} & x_1 + x_2 + x_3 + x_4 + \\ & x_5 + x_6 + x_7 + x_8 + \\ & x_9 + x_{10} + x_{11}. \end{aligned}$$

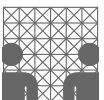
Nebenbedingungen:

„mindestens ein Servicepoint in der Nachbarschaft“

$$x_1 + x_2 + x_3 + x_4 \geq 1 \quad x_1 + x_2 + x_3 + x_5 \geq 1$$

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \geq 1 \quad \text{USW.}$$

Instanz des **set coverability problem**



Travelling Salesperson Problem

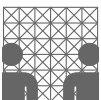
Gegeben: Menge von Orten (oder Knoten)
 $V = \{1, 2, \dots, n\}$ und eine $(n \times n)$ -
Entfernungsmatrix $D = (d_{i,k}) \in \mathbb{N}^{n \times n}$

Gesucht: Eine Rundreise (Hamiltonkreis) K
durch mindestens (exakt) alle n Orte
von V mit minimaler Gesamtlänge.

Antwort: Wegbeschreibung (Folge von Knoten)

Minimiere $\sum_{i=1}^n \sum_{j=1}^{i-1} d_{i,j} x_{i,j}$ mit

Nebenbedingung $\sum_{i \in S} \sum_{j \notin S} x_{i,j} \geq 2$ für alle $S \subset V$

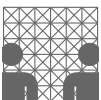


TSP als lin. Programmierproblem

- sehr große Anzahl von Nebenbedingungen

# Städte	# Nebenbedingungen
20	524288
300	1018517988167243043134 2228442046890805257341 9683296812531807022467 7190649881668353091698 688

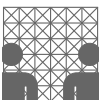
- Dennoch ist dies derzeit für Instanzen zwischen 100 und 1000 Knoten der beste bekannte Ansatz!!!



Lösbarkeit des TSP

- mit dynamischer Programmierung:
 - Teilwege berechnen,
 - Zwischenergebnisse in Tabelle ablegen,
 - Kombination mit *branch and bound*
- konkretes Beispiel im Skript

Exkurs: reines **Backtracking** z.B. zur Lösung des 8-Königinnen-Problems.



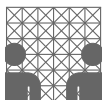
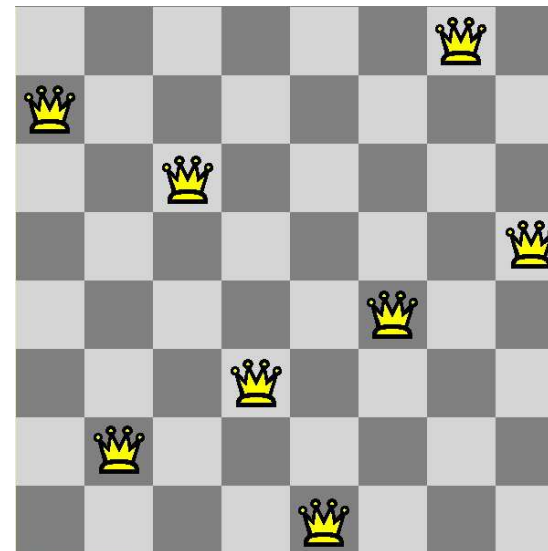
8-Königinnen-Problem

Gegeben: Ein Schachbrett (8×8 Felder), eine beliebige Anzahl von Königinnen

Gesucht: Stellungen, so dass sich keine zwei Königinnen bedrohen.

Antwort: Anzahl solcher Stellungen

Es können niemals mehr als 8 sein \Rightarrow 8 Königinnen-Problem



IP vs. LR

Integer-Programmierung (IP)

Minimiere

$$cx$$

Nebenbedingungen

$$Ax = b$$

$$x \geq 0 \text{ und } x \text{ ist ganzzahlig}$$

Lineare-Programmierung (LR) *engl. linear relaxation*

Minimiere

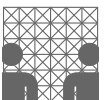
$$cx$$

Nebenbedingungen

$$Ax = b$$

$$x \geq 0$$

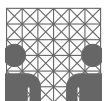
(analog für Maximierungsprobleme)



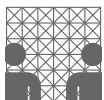
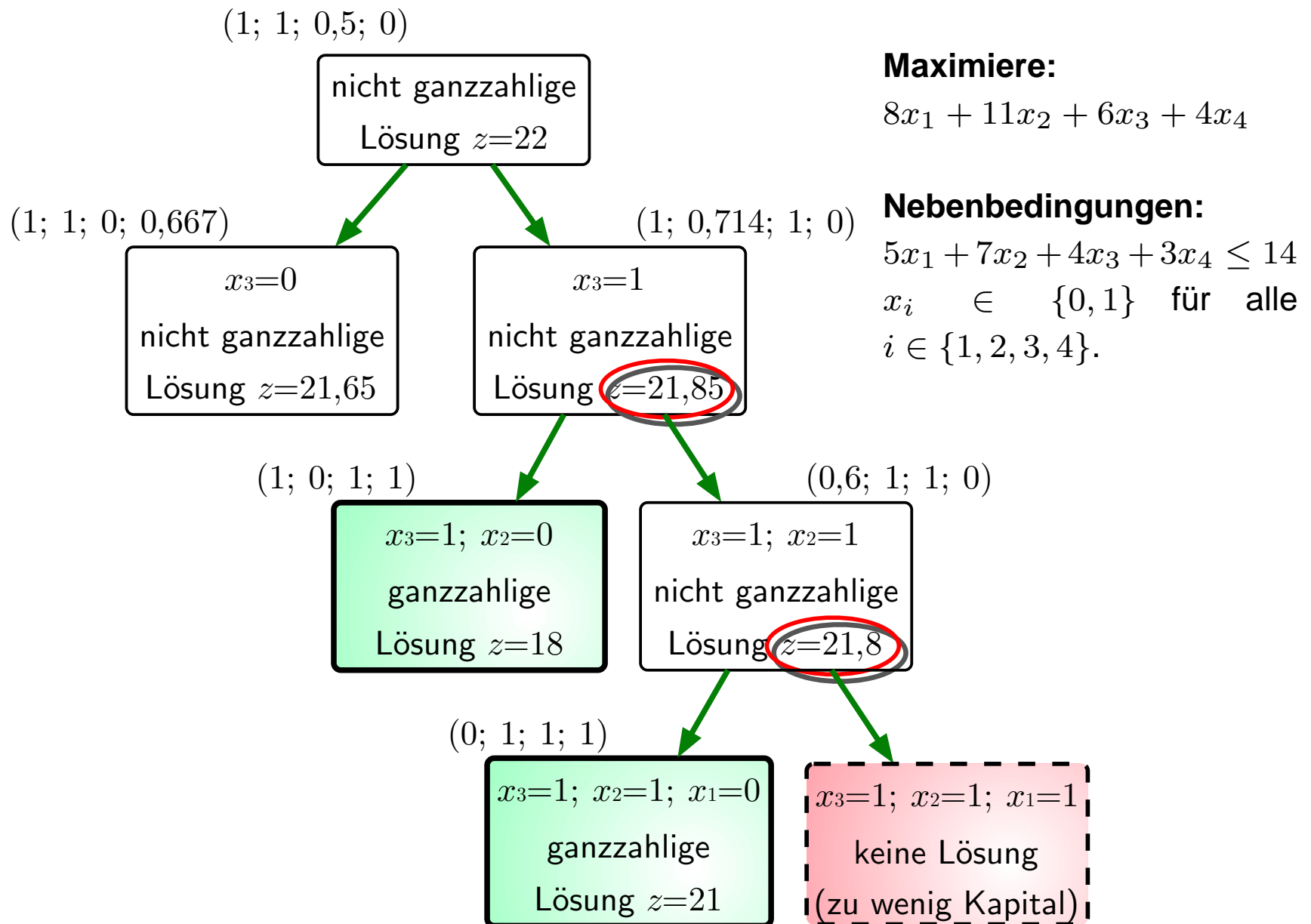
IP vs. LR (2)

Da LR weniger Nebenbedingungen hat als IP gilt:

- Ist IP eine Minimierung (Maximierung), so ist der optimale Wert für LR \leq (\geq) dem für IP.
- Gibt es für LR keine optimale Lösung, so gibt es auch für IP keine.
- Ist die Lösung von LR ganzzahlig, so ist IP lösbar mit genau dieser optimalen Lösung.
- Aufrunden (Abrunden) der optimalen Lösung für LR liefert einen Wert \leq (\geq) dem Optimum für IP bei einem Minimierungs-/Maximierungsproblem.

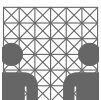


Investitionen (branch and bound)



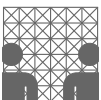
branch and bound (Zusammenf.)

1. **Löse LR-Variante des Problems.** Falls Lösung ganzzahlig \rightarrow fertig, ansonsten erzeuge Subprobleme bzgl. der Belegung einer kritischen Variablen.
2. Ein **Subproblem ist nicht aktiv** falls
 - (a) es bereits für eine Verzweigung verwandt wurde,
 - (b) alle Variablen der Lösung ganzzahlig sind,
 - (c) das Subproblem nicht ganzzahlig lösbar ist,
 - (d) ein *bounding*-Argument es ausschließt.
3. **Wähle ein aktives Subproblem und verzweige** bzgl. einer kritischen Variable. **Wiederhole** dies bis keine aktiven Subprobleme mehr vorhanden sind.

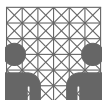
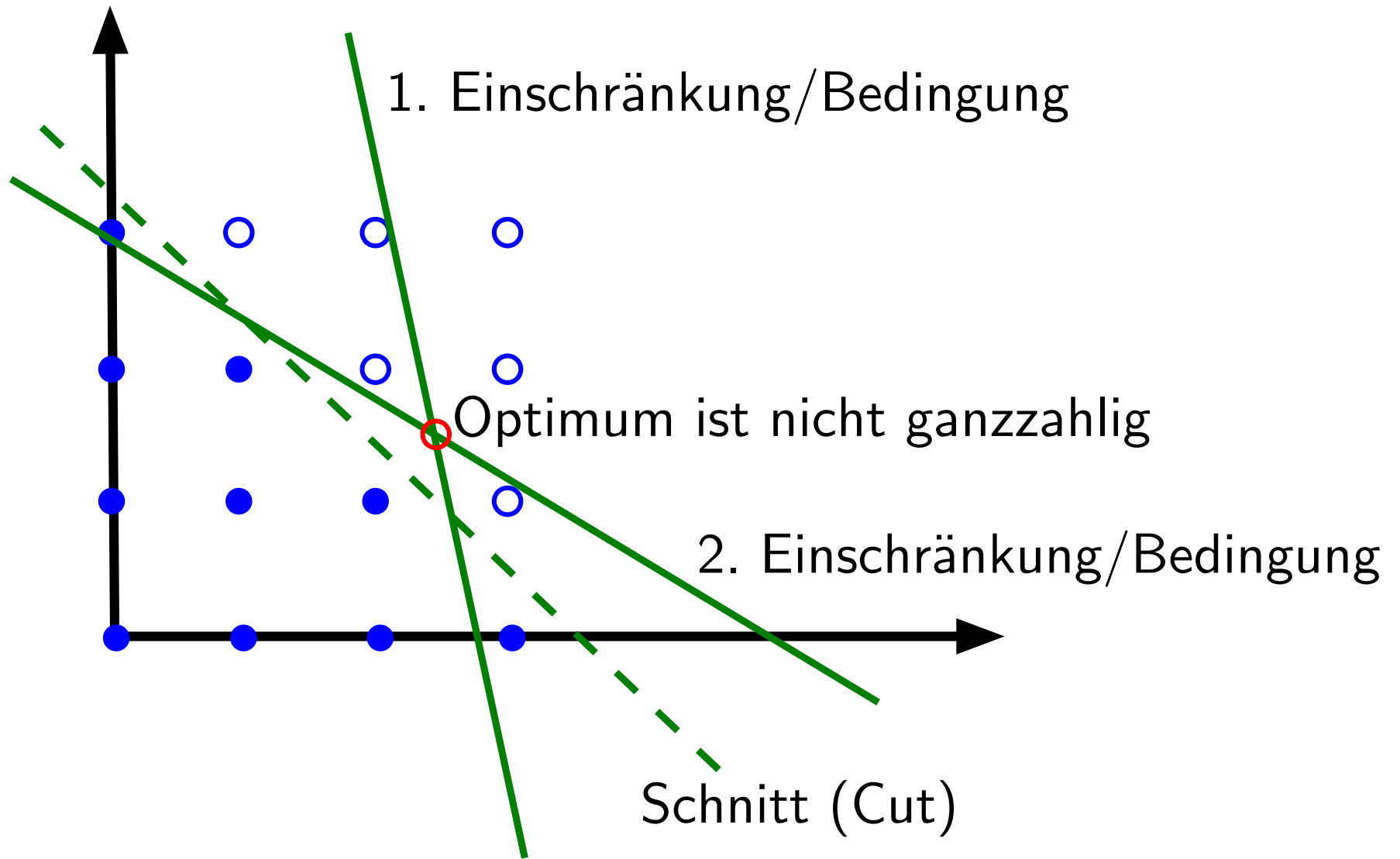


Fazit: branch and bound

- Branch-and-bound Verfahren sind im schlechtesten Falle exponentiell.
- Es ist möglich, dass die optimale Lösung erst gefunden wird, wenn alle möglichen Verzweigungen abgesucht worden sind.
 - Ein vollständiger binärer Verzweigungsbaum der Tiefe n hat 2^n Blätter.
- Auch bei der branch-and-bound-Lösung des TSP bringt die detaillierte Analyse nicht viel, denn wir wissen aus der NP-Vollständigkeit dieses Problems, dass es zur Zeit kein deterministisches polynomiales Verfahren zu dessen Lösung gibt.



Alternativen: Ebenen

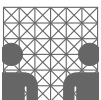


Alternativen: Heuristiken

- Der Begriff **Heuristik** stammt von dem griechischen Mathematiker *Archimedes* (\approx 285 - 212 v.Z.) und erinnert an seinen berühmten Ausspruch:

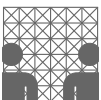
Ευρηκα - Heureka! ("Ich hab's gefunden")
(heurisko (griech.): ich finde).

- Eine **Heuristik** (ein **heuristisches Verfahren**) ist ein Algorithmus, der (im allgemeinen recht gute) Lösungen für Problem-Beispiele erzeugt, aber einer exakten Analyse nicht (oder nur sehr schwer) zugänglich ist. („Die Verfahren funktionieren, aber man weiß nicht genau warum!“).



Heuristik vs. Approximation

- Viele lokale Suchverfahren sind Heuristiken. Weitere Heuristiken sind:
 - Simulated Annealing,
 - Tabu Search,
 - Genetische Algorithmen.
- **Approximationsverfahren** sind Algorithmen zur Erzeugung von Lösungen, die i.a. nur suboptimal sind, aber der optimalen Lösung (nachweislich) nahekommen.



diskrete Optimierung

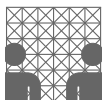
- Ein ein DOP (diskretes Optimierungsproblem) P ist gegeben durch:
 - \mathcal{I} : Menge der **Instanzen** (Beispiele),
 - \mathcal{L} : Menge der **Lösungen**,
 - w : **Wertfunktion (Zielfunktion)** $w : \mathcal{L} \rightarrow \mathbb{R}$.
- $s^*(I)$ heißt **optimale Lösung** für I : \Leftrightarrow
 - Min-Problem: $w(s^*(I)) \leq w(s(I))$ für alle Lösungen $s(I)$ von $I \in \mathcal{I}$,
 - Max-Problem: $w(s^*(I)) \geq w(s(I))$ für alle Lösungen $s(I)$ von $I \in \mathcal{I}$.
- **Kurz:** $OPT(I) := w(s^*(I))$.



PZ-Algorithmus

- Ein Algorithmus \mathcal{A} heißt **Polynomial-Zeit-Approximationsalgorithmus** (PZ-Algorithmus) für P genau dann, wenn \mathcal{A} für jedes $I \in \mathcal{I}$ in polynomialer Zeit eine Lösung $s_{\mathcal{A}}(I)$ von I erzeugt:

$$\mathcal{A}(I) := w(s_{\mathcal{A}}(I)).$$

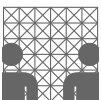


Gütemaß für Approx.alg.

P ist Min-Problem: $R_{\mathcal{A}}(I) = \frac{\mathcal{A}(I)}{OPT(I)}$

P ist Max-Problem: $R_{\mathcal{A}}(I) = \frac{OPT(I)}{\mathcal{A}(I)}$

- Es gilt stets: $1 \leq R_{\mathcal{A}}(I) \leq +\infty$ für alle $I \in \mathcal{I}$ und alle Algorithmen für P .
- \mathcal{A} heißt ein **Optimierungsalgorithmus** für P genau dann, wenn gilt: $\mathcal{A}(I) = OPT(I)$ für alle $I \in \mathcal{I}$.
- Wir betrachten: $R_{\mathcal{A}} := \inf\{c \mid R_{\mathcal{A}}(I) \leq c \text{ für alle } I \in \mathcal{I}\}$
 $\rightarrow 1 \leq R_{\mathcal{A}} \leq +\infty$.
- $R_{\mathcal{A}}$ kann nur selten exakt bestimmt werden. Einfacher kann es sein, $R_{\mathcal{A}}$ abzuschätzen.



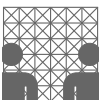
gut / schlecht approximierbar

Man teilt diskrete Optimierungsprobleme (DOP) ein in:

- **gut approximierbare DOP** P : es gibt für P Algorithmen A mit R_A "nahe" bei 1,
- **schlecht approximierbare DOP** sind Probleme, für die es *nachweislich* keine guten Approximationsalgorithmen gibt.

Beispiel: ΔTSP (D symmetrisch)

ΔTSP ist ein Spezialfall des TSP , und zwar soll für D die Dreiecksungleichung gelten: $D = (d_{i,k})$, und für alle i, k, l mit $1 \leq i, k, l \leq n$ sei: $d_{ik} + d_{kl} \geq d_{il}$.



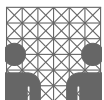
Algorithmus für ΔTSP

Gegeben: $I \in \Delta TSP$

$G = (V, D)$, $E = V \times V$, $D : E \rightarrow \mathbb{R}^+ \cup \{\infty\}$

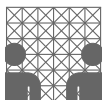
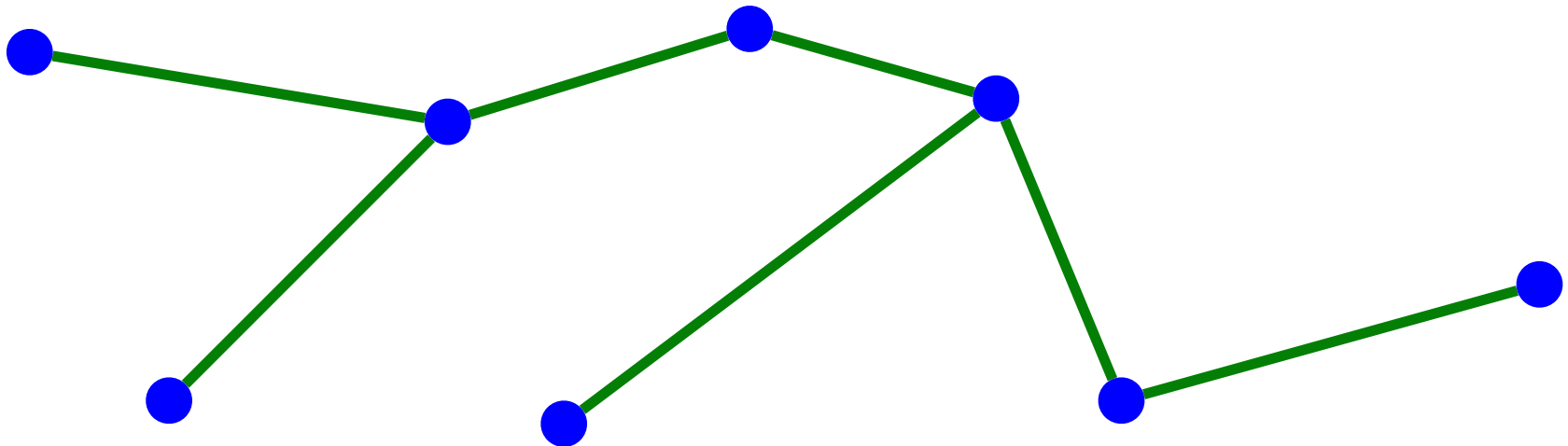
Entfernungsmatrix D erfülle die Δ -Ungleichung.

1. Bestimme ein **Minimalgerüst** H von G .
2. Verdopple die Kanten von H
 \Rightarrow **Multigraph** H' ist **Eulergraph** (gerader Knotengrad).
 $\Rightarrow H'$ besitzt **Eulerkreis** C' . Bestimme C' (Zeit: $O(m)$)
3. Erzeuge aus C' durch Anwendung der Δ -Ungleichung einen Hamiltonkreis C^* von G , indem bereits erfaßte Knotenpunkte übersprungen werden.
4. STOP: Output: C^* mit $l(C^*) = \mathcal{A}(I)$.



Beispiel

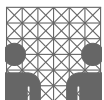
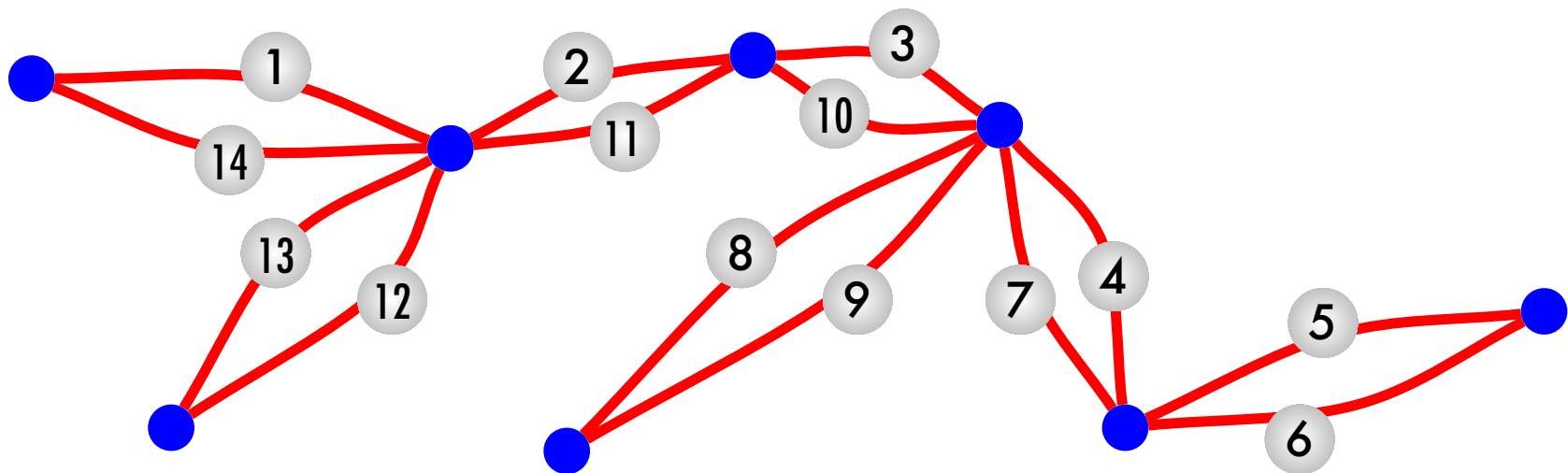
Sei dies ein Minimalgerüst H eines Graphen G :



Beispiel (2)

Wir verdoppeln die Kanten von H und erhalten einen Eulergraphen H' mit dem Eulerkreis C' , für den die Reihenfolge der Kanten hier angegeben ist:

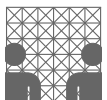
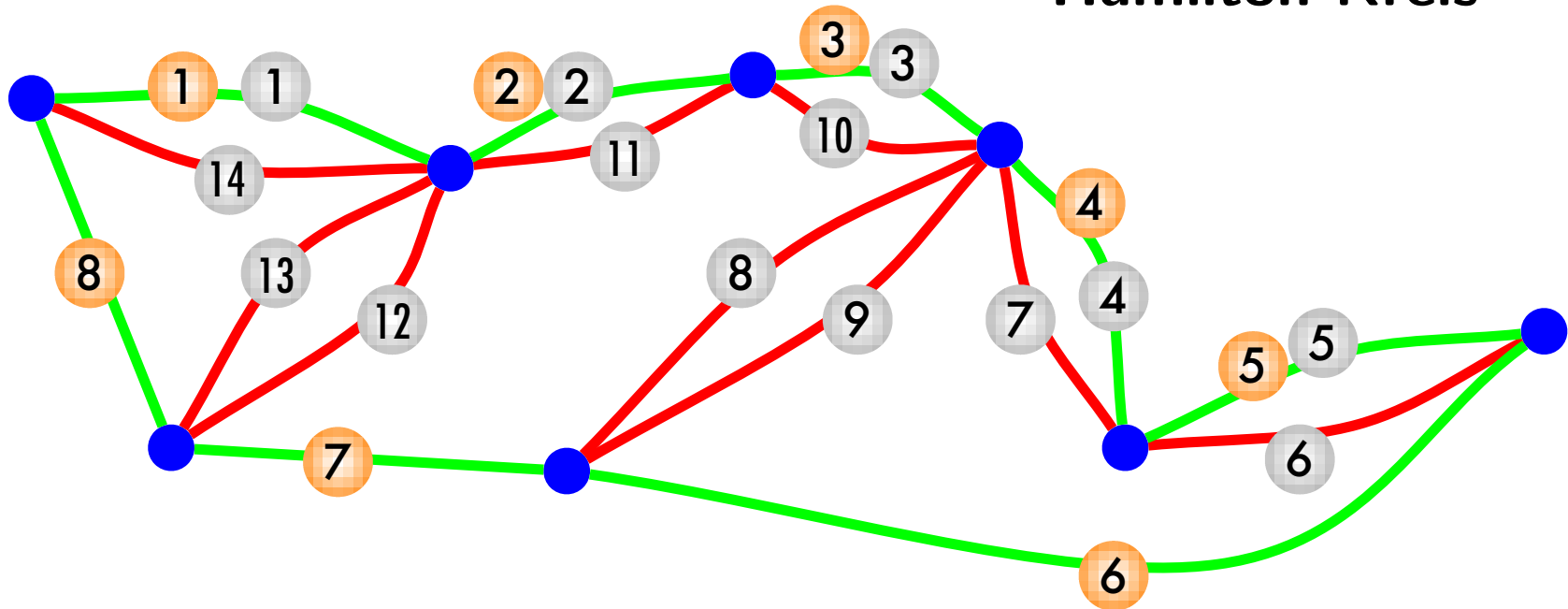
Euler-Kreis



Beispiel

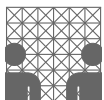
Aus C' konstruieren wir den Hamiltonkreis C^* :

Hamilton-Kreis



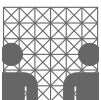
gut / schlecht approximierbar (2)

- Es gilt: $l(H) \leq OPT(I) \leq l(C^*) \leq l(C') = 2 \cdot l(H)$.
- Folglich ist $OPT(I) \leq \mathcal{A}(I) \leq 2 \cdot OPT(I)$, also $R_A(I) = \frac{\mathcal{A}(I)}{OPT(I)} \leq 2$ und somit $R_A \leq 2$.
- Dieses Ergebnis kann kaum verbessert werden, denn es gelten die folgenden Ergebnisse.
 - **Theorem:** Das ΔTSP -Problem ist NP-vollständig. (d.h. es ist ein sehr schwieriges Problem, für das bislang bestenfalls Exponentialzeitalgorithmen existieren!)



gut / schlecht approximierbar (3)

- **Theorem:** Gäbe es für jedes $\epsilon > 0$ einen PZ-Approximationsalgorithmus \mathcal{A} für TSP, der für jede symmetrische Entfernungsmatrix $D \in \mathbb{N}^{n \times n}$ in polynomialer Zeit eine Tour $T_{\mathcal{A}}$ liefert, mit der Länge $\mathcal{A}_{T_{\mathcal{A}}}(I)$, für die $R_{\mathcal{A}} \leq 1 + \epsilon$ gilt, dann ist HAMILTONKREIS $\in \mathcal{P}$, d.h. dann ist $\mathcal{P} = \mathcal{NP}$.
- Also ist TSP vermutlich (d.h. unter der Annahme, dass $\mathcal{P} \neq \mathcal{NP}$ gilt) ein schlecht-approximierbares Problem.
- **Beispiel für ein gut-approximierbares Problem:**
Für das Rucksackproblem gibt es einen PZ-Algorithmus \mathcal{A} mit $R_{\mathcal{A}} \leq 1 + \epsilon$ für beliebige $\epsilon \geq 0$.



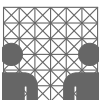
Konstruktion von Wegen

Nearest neighbour: Start-Ort wird zufällig gewählt und es wird immer die nächste noch nicht besuchte Stadt besucht bis die letzte erreicht ist. Dann zurück zum Start-Ort.

Nearest insertion: Beginn mit einer möglichst *kurzen* Strecke zwischen zwei Orten. Dann: Entfernen jeweils einer Kante mit Einfügen einer Stadt als Zwischenstop. Wähle die Stadt so, dass *die Strecke wenig erhöht* wird!

Furthest insertion: Beginn mit möglichst *langer* Strecke. Dann: wie oben, aber wähle die Stadt und Kante so, dass *die Strecke größtmöglich erhöht* wird!

Sweep: Markiere den Mittelpunkt der Landkarte. Auswahl der nächsten Stadt, wie ein Radarstrahl.



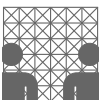
Optimierung von Wegen

Two-opt: Systematischer Vergleich jeweils zweier Kanten und ggf. Austausch gegen günstigere Variante.

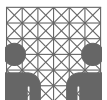
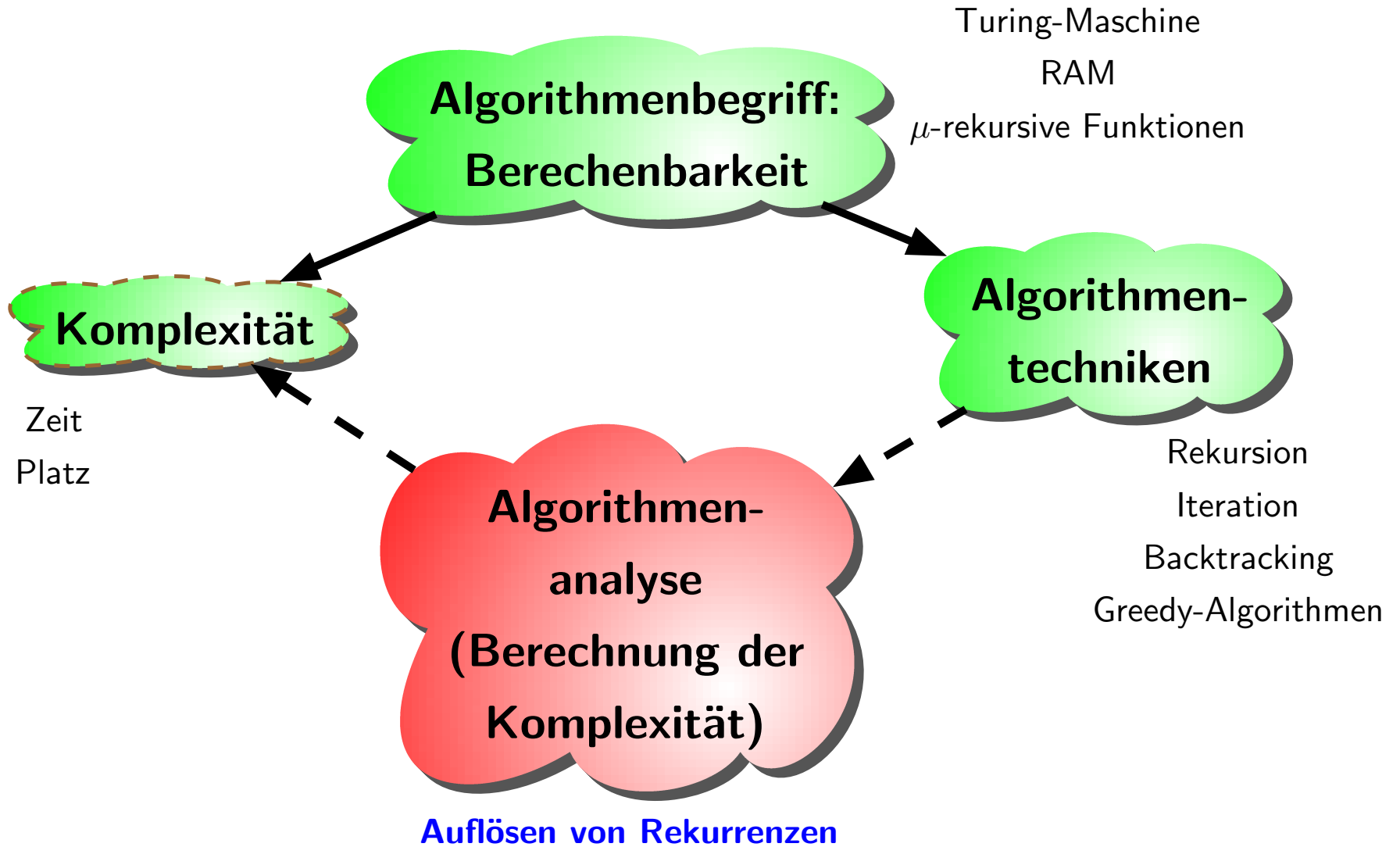
Three-opt: analog Two-opt, aber mit je drei Kanten.

Lin-Kernighan: Entfernen einer Kante aus Rundtour liefert einen Pfad. Ein Ende mit innerem Knoten verbinden und andere Kante löschen, so dass wieder ein Pfad vorliegt.

Wiederholung dieser Prozedur solange die *gain sum* (gelöschte minus eingefügte Kantengewichte) positiv ist und noch nicht behandelte Kanten existieren. Merke jeweilige Kosten für wiederhergestellte Rundtour. Wähle am Ende die Beste!



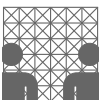
Über-/Rückblick



Einige Feststellungen

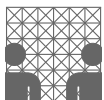
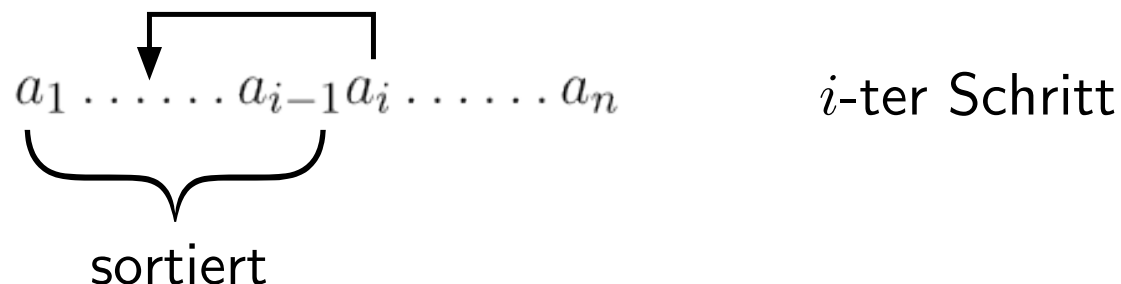
- *Mergesort* hat eine *worst-case*-Zeitkomplexität von $O(n \log n)$.
- *Insertsort* hat eine *worst-case*-Zeitkomplexität von $\Theta(n^2)$.
- Somit ist *Mergesort* *asymptotisch schneller* als *Insertsort*.

Manchmal ist es möglich, die exakte Zeit für einen gegebenen Computer festzustellen, aber das ist meistens nicht lohnenswert.



Asymptotische Analyse

- Analyse der Hauptidee des Algorithmus.
- **Mergesort** ist ein *divide-and-conquer*-Algorithmus:
 - Ein Problem wird auf zwei Probleme etwa halber Größe reduziert (mit linearem Zeitaufwand für *decomposition* und *composition*). $\Rightarrow \Theta(n \log n)$
- **Insertsort** ist ein Sortieralgorithmus, für den nach dem i -ten Zyklus die ersten i Elemente der Folge sortiert sind und im i -ten Schritt das i -te Element in die richtige Position gebracht wird.



Analyse (2)

- Vorsicht ist bei der asymptotischen Analyse geboten – besonders, wenn die Komplexität nicht nur vom Umfang der Eingabedaten abhängt.

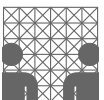
- Für die Zeitkomplexität von Insertsort gilt:

$$T(n) \in \Omega(n)$$

$$T(n) \in O(n^2)$$

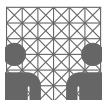
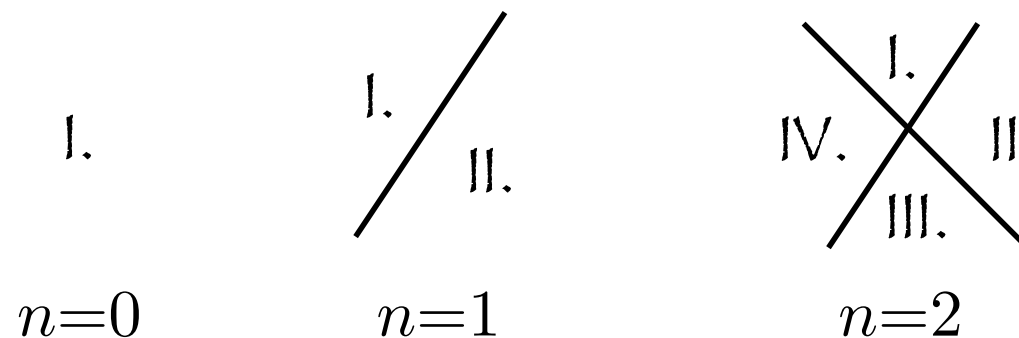
- Kann man nun schreiben „ $T(n) \in \Omega(n^2)$ “ ?
- Nein! Es gibt Eingaben, für die Insertsort $\Theta(n)$ Zeit braucht. Aber für die *worst-case*-Zeitkomplexität gilt:

$$T_{worst}(n) \in \Omega(n^2)$$



Einführungsbeispiel: Ebenen

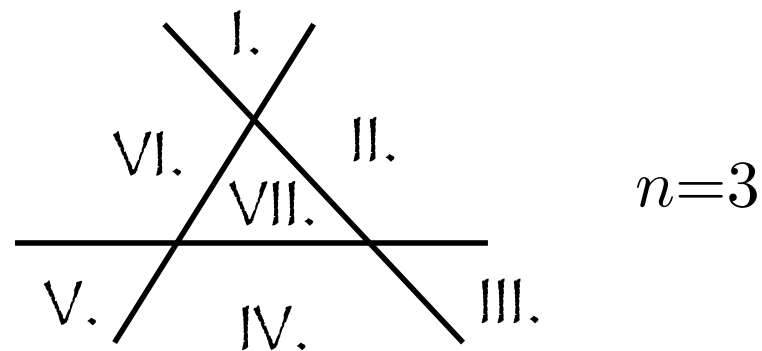
- In maximal wieviele Gebiete zerteilen n Geraden die euklidische Ebene?
- **keine Gerade ($n = 0$):** das Gebiet bleibt unverändert. Also $L_0 = 1$.
- **eine Gerade ($n = 1$):** zwei Gebiete, d.h. $L_1 = 2$.
- **zwei Geraden ($n = 2$):** vier Gebiete, d.h. $L_2 = 4$.



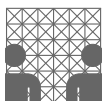
Ebenen (2)

Die Vermutung $L_n = 2^n$, die für $n = 0, 1, 2$ stimmt, wird durch L_3 widerlegt:

- Für $n = 3$ ergibt sich folgendes Bild:



- Also: $L_3 = 7$.
- Ist die neue Gerade zu keiner anderen parallel, so schneidet sie alle vorherigen Geraden genau einmal (höchstens in $n - 1$ verschiedenen Punkten).
- Erweiterung der bisherigen L_{n-1} Gebiete um höchstens n neue!



Rekurrenzgleichung

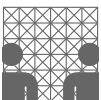
- Wenn die n -te Gerade *nicht* durch einen früheren Schnittpunkt geht, so wird die Maximalzahl erreicht. Es ergibt sich die Rekurrenz $L_n := L_{n-1} + n$ für die gesuchte Zahl L_n mit dem Anfangswert $L_0 := 1$.
- Um nun eine geschlossene Formel für L_n zu gewinnen, wickeln wir die ersten Rekurrenzen ab:

$$L_0 = 1$$

$$L_1 = L_0 + 1 = 1 + 1 = 2$$

$$L_2 = L_1 + 2 = (L_0 + 1) + 2 = 1 + 1 + 2 = 4$$

$$\begin{aligned} L_3 &= (L_1 + 2) + 3 = ((L_0 + 1) + 2) + 3 \\ &= 1 + 1 + 2 + 3 = 7 \end{aligned}$$



Summenformel

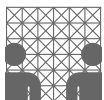
- Die offensichtliche Vermutung ist also

$$L_n = 1 + \sum_{i=1}^n i$$

- Diese Vermutung muss aber noch formal bewiesen werden, was wir durch vollständige Induktion tun werden.

Die **Verankerung** für $n = 0$ ergibt sich wie gewünscht:

$$L_0 = 1 + \sum_{i=1}^0 i = 1$$



Summenformel (2)

Induktionsannahme:

$$L_m = 1 + \sum_{i=1}^m i \quad \text{für ein festes } m \in \mathbb{N}$$

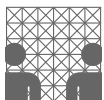
Induktionsschritt:

$$L_{m+1} = L_m + (m + 1) \quad (\text{entsprechend der Rekurrenz})$$

$$= 1 + \sum_{i=1}^m i + (m + 1) \quad (\text{nach Induktionsannahme})$$

$$= 1 + \sum_{i=1}^{m+1} i \quad (\text{trivial})$$

Damit ist die Formel $L_n = 1 + \sum_{i=1}^n i$ bewiesen.



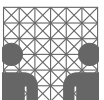
geschlossene Formel

- Die Summenformel ist keine *geschlossene* Formel!
- Mit Hilfe der Gaußschen Formel $\sum_{i=1}^m i = \frac{n(n+1)}{2}$ erhalten wir nun eine geschlossene Formel für L_n :

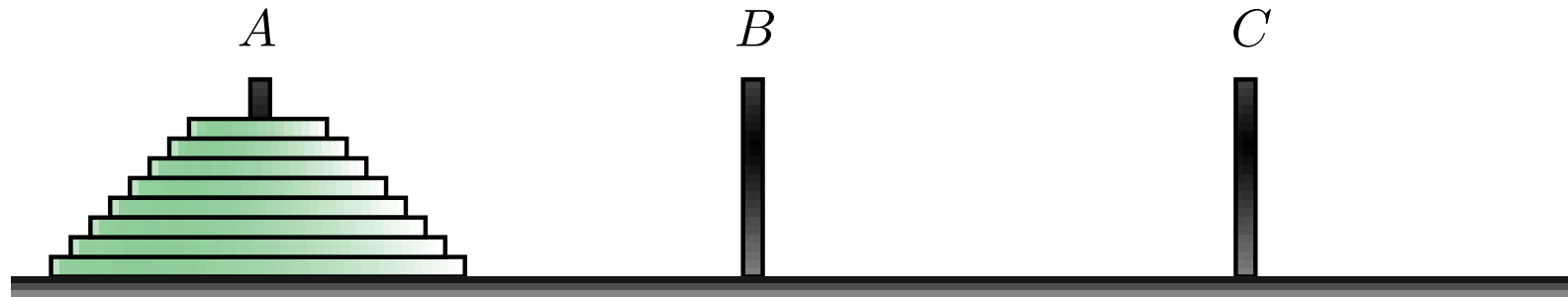
$$L_n = 1 + \frac{n(n+1)}{2}$$

Grundsätzliches Vorgehen beim Lösen von Rekurrenzgleichungen: Zuerst die Werte für kleine Argumente berechnen. Diese Werte können helfen

1. die Lösung zu finden,
2. die Lösung zu verifizieren.



2. Beispiel: Türme von Hanoi

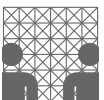


Ausgangssituation:

Seien drei Stangen gegeben und n Scheiben, die der Größe nach sortiert auf Stange A liegen.

Aufgabe:

Die Scheiben von Stange A sollen auf Stange C verschoben werden, wobei in jedem Schritt nur eine Scheibe bewegt und niemals eine größere Scheibe auf eine kleinere gelegt werden kann.



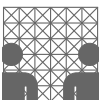
rekursiver Algorithmus

1. Verschiebe die $n - 1$ oberen Scheiben von A auf B.
2. Verschiebe die größte Scheibe von A auf C.
3. Verschiebe alle $n - 1$ Scheiben von B auf C.

Verschiebungs-Analyse: Sei $T(n)$ die Anzahl der Verschiebungen, die obiger Algorithmus braucht, um n Scheiben zu verschieben. Es gilt

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T(n - 1) + 1 & \text{falls } n > 1 \end{cases}$$

Frage: Kann man das schneller tun? **Antwort:** Nein.



Hanoi-Rekurrenz

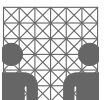
Rekurrenzgleichung für die Türme von Hanoi:

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T(n-1) + 1 & \text{falls } n \geq 2 \end{cases}$$

Tabellarische Darstellung:

n	1	2	3	4	5	6	7	8	9	10
$T(n)$	1	3	7	15	31	63	127	255	511	1023
2^n	2	4	8	16	32	64	128	256	512	1024

Vermutung: $T(n) = 2^n - 1$?



Beweis

Wir betrachten zwei Beweismöglichkeiten:

1. vollständige Induktion
 2. Reduzieren auf Summen
-

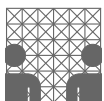
Induktionsbeweis für: $T(n) = 2^n - 1$ ist Lösung.

Induktionsanfang: Für $n = 1$ gilt $T(1) = 2^1 - 1 = 1$.

Induktionsschritt: Nach der *Induktionsannahme* sei für ein gegebenes $n \geq 1$ $T(n) = 2^n - 1$ eine Lösung der Rekurrenzgleichung .

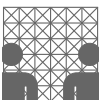
Nach *Rekurrenzgleichung*:

$$T(n + 1) = 2T(n) + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 1$$



Reduzierung auf Summen

- Es kann vorkommen, dass nach dem Studium endlich vieler Anfangswerte keine Vermutung naheliegt, die beweisbar eine Lösung darstellt.
- Hier kann nur noch formal nach einer Lösung gesucht werden.
 - *Abwickeln der Rekursion*
 - Zusammenfassen / Ersetzen bekannter (Teil-)Summen
- Die Methode kann zu einem standardisierten Verfahren zum Auffinden einer geschlossenen Formel für beliebige *lineare* Rekurrenzgleichungen erweitert werden!

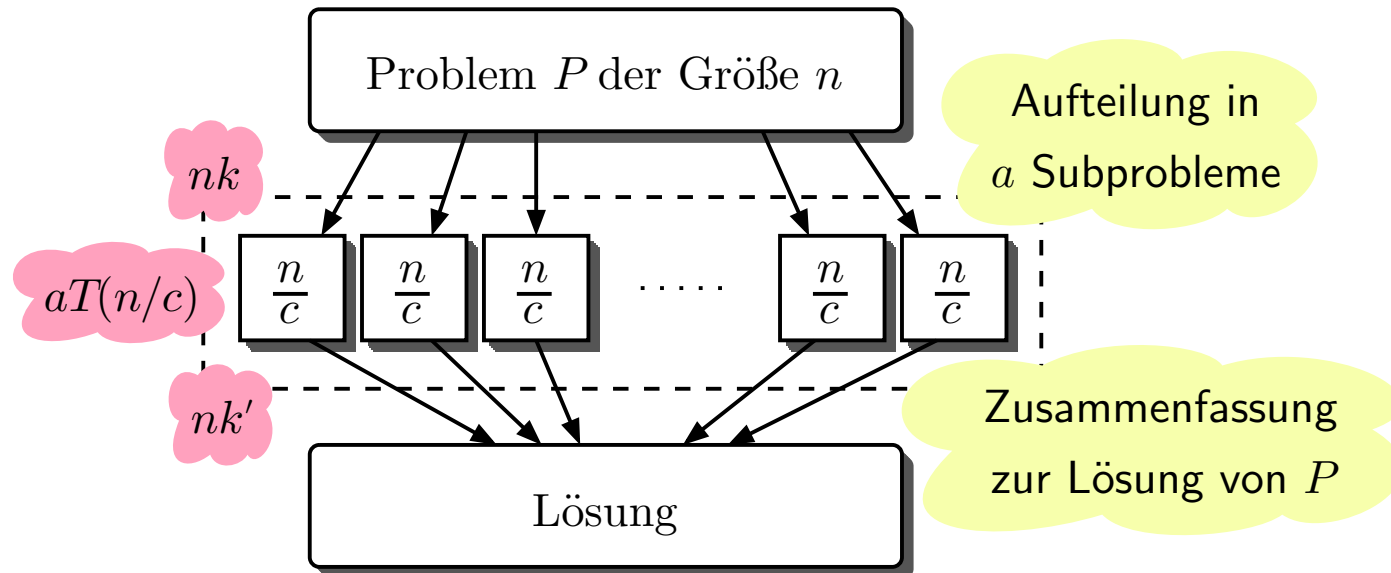


Hanoi — auf Summen reduzieren

$$\begin{aligned}T(n) &= 2T(n-1) + 1 \\&= 2(2T(n-2) + 1) + 1 = 4T(n-2) + 2 + 1 \\&= 4(2T(n-3) + 1) + 2 + 1 \\&= 2^3T(n-3) + 2^2 + 2^1 + 2^0 \\&= 2^kT(n-k) + \sum_{i=0}^{k-1} 2^i \\&= 2^{n-1}T(1) + \sum_{i=0}^{n-2} 2^i \\&= \sum_{i=0}^{n-1} 2^i \quad (\text{geometrische Reihe}) \\&= \frac{2^n - 1}{2 - 1} = 2^n - 1\end{aligned}$$

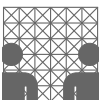


3. Beispiel: divide and conquer



Gesucht: Zeitkomplexität $T(n)$ dieses Algorithmus.

$$T(n) = \begin{cases} d & \text{falls } n = 1 \\ aT\left(\frac{n}{c}\right) + kn + k'n & \text{falls } n > 1 \end{cases}$$

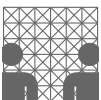


Vereinfachung

- Zur Analyse des *divide-and-conquer*-Algorithmus vereinfachen wir die Rekurrenz zu

$$T(n) = \begin{cases} b & \text{falls } n = 1 \\ aT\left(\frac{n}{c}\right) + bn & \text{falls } n > 1. \end{cases}$$

- k und k' sind für jedes Problem bekannt und konstant!
- $d \leq k + k'$
- Also wählen wir $b := k + k'$

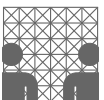


Abwickeln der Rekurrenz

Durch rekursives Einsetzen (Abwickeln) erhalten wir:

$$\begin{aligned}T(n) &= aT\left(\frac{n}{c}\right) + bn = a\left(aT\left(\frac{n}{c^2}\right) + b\frac{n}{c}\right) + bn \\&= a^2T\left(\frac{n}{c^2}\right) + bn\frac{a}{c} + bn \\&= a^2\left(aT\left(\frac{n}{c^3}\right) + b\frac{n}{c^2}\right) + bn\frac{a}{c} + bn \\&= a^3T\left(\frac{n}{c^3}\right) + bn\left(\frac{a}{c}\right)^2 + bn\frac{a}{c} + bn \\&= a^kT\left(\frac{n}{c^k}\right) + bn \cdot \sum_{i=0}^{k-1} \left(\frac{a}{c}\right)^i\end{aligned}$$

Wenn $n = c^k$ ist, so bricht das Verfahren bei $T(n) = a^k b + bn \sum_{i=0}^{k-1} \left(\frac{a}{c}\right)^i$ ab, denn es ist $T(1) = b$.

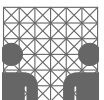


Hin zur geschlossenen Formel

Mit $n = c^k$ folgt $k = \log_c(n)$ und somit

$$\begin{aligned}T(n) &= a^k b + bn \sum_{i=0}^{k-1} \left(\frac{a}{c}\right)^i \\&= a^k b + bn \sum_{i=0}^k \left(\frac{a}{c}\right)^i - bn \left(\frac{a}{c}\right)^k \\&= a^k b \left(1 - \frac{n}{c^k}\right) + bn \sum_{i=0}^k \left(\frac{a}{c}\right)^i \\&= bn \sum_{i=0}^{\log_c(n)} \left(\frac{a}{c}\right)^i\end{aligned}$$

Je nach dem Verhältnis von a zu c ergeben sich unterschiedliche Lösungen!



Fallunterscheidung

Fall 1, $a < c$: $\sum_{i=0}^{\infty} \left(\frac{a}{c}\right)^i$ konvergiert $\Rightarrow T(n)$ proportional zu n

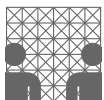
Fall 2, $a = c$: $T(n) = bn(\log_c(n) + 1) \Rightarrow T(n)$ prop. $n \log(n)$

Fall 3, $a > c$: $T(n)$ proportional zu $n^{\log_c(a)}$

Dann folgt

$$\begin{aligned} T(n) &= bn \sum_{i=0}^{\log_c(n)} \left(\frac{a}{c}\right)^i \\ &= bn \frac{\left(\frac{a}{c}\right)^{\log_c(n)+1} - 1}{\frac{a}{c} - 1} \approx bn \left(\frac{a}{c}\right)^{\log_c(n)} \\ &= bn \frac{a^{\log_c(n)}}{n} = ba^{\log_c(n)} = bn^{\log_c(a)} \end{aligned}$$

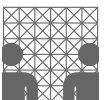
weil $a^{\log_c(n)} = n^{\log_c(a)}$ stets gilt.



Schlussfolgerung

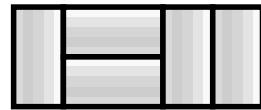
Wichtige Feststellung zu *divide-and-conquer*-Algorithmen:

Die Zeitkomplexität eines *divide-and-conquer*-Algorithmus hängt nur von dem Verhältnis $\frac{a}{c}$ ab – und nicht von der Art des Problems oder vom Lösungsweg –, wenn der Zeitbedarf für die Zerlegung in Teilprobleme und die Zusammenfassung der Teillösungen proportional zur Größe des Problems ist.

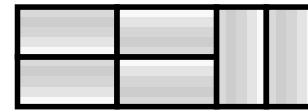


Ein praktisches Problem

Für das Platinenlayout sind Anordnungen von Chips der Maße 1×2 cm auf einer „Bahn“ von 2 cm Höhe und bisher unbestimmter Länge n nötig. Mögliche Anordnungen sind z.B.:

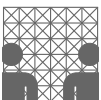


$n=5$



$n=6$

- Sei T_n die Anzahl der verschiedenen Anordnungen auf einer Bahn der Länge n .
- Es gilt $T_n = T_{n-1} + T_{n-2}$ für $n \geq 2$
- Anfangswerte: $T_0 := 1$ und $T_1 := 1$



Fibonacci-Zahlen

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad \text{für } n > 1$$

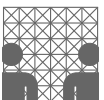
- Wir suchen eine Lösung in der Form $F_n = r^n$ (r ist unbekannte Konstante).
- Existiert eine solche Lösung, dann gilt:

$$r^n = r^{n-1} + r^{n-2} \quad \text{für jedes } n > 1$$

und daraus folgt, dass entweder $r = 0$ oder $r^2 = r + 1$.

- Diese Gleichung hat zwei Lösungen:

$$r_1 = \frac{1 + \sqrt{5}}{2}, \quad r_2 = \frac{1 - \sqrt{5}}{2}$$



Fibonacci-Zahlen (2)

■ Eine allgemeine Lösung der Gleichung hat die Form $\lambda r_1^n + \mu r_2^n$ wobei λ und μ Konstanten sind.

■ Daraus folgt: $\lambda + \mu = F_0 = 0$, $\lambda r_1 + \mu r_2 = F_1 = 1$

■ $\lambda = -\mu = \frac{1}{\sqrt{5}}$ und

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right)$$

■ Wegen $\lim_{n \rightarrow \infty} \left(\frac{1 - \sqrt{5}}{2} \right)^n = 0$ gilt:

$$F_n \approx \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n \quad \text{für } n \rightarrow \infty$$



Der goldene Schnitt

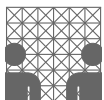
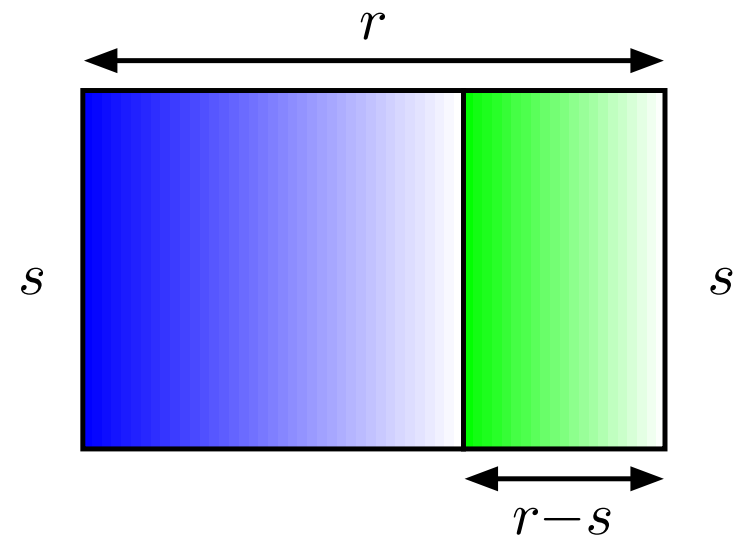
- Das abgetrennte Rechteck soll das gleiche Seitenverhältnis haben, wie d

- Dafür muss $\frac{r}{s} = \frac{s}{r-s}$ gelten.

- Es folgt $x = \frac{r}{s} = \frac{s}{r-s} = \frac{1}{x-1}$
oder $x^2 - x - 1 = 0$.

- Lösungen: $x = \Phi = \frac{1+\sqrt{5}}{2}$ und $x = \hat{\Phi} = \frac{1-\sqrt{5}}{2}$

- Eine Größe r wird nach dem goldenen Schnitt geteilt, wenn der Teil s das geometrische Mittel von r und $r - s$ ist: $s = \sqrt{r(r - s)}$.



Überblick

$$f_n := 3f_{n-1} + 4f_{n-2} + 1$$

Beispiele für Rekurrenzen

Abwickeln

Summen

Raten

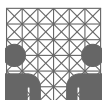
Beweisen

Formelsammlung

Chaos ?!



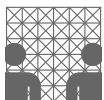
Suche nach einer Systematik



Allgemeines Verfahren

Bestimmung einer **geschlossenen Formel** für durch **lineare Rekurrenzgleichung** definierte Folge $\langle g_n \rangle_{n \geq 0}$:

1. Bilde eine **einzigste Gleichung**, in der g_n durch andere Terme der Folge $\langle g_n \rangle_{n \geq 0}$ ausgedrückt wird.
2. Multipliziere die beiden Seiten der Gleichung mit z^n und summiere für alle n . Die linke Seite wird zu $G(z) = \sum_n^\infty g_n z^n$. Man erhält eine **erzeugende Funktion** für $\langle g_n \rangle_{n \geq 0}$.
3. Löse diese Gleichung nach $G(z)$ auf, um eine **geschlossene Formel** für $G(z)$ zu bekommen.
4. Entwickle $G(z)$ in eine **formale Potenzreihe**, um den Koeffizienten von z^n zu bestimmen.



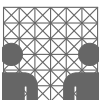
Beispiel

Finde eine geschlossene Formel für die **Fibonacci-Zahlen!**
Fibonacci-Zahlen sind durch folgende Rekurrenzgleichung definiert:

$$f_n = \begin{cases} 0 & \text{falls } n \leq 0 \\ 1 & \text{falls } n = 1 \\ f_{n-1} + f_{n-2} & \text{falls } n > 1 \end{cases}$$

Dies ist offensichtlich **keine** geschlossene Formel!

Was kann nun getan werden, um eine solche zu finden?



Prädikate als Zahlenfunktionen

Definition: [K. Iverson, 1962]

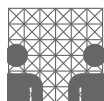
Sei $P(n)$ ein **Prädikat**. Dann ist $[P(n)]$, folgendermaßen definiert:

$$[P(n)] = \begin{cases} 1 & \text{falls } P(n) \text{ wahr ist} \\ 0 & \text{falls } P(n) \text{ nicht wahr ist} \end{cases}$$

Beispiel:

$$f_n := \begin{cases} 2 & \text{falls } n = 0 \\ 3 & \text{falls } n = 1 \\ 2f_{n-1} + f_{n-2} & \text{sonst} \end{cases}$$

$$f_n := 2f_{n-1} + f_{n-2} + 2[n = 0] - [n = 1]$$



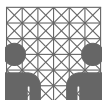
1. Zusammenfassen

- Bilde eine einzige Gleichung, in der g_n durch andere Terme der Folge $\langle g_n \rangle_{n \geq 0}$ ausgedrückt wird.
- Diese Gleichung soll für alle n gültig sein.
- Wir setzen fest: $g_n = 0$ falls $n < 0$.

Beispiel: Fibonacci-Zahlen

Finde eine **einzige Gleichung** für f_n , die für alle n gilt (auch für $n < 0$!).

$$f_n = f_{n-1} + f_{n-2} + [n = 1]$$



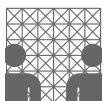
2. Erzeugende Funktion

- Multipliziere Gleichung mit z^n und summiere für n .
- Erzeugende Funktion für $\langle g_n \rangle_{n \geq 0}$: $G(z) = \sum_{n=0}^{\infty} g_n z^n$
- Rechte Seite manipulieren \Rightarrow Ausdruck von $G(z)$.

Beispiel: Fibonacci-Zahlen

Transformiere $f_n = f_{n-1} + f_{n-2} + [n = 1]$:

$$\begin{aligned} F(z) &= \sum_n f_n z^n \\ &= \sum_n (f_{n-1} + f_{n-2} + [n = 1]) z^n \\ &= \sum_n f_{n-1} z^n + \sum_n f_{n-2} z^n + \sum_n [n = 1] z^n \\ &= zF(z) + z^2 F(z) + z \end{aligned}$$



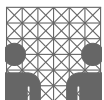
3. Gleichung nach $G(z)$ auflösen

- Löse diese Gleichung nach $G(z)$ auf, um eine geschlossene Formel für $G(z)$ zu bekommen.

Beispiel: Fibonacci-Zahlen

Löse die Gleichung nach $F(z)$ auf:

$$\begin{aligned} F(z) &= zF(z) + z^2F(z) + z \\ \iff F(z) - zF(z) - z^2F(z) &= z \\ \iff F(z) \cdot (1 - z - z^2) &= z \\ \iff F(z) &= \frac{z}{1 - z - z^2} \end{aligned}$$

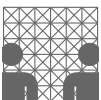


4. Formale Potenzreihe

- Entwickle $G(z)$ in eine formale Potenzreihe, um den Koeffizienten von z^n zu bestimmen.
(Dieser Schritt ist im allgemeinen der schwierigste!)

Beispiel: Fibonacci-Zahlen

- Versuch $\frac{z}{1-z-z^2}$ als Summe zweier Brüche in der Form $z \left(\frac{a}{1-\alpha z} + \frac{b}{1-\beta z} \right)$ zu schreiben.
- Dies bedeutet, den Nenner $(1 - z - z^2)$ in der Form $(1 - \alpha z)(1 - \beta z)$ zu schreiben und danach die Größen a und b zu suchen.

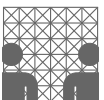


4. Schritt (Forts.)

- Wenn dies geschafft ist, so wissen wir, dass $\frac{a}{1-\alpha z}$ erzeugende Funktion für $a \sum_{n \geq 0} (\alpha z)^n$ ist, also

$$\begin{aligned} F(z) &= z \left(a \sum_{n \geq 0} (\alpha z)^n + b \sum_{n \geq 0} (\beta z)^n \right) \\ &= \sum_{n \geq 0} (a\alpha^{n-1} + b\beta^{n-1}) z^n \end{aligned}$$

- Mit $[z^n]F(z) = f_n = a\alpha^{n-1} + b\beta^{n-1}$ erhalten wir also die n -te Fibonacci-Zahl in **geschlossener Form!**



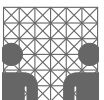
Faktorisierung des Polynoms

- Die **Faktorisierung** von $(1 - z - z^2)$ in die Form $(1 - \alpha z)(1 - \beta z)$ gelingt vielleicht noch direkt.
- Allgemeine Methode für beliebige Polynome:

Definition: Sei $q(z)$ ein Polynom über \mathbb{C} vom Grad d , so heißt $q^R(z) := z^d \cdot q\left(\frac{1}{z}\right)$ das **reflektierte Polynom** von q .

Satz: Sei $q(z) = 1 + q_1z + q_2z^2 + q_3z^3 + \dots + q_dz^d$ ein Polynom über \mathbb{C} vom Grad d und $\alpha_1, \alpha_2, \dots, \alpha_d$ die Nullstellen des reflektierten Polynoms $q^R(z)$, dann gilt:

$$q(z) = (1 - \alpha_1z)(1 - \alpha_2z) \cdot \dots \cdot (1 - \alpha_dz)$$



Fast am Ziel (Fibonacci)

- Bei der noch zu analysierenden Fibonacci-Zerlegung sind also die Nullstellen von $(1 - z - z^2)^R = z^2 - z - 1 = (z - \alpha)(z - \beta)$ zu finden. Es ergeben sich die Nullstellen

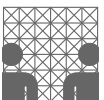
$$\alpha = \frac{1 + \sqrt{5}}{2} \quad \text{und} \quad \beta = \frac{1 - \sqrt{5}}{2}$$

die wir auch schon vom **goldenen Schnitt** kennen.

- Es sind jetzt noch a und b in

$$\frac{1}{(1 - \alpha z)(1 - \beta z)} = \frac{a}{1 - \alpha z} + \frac{b}{1 - \beta z}$$

zu bestimmen.



Noch ein wenig rechnen ...

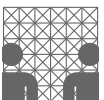
- Wir bringen die Summe auf den Hauptnenner $(1 - \alpha z)(1 - \beta z)$ und es folgt

$$(a+b) - (a\beta + b\alpha)z = 1 = (a+b) - \frac{a(1 - \sqrt{5}) + b(1 + \sqrt{5})}{2}z.$$

- Also $a + b = 1$ sowie $a\beta + b\alpha = 0$ (*warum?!*), woraus $a = \frac{\alpha}{\sqrt{5}} = \frac{1+\sqrt{5}}{2\sqrt{5}}$ und $b = -\frac{\beta}{\sqrt{5}} = \frac{\sqrt{5}-1}{2\sqrt{5}}$.

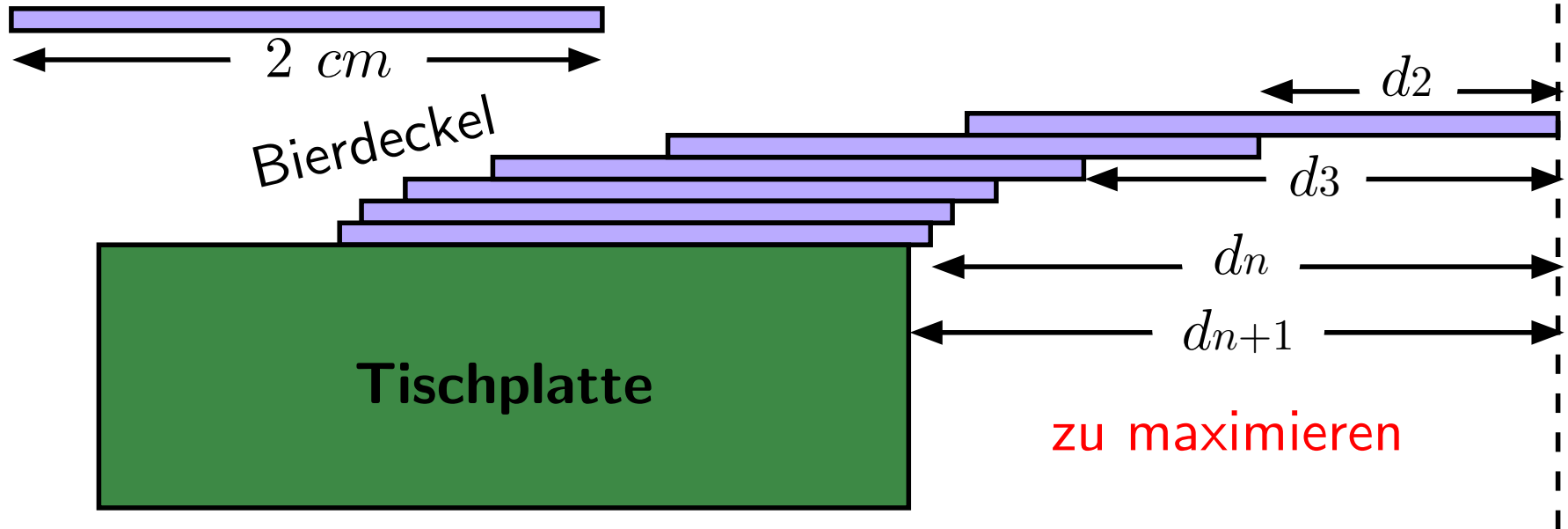
- Mit $f_n = a\alpha^{n-1} + b\beta^{n-1}$ finden wir letztendlich

$$f_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right).$$



Weitere Beispiele ...

... sind im Skript zu finden!

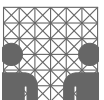


Offensichtlich gilt

■ $d_1 = 0$

■ $d_2 = 1 = \frac{d_1+1}{1}$

■ $d_3 = \frac{(d_1+1)+(d_2+1)}{2}$

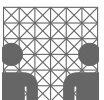


Bierdeckel

... und allgemein

$$\begin{aligned}d_{k+1} &= \frac{1}{k} \cdot \sum_{i=1}^k (d_i + 1) \\ &= \frac{1}{k} \left(k + \sum_{i=1}^k d_i \right) \\ &= 1 + \frac{1}{k} \cdot \sum_{i=1}^k d_i\end{aligned}$$

Somit ist $k \cdot d_{k+1} = k + \sum_{i=1}^k d_i$.



Bierdeckel (Forts.)

- Einsetzen von $k - 1$ statt k in $kd_{k+1} = k + \sum_{i=1}^k d_i$ ergibt:

$$(k - 1) d_k = k - 1 + \sum_{i=1}^{k-1} d_i$$

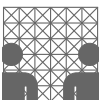
- Durch Subtraktion beider Gleichungen:

$$kd_{k+1} - (k - 1) d_k = 1 + d_k$$

- Umgerechnet die einfache Rekurrenz

$$d_{k+1} = d_k + \frac{1}{k}$$

mit der Lösung $d_{k+1} = \sum_{i=1}^k \frac{1}{i}$.

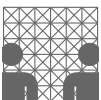


Problem

- **ABER:** Kann man zu jeder Summenformel auch eine geschlossenen Formel finden?
- **Antwort:** Nein, z.B. nicht für das Bierdeckelproblem!
- Bei der Analyse von Algorithmen trifft man oft auf die **harmonischen Zahlen:** (H_n : ist n -te Harmonische Zahl)

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{i=1}^n \frac{1}{i}$$

- H_n kann für jedes n exakt berechnet werden!
- Das reicht jedoch nicht aus, denn es gibt keine geschlossene Formel für H_n . Deshalb wird eine gute **Approximation** für H_n benötigt.



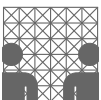
Abschätzung durch Integrale

- **Summenformeln** können oft nach oben und unten durch **Integrale** abgeschätzt werden.

- **Satz** Für $G(n) := \sum_{i=1}^n g(i)$ mit $g(i) \leq g(i+1)$ gilt:

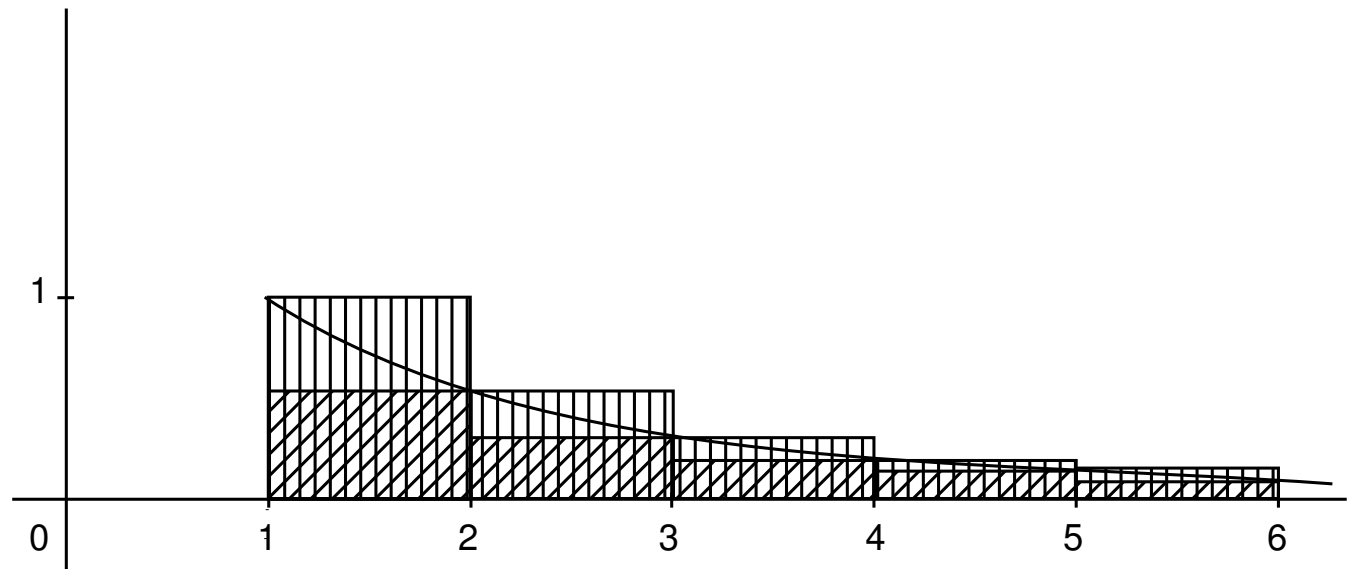
$$\int_0^n g(x) dx \leq G(n) \leq \int_1^{n+1} g(x) dx$$

- Ist eine Funktion $g(x)$ **monoton fallend**, aber gilt stets $g(x) > 0$, so läßt sich das Verfahren der Abschätzung von $G(n) := \sum_{i=1}^n g(i)$ mit Hilfe von Integralen immer noch gut anwenden.

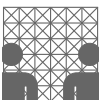


Harmonische Zahlen

- $H_n = \sum_{i=1}^n \frac{1}{i}$, so dass $g(x) = \frac{1}{x}$ ist monoton fallend.
- Wegen $\int \frac{1}{x} dx = \ln(x)$ kann mit dieser Methode eine Abschätzung von H_n vorgenommen werden.
- Für $g(x) = \frac{1}{x}$ ergibt sich:



- Die Rechtecke stellen die Flächen $\frac{1}{x}$ dar.



Approximation von H_n

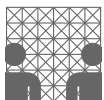
- Es folgt nun $\int_1^n \frac{1}{x} dx < H_n < 1 + \int_1^n \frac{1}{x} dx$ und somit

$$\ln n < H_n < \ln n + 1 \quad \text{falls } n > 1$$

und diese **Approximation** ist schon nicht schlecht.

- Viel besser aber ist die Approximation

$$H_n = \ln n + 0.5772156649 + \frac{1}{2n} - \frac{1}{12n^2} + \underbrace{\Theta(n^{-4})}_{\text{„wächst wie } n^{-4}\text{“}}$$

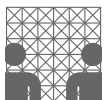


Schriftliches Multiplizieren

- Wieviel Zeit braucht ein Mensch (ein Computer), um zwei n -stellige ganze Zahlen zu multiplizieren, wenn er den gewöhnlichen Schulalgorithmus benutzt?

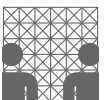
$$\begin{array}{cccccccc} & & & & a_1 & a_2 & \dots & a_n & \times & b_1 & b_2 & \dots & b_n \\ & & & & \bullet & \bullet & \dots & \bullet & & & & & \\ & & & \bullet & \bullet & \dots & \bullet & & & & & & \\ & & \vdots & & \vdots & & & & & & & & \\ & \bullet & \bullet & \bullet & & \dots & \bullet & \bullet & & & & & \end{array} \quad \begin{array}{l} n \text{ Zeilen} \\ \\ 2n \text{ Stellen} \end{array}$$

- Die exakte Antwort ist wenig aussagekräftig, aber es gilt:
Für k -mal größere Zahlen wird k^2 -mal mehr Zeit benötigt.
- Daher braucht man $O(n^2)$ Zeit, um n -stellige ganze Zahlen mit dem Schulalgorithmus zu multiplizieren.



Optimalitätsfrage

- **Gibt es einen schnelleren Algorithmus?**
- x, y : zwei n -stellige ganze Zahlen (n gerade) in Binärdarstellung.
- $x = x_1 2^{\frac{n}{2}} + x_2$ und $y = y_1 2^{\frac{n}{2}} + y_2$, wobei x_1, x_2, y_1, y_2 $\frac{n}{2}$ -stellige Binärdarstellungen ganzer Zahlen sind
- Dann ist $xy = x_1 y_1 2^n + (x_1 y_2 + x_2 y_1) 2^{\frac{n}{2}} + x_2 y_2$
- **vier Multiplikationen** der $\frac{n}{2}$ -stelligen Zahlen, **drei Additionen** und **zwei Shifts** um n und $\frac{n}{2}$ Positionen.
- Mit dem Schulalgorithmus: Komplexität in $O(n^2)$.



Multiplikation anders

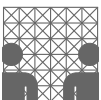
- Andere Methode, die oben benötigten drei Werte x_1y_1 , $x_1y_2 + x_2y_1$ und x_2y_2 zu berechnen:

$$x_1y_1, \quad x_2y_2$$

$$z_1 = (x_1 + x_2)(y_1 + y_2) = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2$$

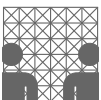
$$z_1 - x_1y_1 - x_2y_2 = x_1y_2 + x_2y_1$$

- Reduktion des Problems der Multiplikation n -stelliger Zahlen auf das Problem von **drei Multiplikationen** $\frac{n}{2}$ -stelliger Zahlen in $O(n)$ Zeit.
- Analyse der Zeitkomplexität von *divide-and-conquer*:
Zeitkomplexität $O(n^{1.58})$ ist!
Das ist besser als die Schulmethode!!!



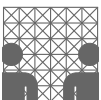
Strukturelle Komplexitätstheorie

- systematisches Studium der Komplexität von Problemen (nicht konkreten Algorithmen!)
 - Probleme als Mengen
 - Problemlösung entspricht Akzeptierung von Wörtern
 - Modell ist die Turing-Maschine (DTM vs. NTM)
 - Zeit- und Platzbedarfe werden abgeschätzt
 - untere und obere Schranken
 - in Abhängigkeit der Eingabegröße
- Eingabelänge zählt nicht zum Platzbedarf!



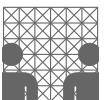
Probleme als Mengen

- Berechenbarkeit einer beliebigen Funktion ist manchmal umständlich zu zeigen!
- Idee: Akzeptierung verwenden, um die Komplexität von Problemen zu untersuchen.
 - intern wird etwas berechnet
 - das Ergebnis muss nicht (standardisiert) ausgegeben werden
 - auch NTM einsetzbar
 - damit ist ein Vergleich zwischen deterministischen und nichtdeterministischen Problemlösungen direkt möglich
- *Wie codiere ich ein Problem als Menge?*



Probleme als Mengen (2)

- Probleme bestehen aus:
 - Fragestellung
 - Eingaben/Voraussetzungen
 - Lösung
- Problemstellung & Lösung(svorschlag) stellen die **Probleminstanz** dar.
- Bestandteile werden als Wörter über einem endlichen Alphabet codiert.
- Genau die Probleminstanzen, die eine Lösung darstellen sollen akzeptiert werden.

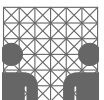


Hamilton-Kreis

Gegeben: Ein ungerichteter Graph $G = (V, E)$

Gesucht: Gibt es einen geschlossenen Kreis in G , bei dem jeder Knoten genau einmal durchlaufen wird, d.h. existiert eine Folge von Knoten v_1, v_2, \dots, v_n mit $n = |V|$, $V = \{v_1, v_2, \dots, v_n\}$ und $\{v_i, v_{i+1}\} \in E$ und $\{v_n, v_1\} \in E$ für alle $i \in \mathbb{N}$ mit $1 \leq i < n$?

Antwort: JA / NEIN

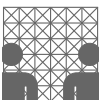


Hamilton-Kreis als Menge



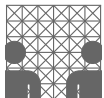
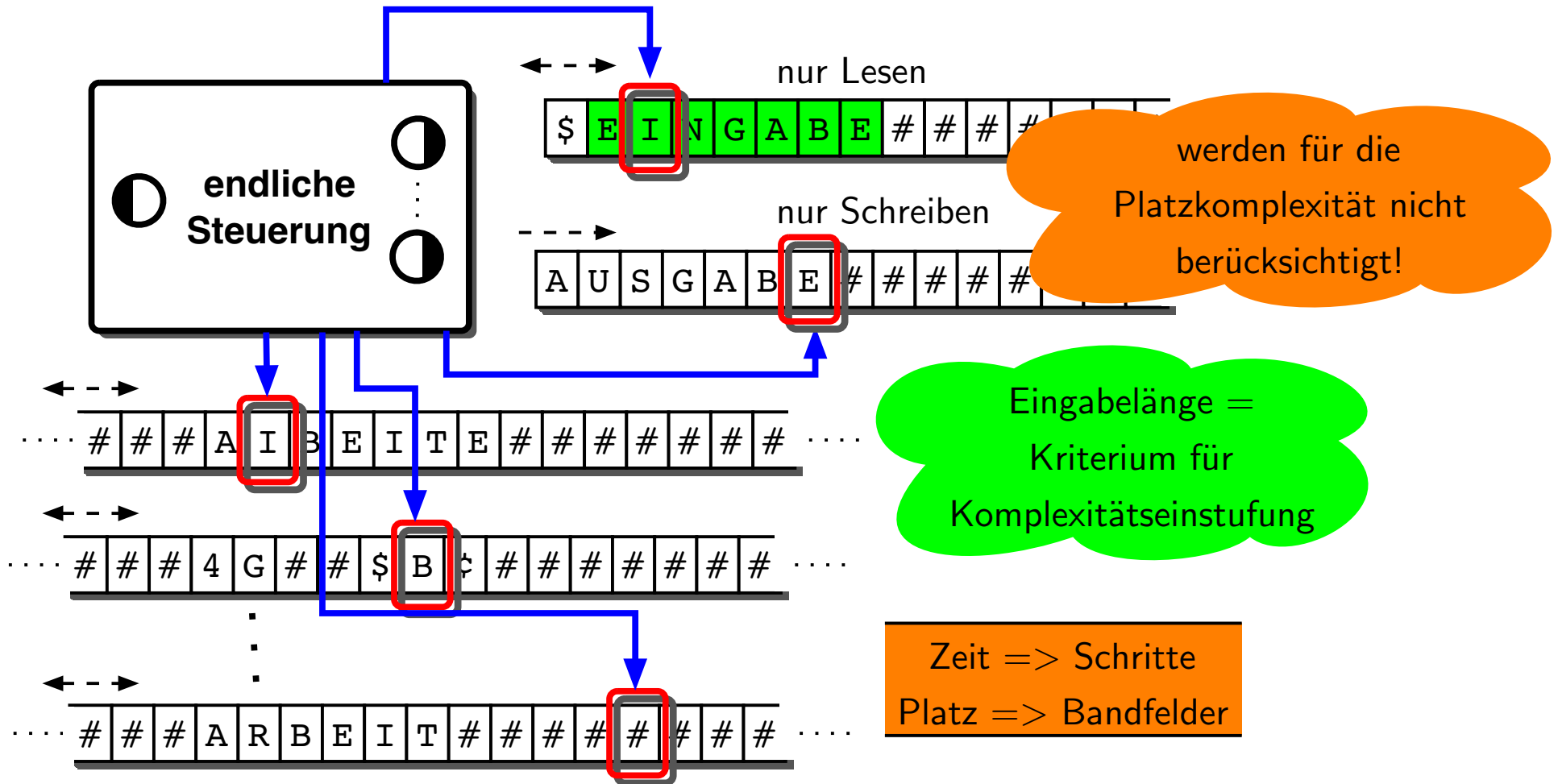
$$G = (\{1, 2, 3, 4\}, \{(1, 2), (2, 3), (3, 4)\})$$

- Mögliche Darstellung: $1, 2, 3, 4(1, 2)(2, 3)(3, 4)$
 - Problem: kein **endliches** Alphabet
- Alternative: $1, 11, 111, 1111(1, 11)(11, 111)(111, 1111)$
- Darstellung mit den Zeilen der **Adjazenzmatrix**:
 $0100\#1010\#0101\#0010$
- Länge ist stets $|V|^2 + |V| - 1$ Zeichen. (obiges Beispiel hat die Länge 19.)



k -Band-offline-Turing-Maschine

Wir verwenden künftig generell k -Band-offline-TM



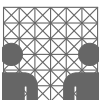
Begriffe: Zeit und Platz

Definition: Für eine konkrete Rechnung *benötigt* eine NTM soviel

- **Zeit**, wie Konfigurationenswechsel auftreten.
- **Platz**, wie maximal Felder auf den Arbeitsbändern besucht werden.

Definition Eine NTM A verarbeitet ein Wort w mit der

- **Zeitbeschränkung** t genau dann, wenn die *kürzeste* Rechnung für w höchstens $\lceil t \rceil$ Zeit benötigt.
- **Platzbeschränkung** $s \in \mathbb{R}$, wenn die Rechnung mit dem geringsten Platzverbrauch für w höchstens $\lceil s \rceil$ Platz benötigt.

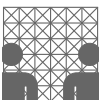


Komplexitätsklassen

- Zusammenfassen ähnlicher Eingaben
 - über die Art des Problems (Semantik) ist in der Regel nur sehr schwer eine Aussage zu machen
 - Kategorisierung nach **Länge des Eingabewortes**.
-

Definition: Seien $s : \mathbb{R} \rightarrow \mathbb{R}$, $t : \mathbb{R} \rightarrow \mathbb{R}$. Eine NTM A ist

- **t -zeitbeschränkt** gdw. sie für **jedes akzeptierte** Eingabewort der Länge n mit der Zeitbeschränkung $t(n)$ arbeitet.
- **s -platzbeschränkt** genau dann, wenn sie für *jedes akzeptierte* Eingabewort der Länge n mit der Platzbeschränkung $s(n)$ arbeitet.



Beispiele von Komplexitätsklassen

Definition Für $s, t : \mathbb{N} \rightarrow \mathbb{R}$ mit $t(n) \geq n + 1$ und $s(n) \geq 1$, bezeichne:

$$\mathcal{D}Time(t) := \{L \mid L = L(A) \text{ für eine } t\text{-zeitbeschr. DTM } A\}$$

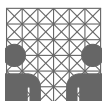
$$\mathcal{N}Time(t) := \{L \mid L = L(A) \text{ für eine } t\text{-zeitbeschr. NTM } A\}$$

$$\mathcal{D}Space(s) := \{L \mid L = L(A) \text{ für eine } s\text{-platzbeschr. DTM } A\}$$

$$\mathcal{N}Space(s) := \{L \mid L = L(A) \text{ für eine } s\text{-platzbeschr. NTM } A\}$$

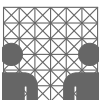
$$\mathcal{P} := \{L \mid \text{Es gibt ein Polynom } p : \mathbb{N} \rightarrow \mathbb{R} \text{ und eine } p\text{-zeitbeschränkte DTM } A \text{ mit } L = L(A)\}$$

$$\mathcal{NP} := \{L \mid \text{Es gibt ein Polynom } p : \mathbb{N} \rightarrow \mathbb{R} \text{ und eine } p\text{-zeitbeschränkte NTM } A \text{ mit } L = L(A)\}$$



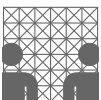
Einfache Zusammenhänge

- Eine TM, die $t(n)$ Zeit (d.h. Schritte) zur Verfügung hat, kann nicht mehr als $t(n)$ Bandzellen besuchen.
 - Umgekehrt gilt dies nicht!
- Platz kann **wiederverwendet** werden, Zeit nicht!
- *Kann man trotzdem eine maximale Rechenzeit in Abhängigkeit vom verbrauchten Platz angeben?*
 - Ja! **Idee:** Sobald eine Konfiguration ein zweites Mal besucht wird, befinden wir uns in einer Endlosschleife!
 - Bleibt zu berechnen, **wieviele verschiedene Konfigurationen** bei einer gegebenen Platzbeschränkung möglich sind.



Anzahl der Konfigurationen

- Wovon hängt die Anzahl der möglichen unterschiedlichen Konfigurationen ab?
 - maximal verbrauchter Platz
 - **Kopfpositionen** auf den Arbeitsbändern
 - **Länge** möglicher Bandinschriften
 - Anzahl der Bandsymbole
 - **Permutationen** von Bandsymbolen
 - Länge der Eingabe
 - **Kopfpositionen** auf dem Eingabeband
 - Anzahl der Zustände in der endlichen Steuerung
 - aktueller **Zustand** einer Konfiguration



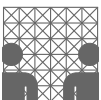
Randbedingungen für Schranken

- Wenn eine TM mit der Platzbeschränkung $s(n)$ arbeitet, dann ist $s(n) \geq 1$ für jedes $n \in \mathbb{N}$, denn die TM muss sich wenigstens ein Feld auf dem Arbeitsband ansehen!
 - **TM arbeitet mit Platzbeschränkung** $s(n)$, wenn sie $\max\{1, \lceil s(n) \rceil\}$ -platzbeschränkt ist.
- Es ist ebenfalls nützlich anzunehmen, dass eine TM ihre Eingabe stets vollständig liest und ein weiteres Feld vorrückt, um deren Ende festzustellen.
 - **TM arbeitet mit Zeitbeschränkung** $t(n)$, wenn sie $\max\{n + 1, t(n)\}$ -zeitbeschränkt ist.



Beispiel $\{w\$w^{rev} \mid w \in \{a, b\}^*\}$

- $\{w\$w^{rev} \mid w \in \{a, b\}^*\}$ ist kontextfrei.
- Mit dem Verfahren von Cocke/Younger/Kasami kann sie somit in Polynomzeit analysiert werden.
- Wieviel Platz wird gebraucht? $s(n) = n$ genügt.
 - **Idee:** Akzeptieren nach dem **Kellerprinzip**, für Wörter, die in der Sprache enthalten sind, reicht dann sogar $\lceil \frac{n+1}{2} \rceil$.
- Es geht aber noch besser: $\log(n)$ Platzbedarf genügt!
 - Codieren der Anzahl bereits bearbeiteter Symbole von w bzw. w^{rev} in $w\$w^{rev}$ als Binärzahl: Ein Zähler gibt das nächste zu bearbeitende Zeichen an.



Bandkompression

Theorem:

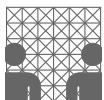
Zu jeder $s(n)$ -platzbeschränkten TM und jeder reellen Zahl $c \in \mathbb{R}$ mit $c > 0$ gibt es eine äquivalente TM die $c \cdot s(n)$ -platzbeschränkt ist.

Beweis:

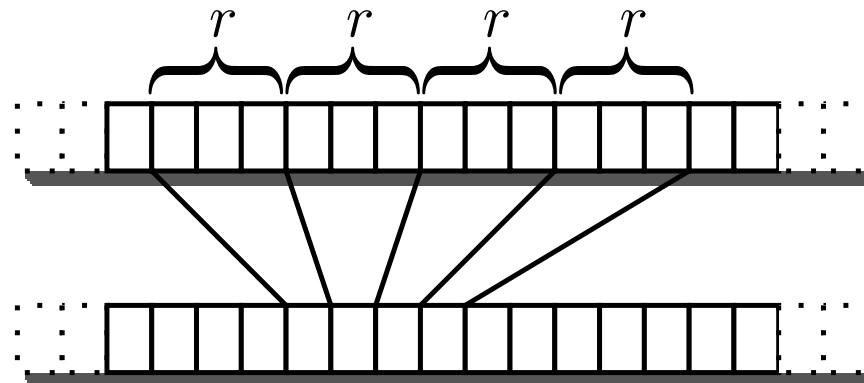
1. Fall Für $c \geq 1$ ist nichts zu beweisen.

2. Fall Für $c < 1$:

- Konstruktion analog zum $|w|$ -platzbeschränkten LBA B aus einem beliebigen LBA A für eine TM durchführen!
- Wenn A $s(n)$ -platzbeschränkt ist, so ist B dann nur noch $c \cdot s(n)$ -platzbeschränkt.



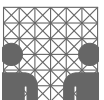
Bandkompression (2)



- Zusammenfassen von jeweils r Symbolen
- Erweiterung des Bandalphabets
- r berechnet sich aus $r = \lceil \frac{s(n)}{c} \rceil$

Also: $x\text{Band}(f) = x\text{Band}(cf)$ für $x \in \{\mathcal{D}, \mathcal{N}\}$.

... zum Beispiel: $f(n) := 3n^2 - 70n + 5$ und $g(n) := n^2$
als Platzschränken nicht unterscheidbar!



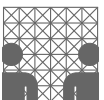
linear speed-up

Theorem: (lineare Beschleunigung)

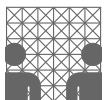
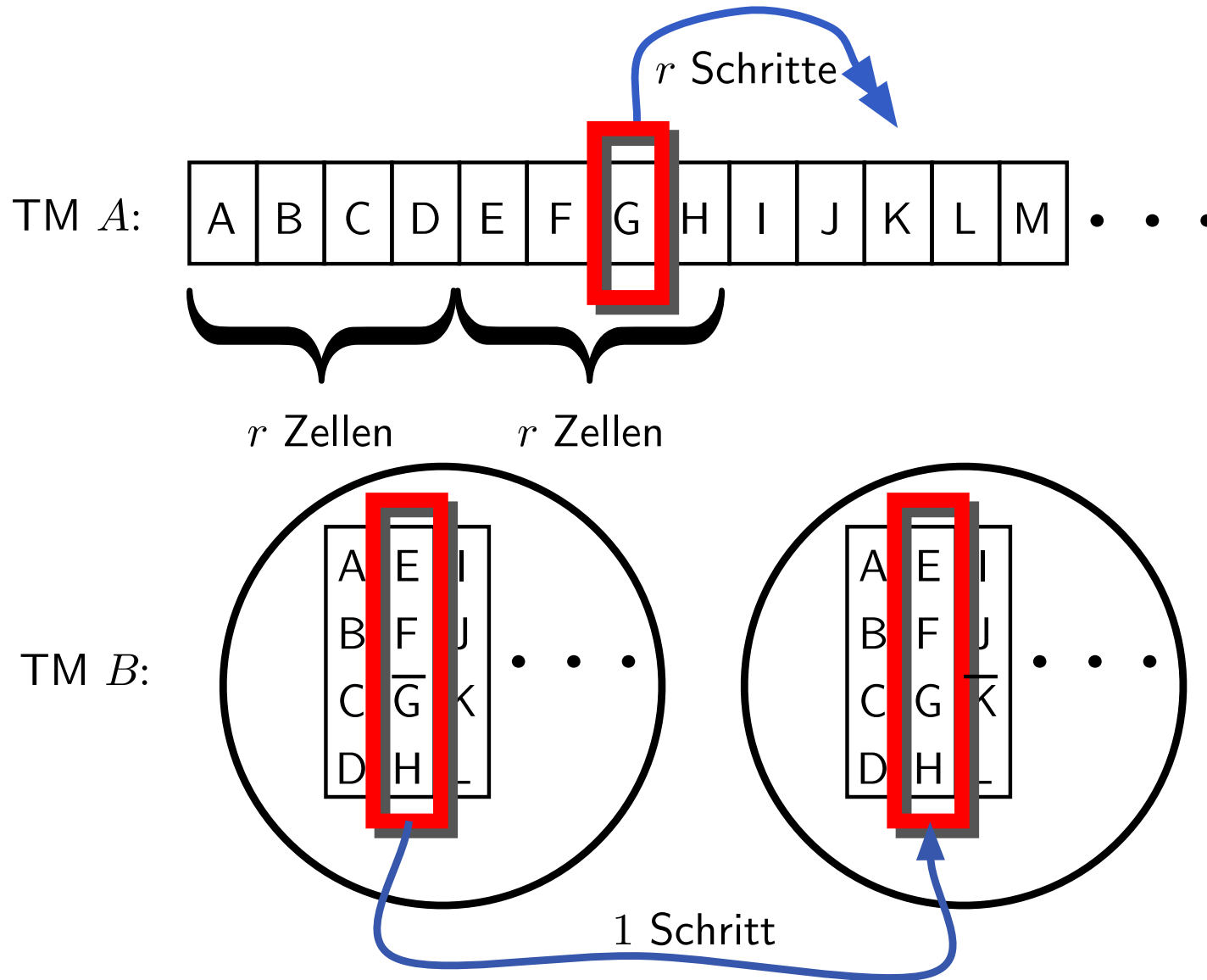
Zu jeder $t(n)$ -zeitbeschränkten TM mit $\inf_{n \rightarrow \infty} \left(\frac{t(n)}{n} \right) = \infty$ und jedem $c \in \mathbb{R}$ mit $c > 0$ gibt es eine äquivalente $c \cdot t(n)$ -zeitbeschränkte TM.

Beweisidee:

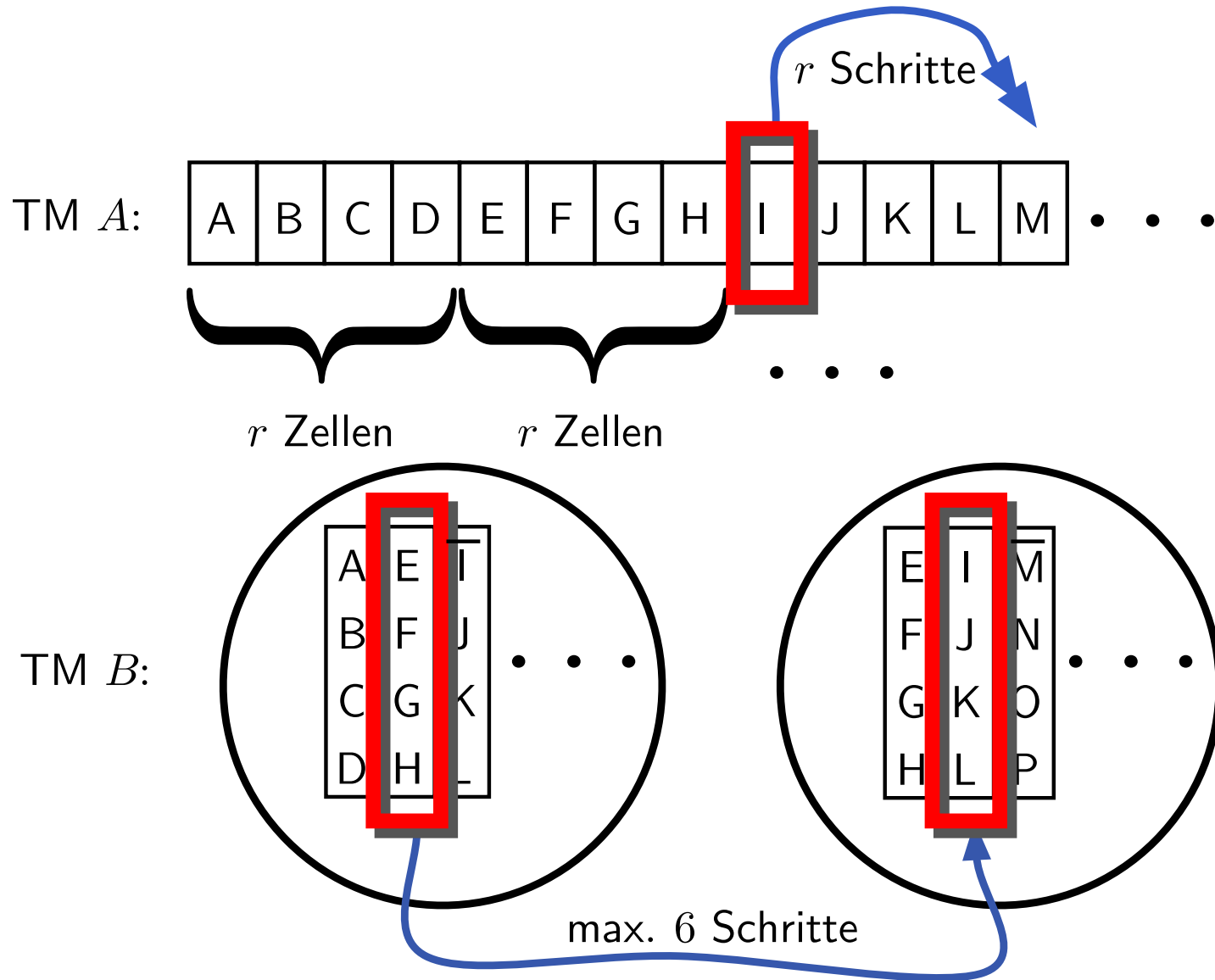
- Blockbildung (Länge r , wie bei der Bandkompression).
- Speichern von jeweils 3 Blöcken im Zustand der endlichen Kontrolle.
- Ein Schritt simuliert r Schritte der ursprünglichen TM.



Beweis: Speed-Up-Theorem

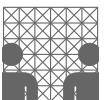


Beweis: Speed-Up-Theorem



Beweis: Speed-Up-Theorem (2)

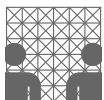
- Verlassen des 3-Block-Bereiches kostet A mind. r Schritte, B nur 8 Schritte. \Rightarrow Für $t(n)$ Schritte von A benötigt B max. $\lceil \frac{8t(n)}{r} \rceil$ Schritte.
- n Schritte, zum Lesen und Codieren der Eingabe.
- $\lceil \frac{n}{r} \rceil$ Schritte, Repositionieren des LSK von B .
- Wegen $\lceil x \rceil \leq x + 1$, sind max. $n + 1 + \frac{n}{r} + 1 + 8 \cdot \left(\frac{t(n)}{r}\right)$ Schritte von B nötig.
- $\inf_{n \rightarrow \infty} \left(\frac{t(n)}{n}\right) = \infty \Rightarrow \forall d : \exists n_d : \forall n \geq n_d : \left(\frac{t(n)}{n}\right) \geq d$
(d.h. auch $n \leq \left(\frac{t(n)}{d}\right)$).
- Für $n \geq 2$ (und somit $n + 2 \leq 2n$) und $n \geq n_d$: Schrittzahl von B max. $t(n) \left(\frac{2}{d} + \frac{1}{dr} + \frac{8}{r}\right)$.



Beweis: Speed-Up-Theorem (3)

- Wählen wir $r := \lceil \frac{16}{c} \rceil$ und $d := \lceil \frac{4}{c} \rceil + \frac{1}{8}$ dann gilt $r \cdot c \geq 16$ und $d \geq \frac{32+c}{8c}$.
- Einsetzen in die Formel für die max. Laufzeit von B :

$$\begin{aligned} \frac{2}{d} + \frac{1}{dr} + \frac{8}{r} &= \frac{2}{d} + \frac{c}{drc} + \frac{8c}{rc} \\ &\leq \frac{2}{d} + \frac{c}{16d} + \frac{8c}{16} \\ &= \frac{32+c}{16d} + \frac{c}{2} \\ &= \frac{(32+c)c}{16cd} + \frac{c}{2} \\ &= \frac{(32+c)c}{8c \cdot 2d} + \frac{c}{2} \leq \frac{cd}{2d} + \frac{c}{2} = c \end{aligned}$$



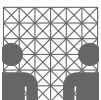
Asymptotische Analyse

- **Konstante Faktoren** und **additive Glieder** spielen bei Beschränkungsfunktionen keine entscheidende Rolle.
- Zur Klassifizierung von Funktionen verwendet man deshalb die bekannten Landau-Schreibweisen: die O -, Ω - und θ -Notation.
- *Wichtig:* Bei der asymptotischen Analyse gilt:

THINK BIG

das heißt, es kommt nur darauf an, wie sich die Funktionen für *große Argumente* verhalten.

- Eine **endliche Anzahl** von „Ausreißern“ ist immer erlaubt!



Zwei extreme Beispiele (1)

■ Es gilt $\log(n) \in o(n^{0.0001})$.

■ Das heißt, $\lim_{n \rightarrow \infty} \frac{\log(n)}{n^{0.0001}} = 0$

■ kurz: $n^{0.0001}$ wächst viel schneller als $\log n$.

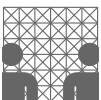
■ Für $n = 10^{100}$:

$$\log 10^{100} = 100 > (10^{100})^{0.0001} = 10^{0.01} = 1,023$$

■ Für $n = 10^{10^{100}}$:

$$\log 10^{10^{100}} = 10^{100} < \left(10^{10^{100}}\right)^{0.0001} = 10^{10^{96}}$$

■ Also: $\forall n > 10^{10^{100}} : \log(n) < n^{0.0001}$.

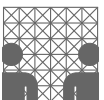


Zwei extreme Beispiele (2)

- Sei $\epsilon = 1/10^{10^{100}}$.
- Es gilt $\log(n) \in o(n^\epsilon)$.
- Seien k und n so gewählt, dass $\epsilon > 10^{-k}$ und $n = 10^{10^{2k}}$.
- Dann gilt:

$$\log n = 10^{2k},$$
$$n^\epsilon > \left(10^{10^{2k}}\right)^{10^{-k}} = 10^{10^k}$$

- Also: $\forall n > 10^{10^{2k}} : \log(n) < n^\epsilon$.



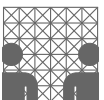
Einfache Folgerungen

- Es gilt wegen des *speed-up*-Satzes:

$$\mathcal{P} = \bigcup_{i \geq 1} \mathcal{D}Time(n^i) \quad \text{und} \quad \mathcal{NP} = \bigcup_{i \geq 1} \mathcal{N}Time(n^i)$$

- Ähnlich definiert man polynomiale Platz-Klassen:

$$\mathcal{P}Space := \bigcup_{i \geq 1} \mathcal{D}Space(n^i)$$
$$\mathcal{NP}Space := \bigcup_{i \geq 1} \mathcal{N}Space(n^i)$$



Einfache Inklusionen

Korollar:

Die sich direkt aus den Definitionen ergebenden trivialen Beziehungen zwischen diesen Komplexitätsklassen sind offensichtlich die folgenden:

$$\mathcal{D}Space(f) \subseteq \mathcal{N}Space(f)$$

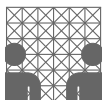
$$\mathcal{D}Time(f) \subseteq \mathcal{N}Time(f)$$

$$\mathcal{D}Time(f) \subseteq \mathcal{D}Space(f)$$

$$\mathcal{N}Time(f) \subseteq \mathcal{N}Space(f)$$

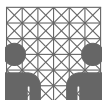
$$\mathcal{P} \subseteq \mathcal{NP}$$

$$\mathcal{P}Space \subseteq \mathcal{NP}Space$$



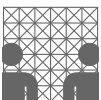
\mathcal{P} und die Praxis

- Von Problemen aus der Komplexitätsklasse \mathcal{P} sagt man, dass sie **effizient lösbar** seien.
- Probleme in \mathcal{P} sind anscheinend wirklich in der Praxis noch mit vertretbarem Aufwand lösbar.
- Nehmen wir also an, dass tatsächlich alle Probleme in \mathcal{P} in der Praxis vernünftig gelöst werden können.
 - Was ist dann mit \mathcal{NP} ?
 - Sind Probleme aus \mathcal{NP} nicht mehr praktisch verwendbar/lösbar?
 - Handelt es sich nur um praktisch nicht relevante Probleme?



Die Klasse \mathcal{NP}

- Jedes Problem $L \in \mathcal{NP}$, ist mit exponentiellem Zeitaufwand (in $\mathcal{DTime}(2^{p(n)})$ für ein Polynom p) deterministisch lösbar.
 - Bessere Transformationen sind i.a. nicht bekannt.
- Viele der *praktisch wichtigen* Fragestellungen haben Lösungen mit Algorithmen, die **nur dann in Polynomzeit arbeiten, wenn sie nicht-deterministisch sind**, jedenfalls kennt man zur Zeit noch nicht Besseres!
- Eine **Implementierung** erfordert ein **deterministisches Verfahren**.
- Unterscheiden sich \mathcal{P} und \mathcal{NP} überhaupt?!



Einige Probleme

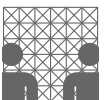
- Ähnliche Probleme liegen trotzdem oft in (vielleicht wirklich) unterschiedlichen Komplexitätsklassen!

Gegeben: Matrix $A \in \mathbb{Z}^{m \times n}$ und Vektor $b \in \mathbb{Z}^m$	
Frage: Gibt es $x \in \mathbb{Z}^n$ mit $Ax = b$	Frage: Gibt es $x \in \mathbb{N}^n$ mit $Ax = b$
\mathcal{P}	\mathcal{NP}



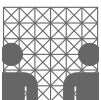
Einige Probleme (2.1)

Gegeben: Ungerichteter, kantenbewerteter Graph $G = (V, E)$, eine Gewichtsfunktion $g : E \rightarrow \mathbb{N}$, zwei Knoten $s, t \in V$ und eine Schranke $k \in \mathbb{N}$	
Frage: Gibt es einen einfachen Pfad p von s nach t mit $g(p) \leq k$? („Kürzester Weg zwischen zwei Knoten“)	Frage: Gibt es einen einfachen Pfad p von s nach t mit $g(p) \geq k$? („Längster Weg zwischen zwei Knoten“)
\mathcal{P}	\mathcal{NP}



Einige Probleme (2.2)

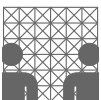
- $L_w := \{\langle G, s, t, k \rangle \mid G \text{ besitzt einen Pfad } p \text{ von } s \text{ nach } t \text{ mit } g(p) \geq k\}$ ist die dem Problem „Längster Weg zwischen zwei Knoten“ zugeordnete Sprache.
- **Theorem:** Das Problem zu L_w ist nichtdeterministisch in Polynomzeit zu lösen, d.h. $K_w \in \mathcal{NP}$.
Idee: Raten eines geeigneten Pfades.
- $K_w := \{\langle G, s, t, k \rangle \mid G \text{ besitzt einen Pfad } p \text{ von } s \text{ nach } t \text{ mit } g(p) \leq k\}$ ist die Sprache, die zu dem Problem „Kürzester Weg zwischen zwei Knoten“ gehört.
- **Theorem:** Das Problem zu K_w ist in \mathcal{P} .
Idee: Systematische Suche der minimalen Pfadlängen.



Einige Probleme (3)

Gegeben: Ungerichteter Graph $G = (V, E)$	
Frage: Gibt es einen geschlossenen Kreis, in dem jede Kante genau einmal auftritt? („Euler-Kreis“)	Frage: Gibt es einen geschlossenen Kreis, in dem jeder Knoten genau einmal auftritt? („Hamilton-Kreis“)
\mathcal{P}	\mathcal{NP}

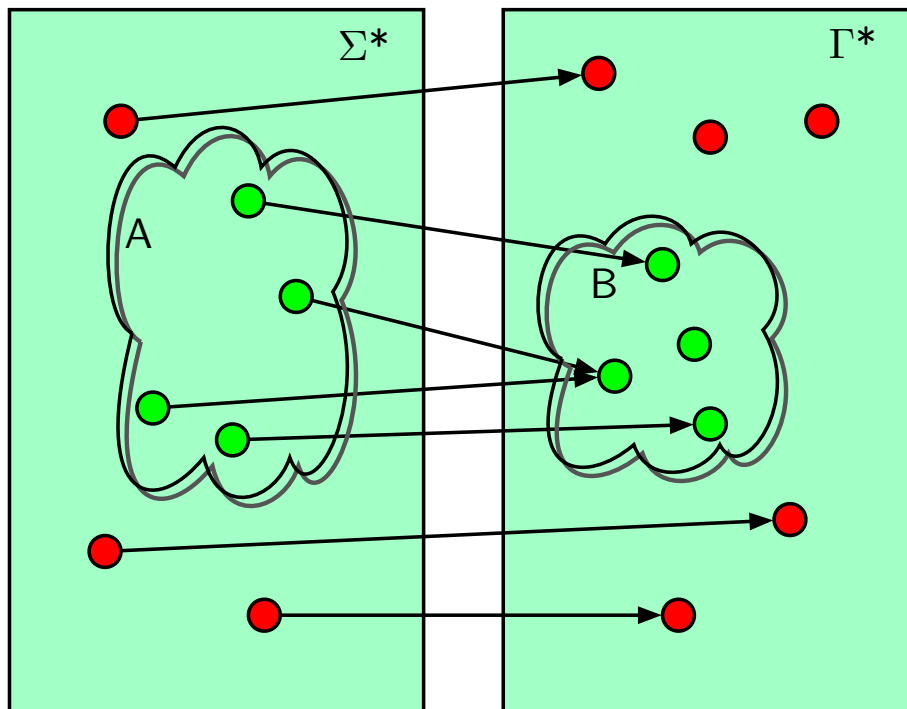
- (Leider) sind häufig Probleme aus \mathcal{NP} die (für die Praxis) interessantesten!
- Manchmal kann man das aber auch ausnutzen. Z.B. für die Kryptographie.



Reduktion

- Seien $A \subset \Sigma^*$ und $B \subset \Gamma^*$.
- Man sagt „ A ist **reduzierbar** auf B “ ($A \leq B$) gdw.

$$\exists f : \Sigma^* \rightarrow \Gamma^* : \forall x \in \Sigma^* : x \in A \iff f(x) \in B$$



von speziellem Interesse:

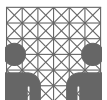
Polynomialzeitreduktion

(\leq_{pol}),

logarithmische-Platz-

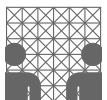
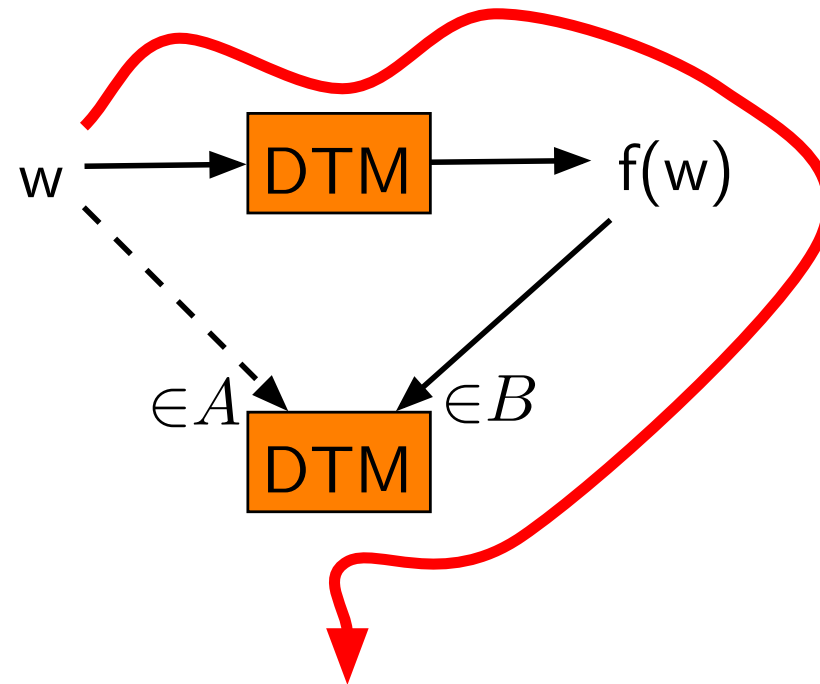
Reduktion

(\leq_{log}).



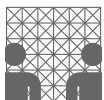
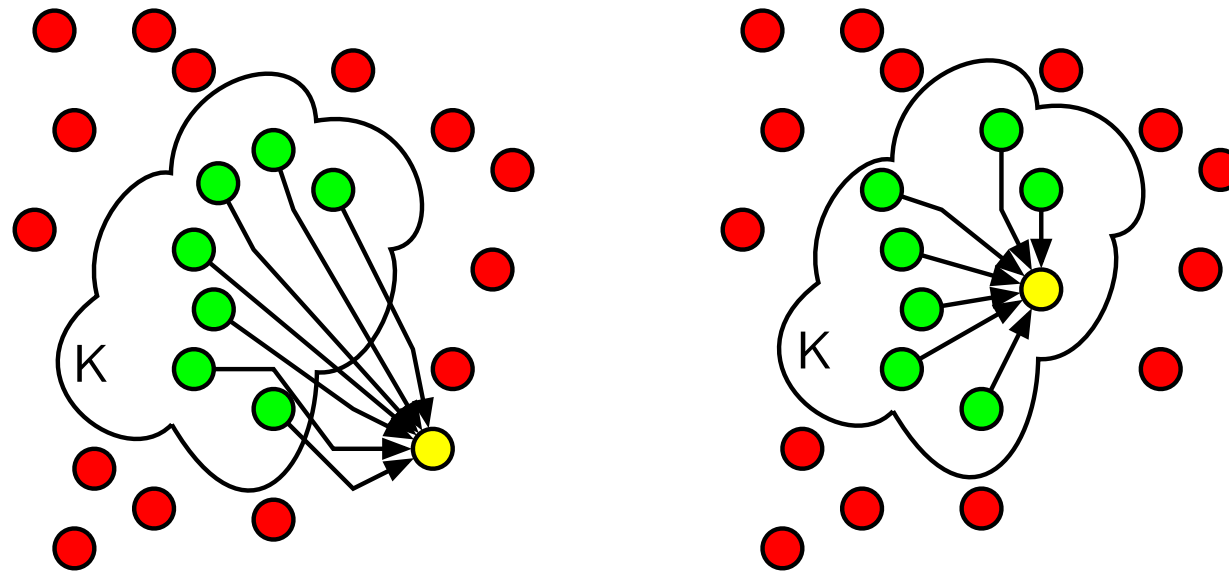
Reduktion (2)

- Zurückführen der Entscheidbarkeit von A auf die Entscheidbarkeit von B :
- Voraussetzung $A \leq B$.



\mathcal{K} -Vollständigkeit

- Eine Menge $A \subseteq \Sigma^*$ heißt **hart** (oder besser: **schwer**) für eine Klasse \mathcal{K} gdw. $\forall B \in \mathcal{K} : B \leq A$
- Eine Menge $A \subseteq \Sigma^*$ heißt **vollständig** für eine Klasse \mathcal{K} gdw. $A \in \mathcal{K} \wedge \forall B \in \mathcal{K} : B \leq A$

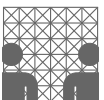


\mathcal{NP} -Vollständigkeit

- besonders interessanter Spezialfall der Vollständigkeit:
- Eine Menge $A \subseteq \Sigma^*$ heißt \mathcal{NP} -**vollständig** gdw.

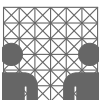
$$A \in \mathcal{NP} \wedge \forall B \in \mathcal{NP} : B \leq_{\text{pol}} A$$

- Ein \mathcal{NP} -vollständiges Problem ist somit eines der **schwersten** bzw. **umfassendsten** Probleme innerhalb der Klasse \mathcal{NP} .
- Wichtige Eigenschaft: Transitivität von \leq_{pol} .
- Ist A ein \mathcal{NP} -vollständiges Problem und gilt $A \leq_{\text{pol}} B$, so ist auch B \mathcal{NP} -vollständig.



Ausblick

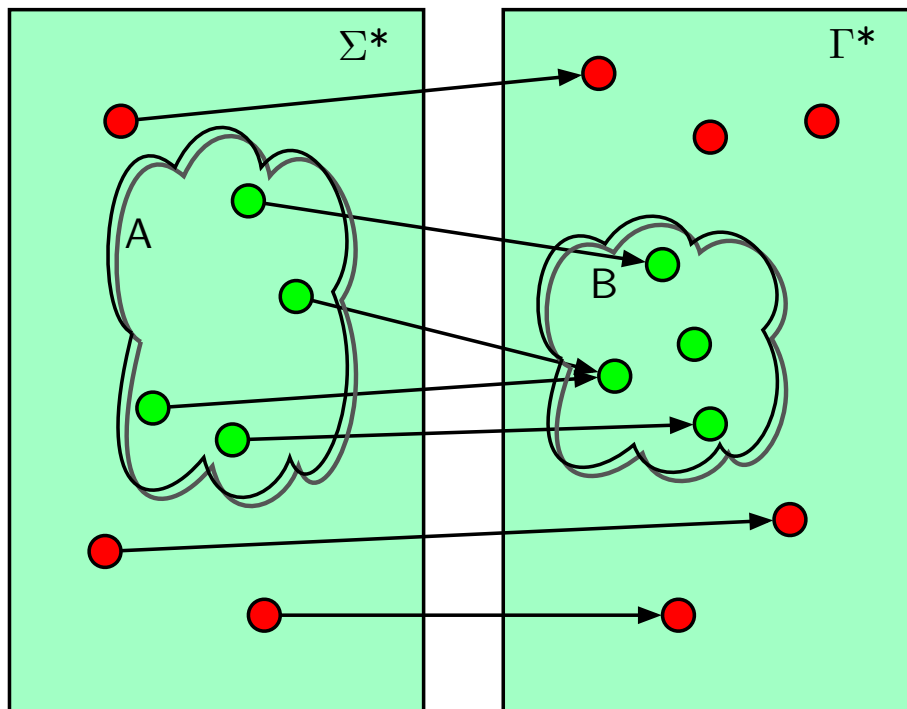
- Existenz eines \mathcal{NP} -vollständigen Problems
 - Konstruktion am Beispiel des Dominoproblems (Parkettierung)
- Einfache Folgerungen
- das $\mathcal{P}=\mathcal{NP}$ -Problem



Reduktion

- Seien $A \subset \Sigma^*$ und $B \subset \Gamma^*$.
- Man sagt „ A ist **reduzierbar** auf B “ ($A \leq B$) gdw.

$$\exists f : \Sigma^* \rightarrow \Gamma^* : \forall x \in \Sigma^* : x \in A \iff f(x) \in B$$



von speziellem Interesse:

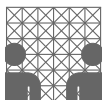
Polynomialzeitreduktion

(\leq_{pol}),

logarithmische-Platz-

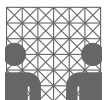
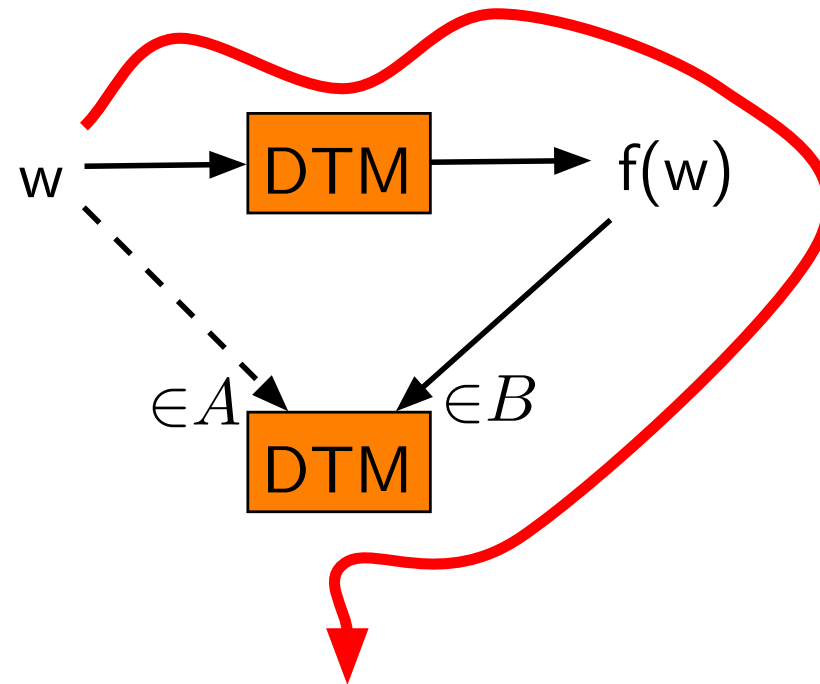
Reduktion

(\leq_{log}).



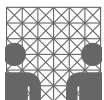
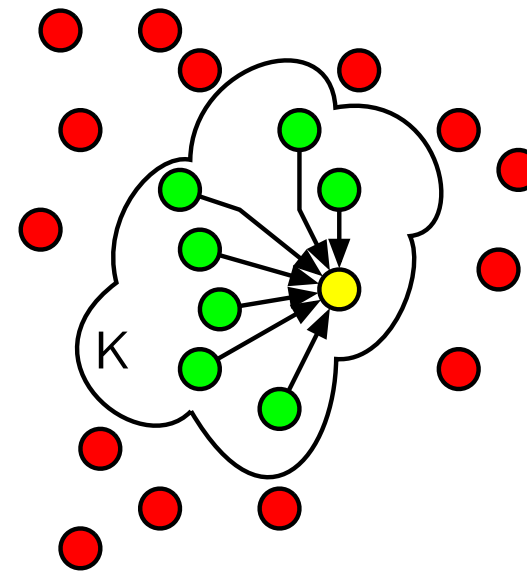
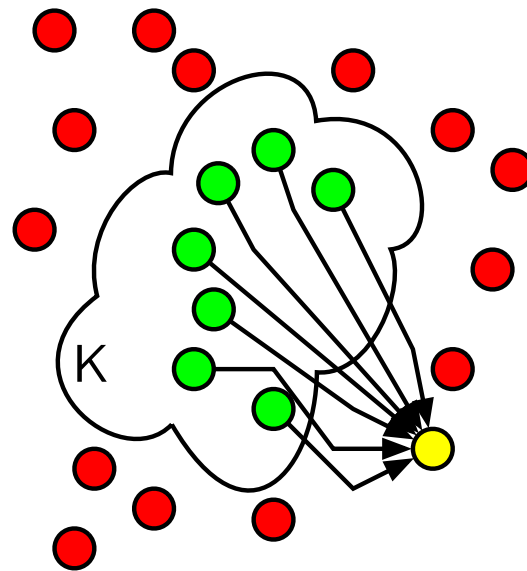
Reduktion (2)

- Zurückführen der Entscheidbarkeit von A auf die Entscheidbarkeit von B :
- Voraussetzung $A \leq B$.



\mathcal{K} -Vollständigkeit

- Eine Menge $A \subseteq \Sigma^*$ heißt **hart** (oder besser: **schwer**) für eine Klasse \mathcal{K} gdw. $\forall b \in \mathcal{K} : B \leq A$
- Eine Menge $A \subseteq \Sigma^*$ heißt **vollständig** für eine Klasse \mathcal{K} gdw. $A \in \mathcal{K} \wedge \forall b \in \mathcal{K}. B \leq A$

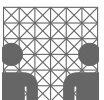


\mathcal{NP} -Vollständigkeit

- besonders interessanter Spezialfall der Vollständigkeit:
- Eine Menge $A \subseteq \Sigma^*$ heißt \mathcal{NP} -**vollständig** gdw.

$$A \in \mathcal{NP} \wedge \forall B \in \mathcal{NP} : B \leq_{\text{pol}} A$$

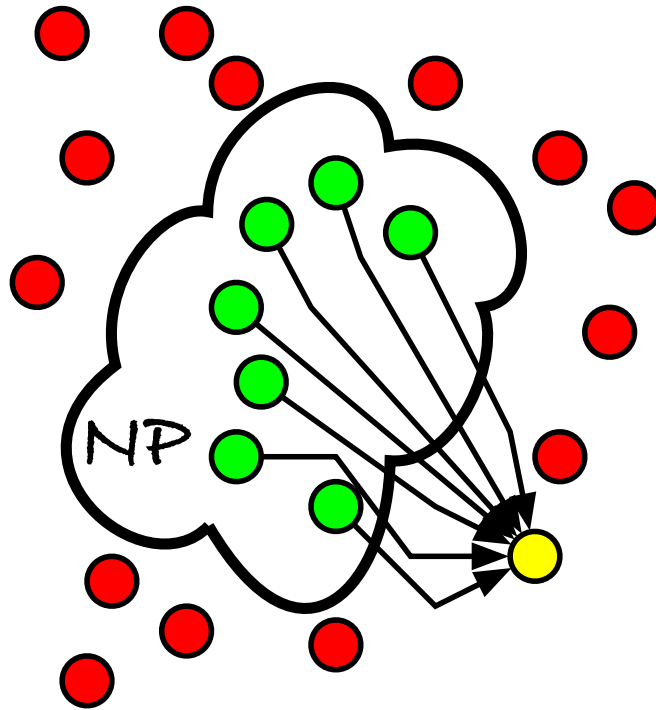
- Ein \mathcal{NP} -vollständiges Problem ist somit eines der **schwersten** bzw. **umfassendsten** Probleme innerhalb der Klasse \mathcal{NP} .
- Wichtige Eigenschaft: Transitivität von \leq_{pol} .
- Ist A ein \mathcal{NP} -vollständiges Problem und gilt $A \leq_{\text{pol}} B$, so ist auch B \mathcal{NP} -vollständig.



\mathcal{NP} -hart

■ Eine Menge $A \subseteq \Sigma^*$ heißt \mathcal{NP} -hart gdw.

$$\forall b \in \mathcal{NP} : B \leq_{\text{pol}} A$$

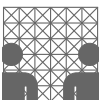


● ist \mathcal{NP} -hart

gdw.

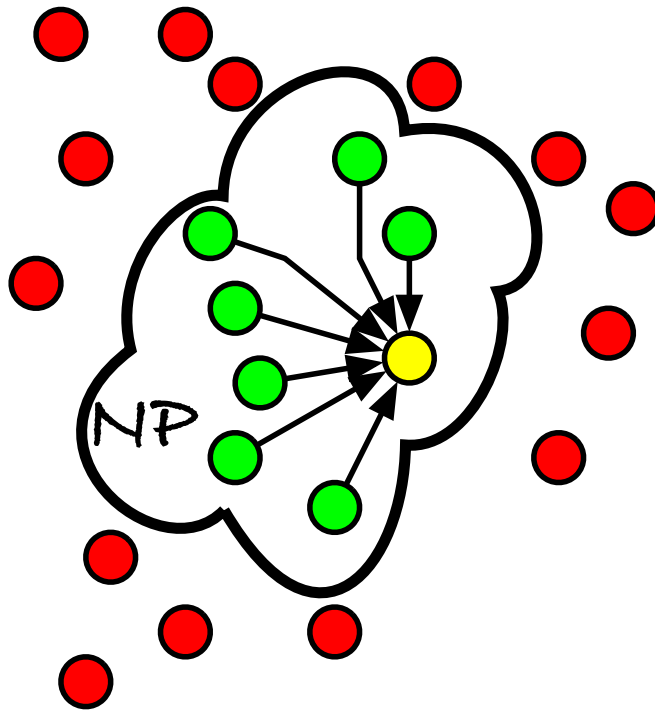
für alle ● aus \mathcal{NP}

gilt ● \leq_{pol} ●



\mathcal{NP} -vollständig

- Eine Menge $A \subseteq \Sigma^*$ heißt \mathcal{NP} -vollständig gdw.
 $A \in \mathcal{NP} \wedge \forall b \in \mathcal{NP} : B \leq_{\text{pol}} A$



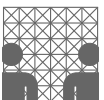
● ist \mathcal{NP} -vollständig

gdw.

1. ● liegt in \mathcal{NP}

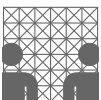
und

2. ● ist \mathcal{NP} -hart



Ein \mathcal{NP} -vollständiges Problem

- Bisher nur Definition der \mathcal{NP} -Vollständigkeit
- Aber: Gibt es überhaupt solche Probleme?
 - Diese Frage war lange Zeit ungeklärt!
 - Nach dem ersten folgten aber sofort viele \mathcal{NP} -vollständige Probleme.
 - Warum?!
- Idee für ein erstes \mathcal{NP} -vollständiges Problem:
 - Codierung aller Polynomialzeit-TM-Rechnungen
 - als Formel (SAT)
 - als Kachelproblem (2D-Domino)

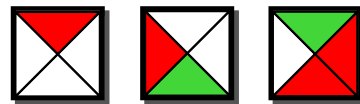




Das Kachelproblem ist in \mathcal{NP}

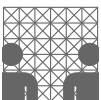
Gegeben: Eine Menge von r Kacheltypen $\mathcal{R} = \{K_1, K_2, \dots, K_r\}$, $n \in \mathbb{N}$ (beliebig viele Kacheln von jedem Typ)

Gesucht: Kann man eine Fläche der Größe $n \times n$ korrekt kacheln?

Antwort: JA oder NEIN

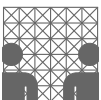


-  nur gleichfarbige Seiten dürfen aneinandergelegt werden!
-  Steine dürfen nicht gedreht werden!

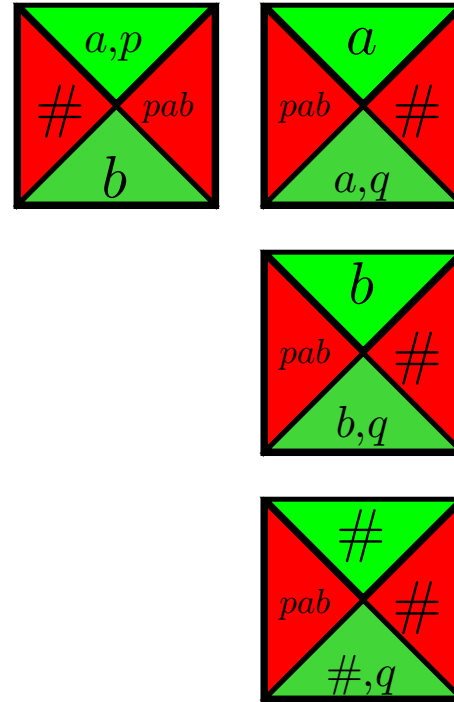
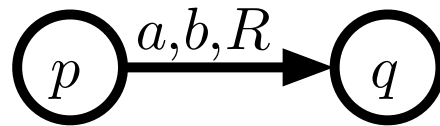


Das spezielle Kachelproblem

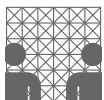
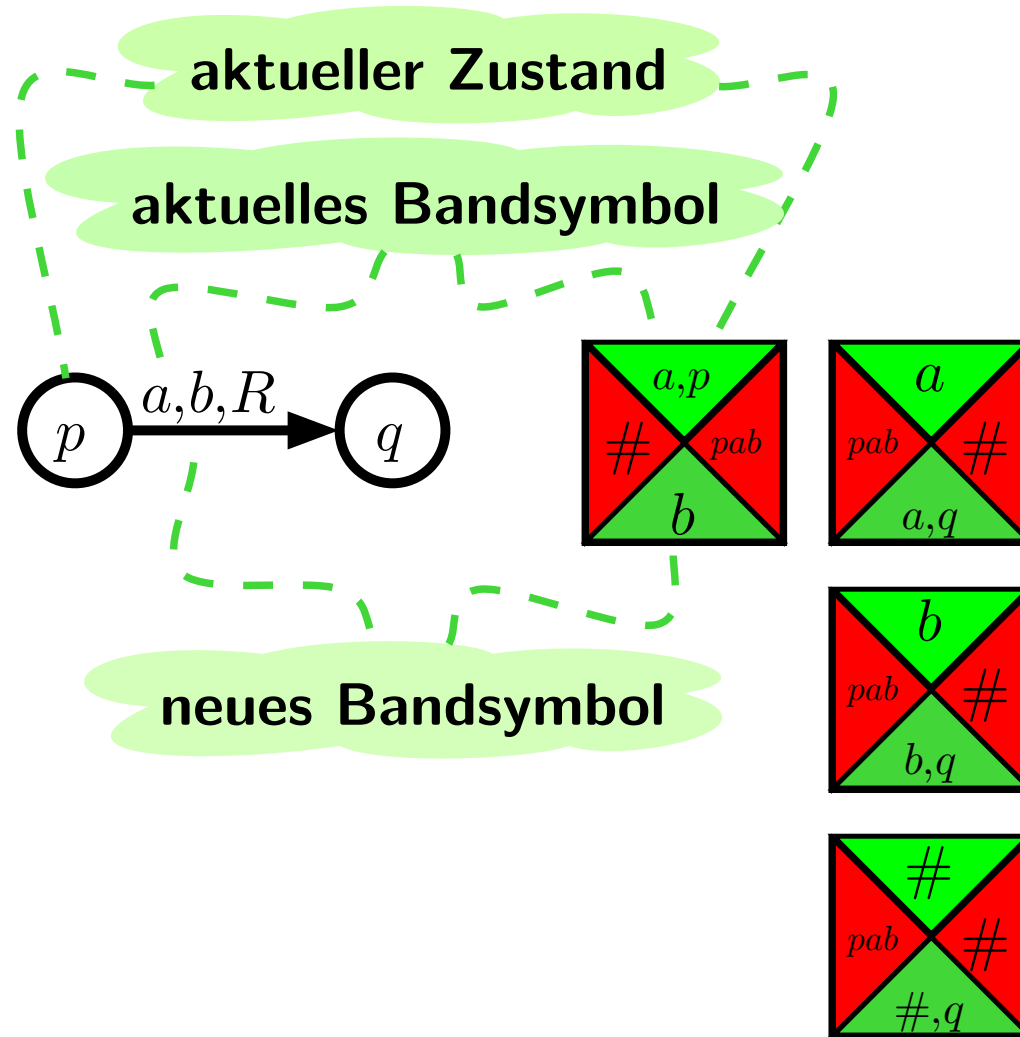
- endlicher Satz von Kacheln
- endliches Spielfeld der Größe $n \times n$
- erste Zeile festgelegt
 - hier: durch die Anfangskonfiguration einer (polynomialzeitbeschränkten) TM
 - Zeilenwechsel entspricht Konfigurationswechsel der TM
- Dieser Spezialfall des Kachelproblems wird nun als \mathcal{NP} -vollständig nachgewiesen.



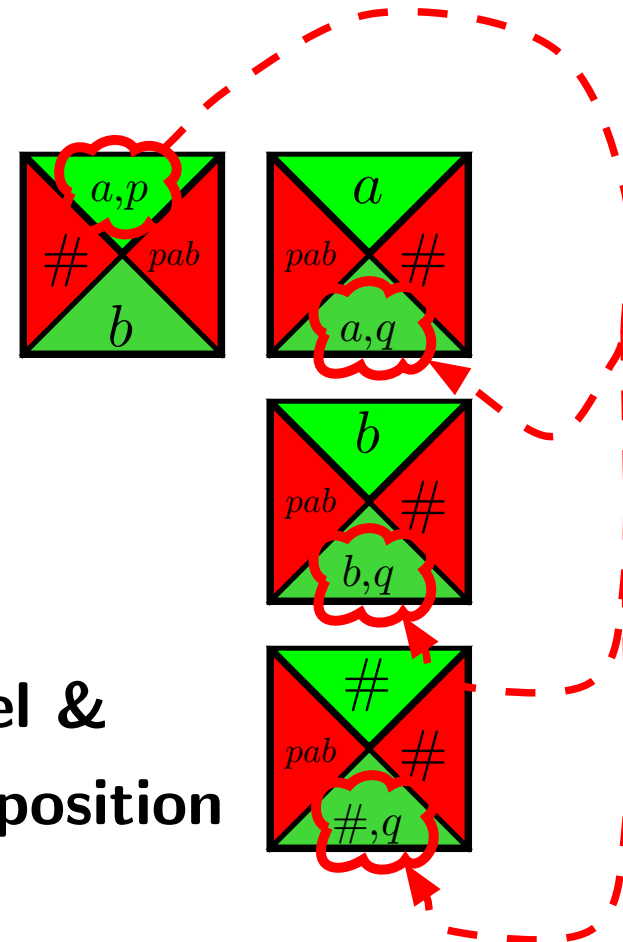
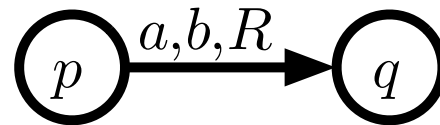
Das spezielle Kachelproblem



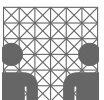
Das spezielle Kachelproblem



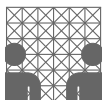
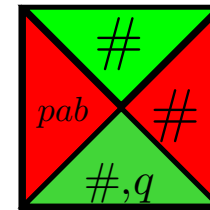
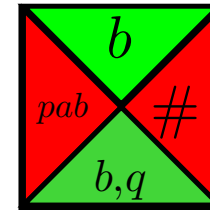
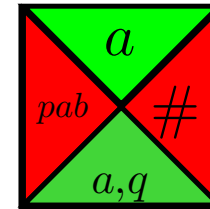
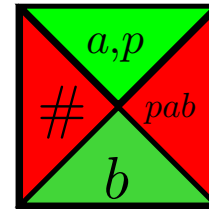
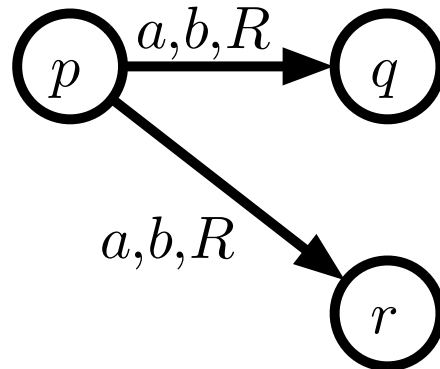
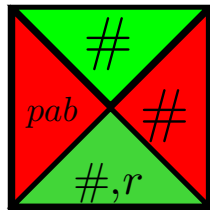
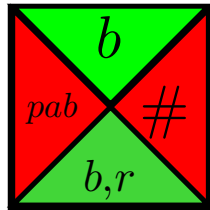
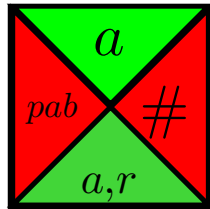
Das spezielle Kachelproblem



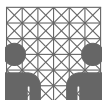
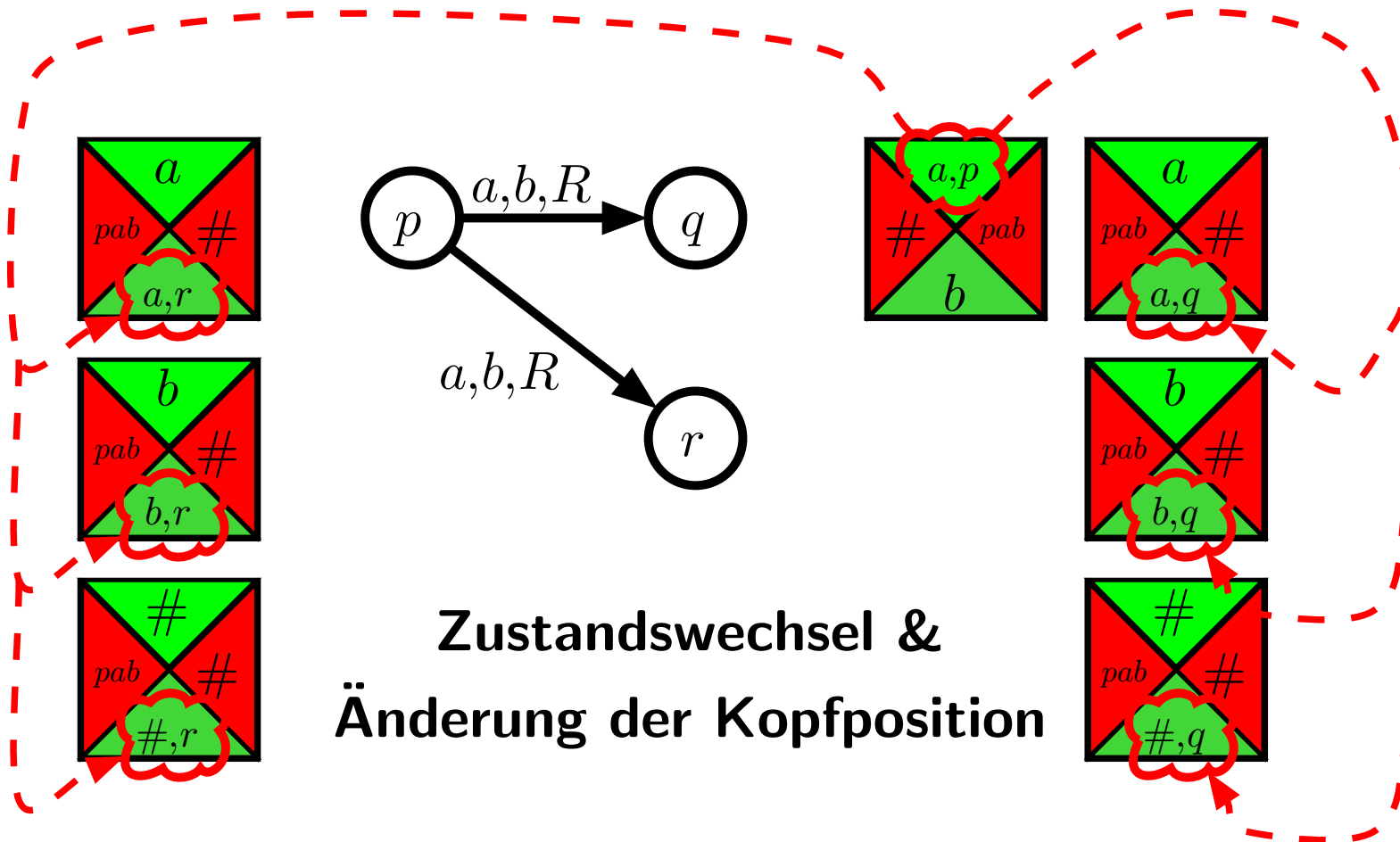
Zustandswechsel &
Änderung der Kopfposition



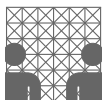
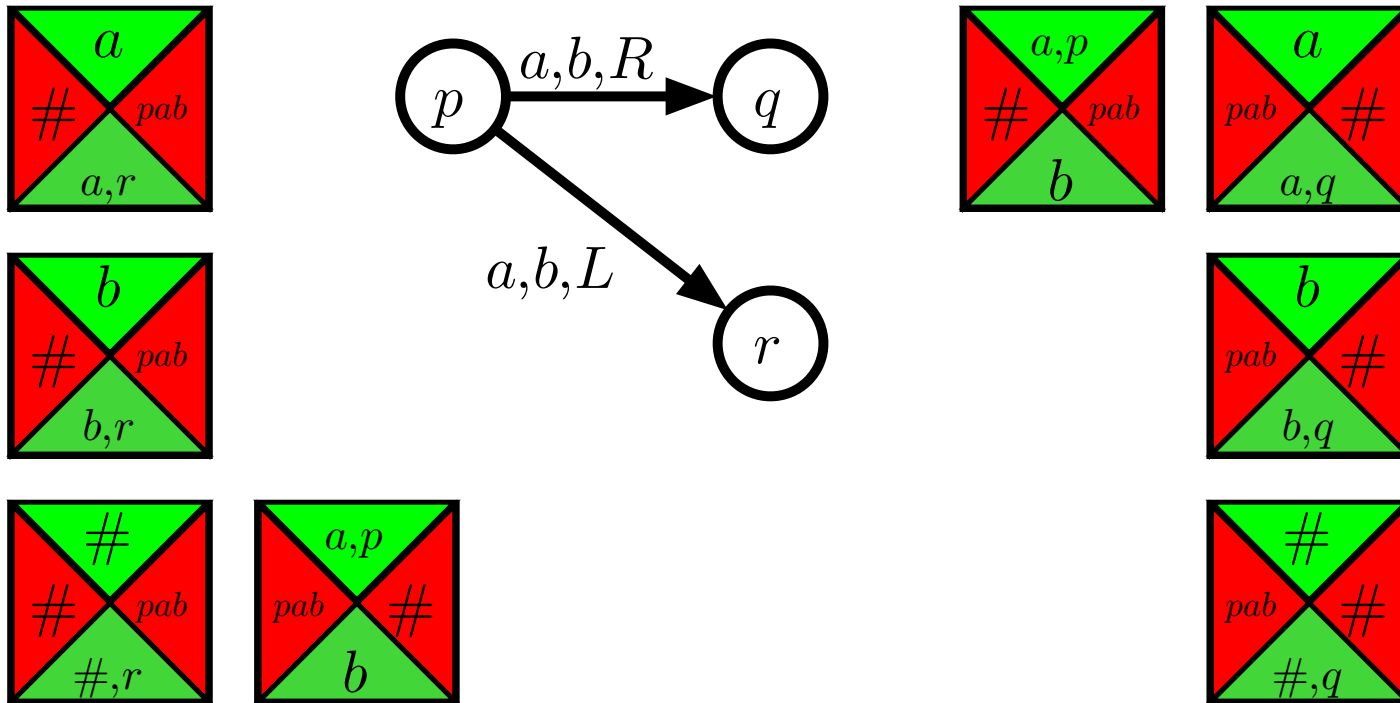
Das spezielle Kachelproblem



Das spezielle Kachelproblem

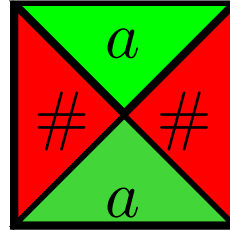


Das spezielle Kachelproblem

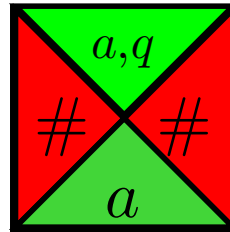


Formale Transformation

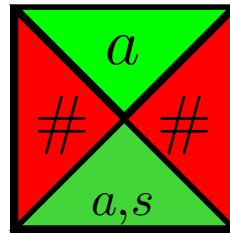
Für alle $a \in \Gamma$:



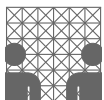
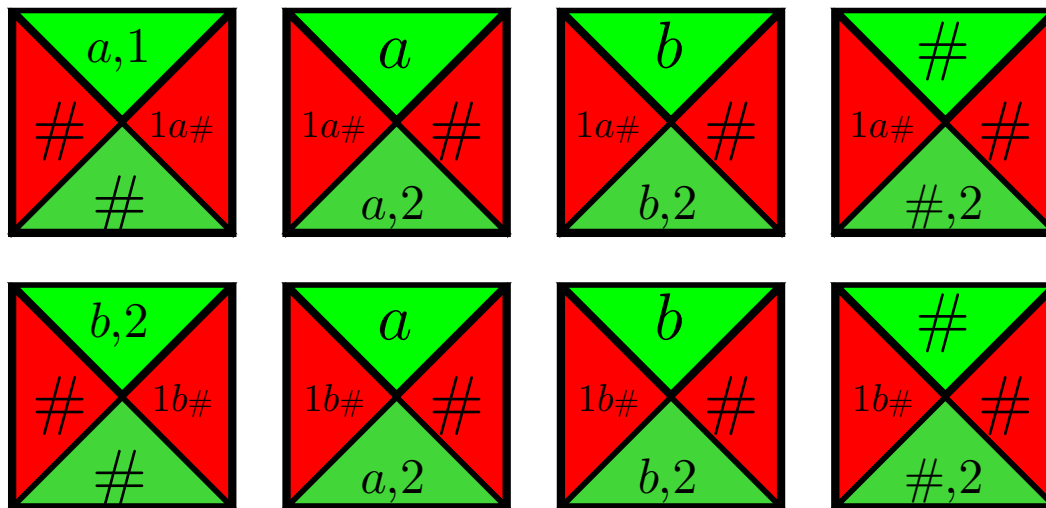
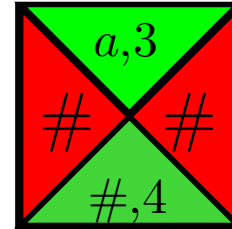
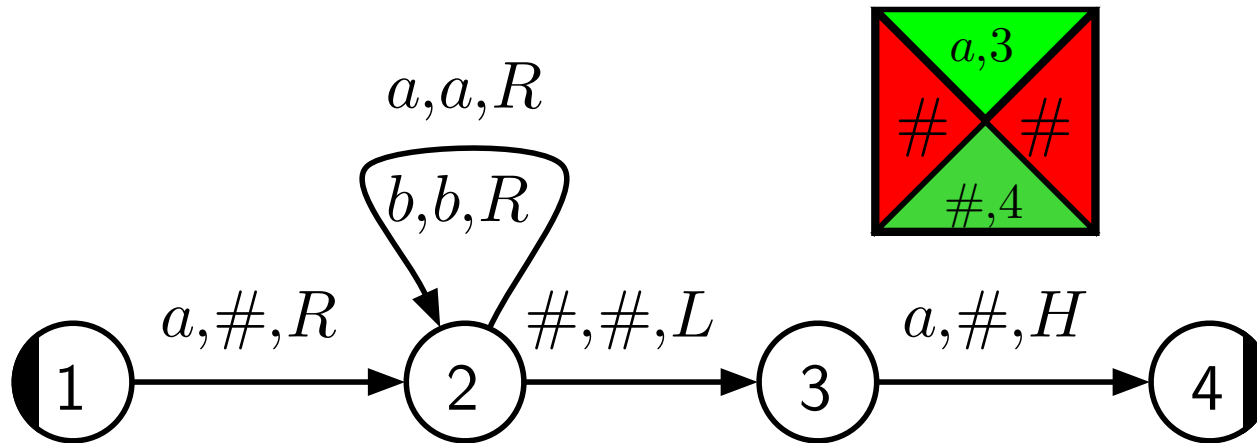
Für alle $q \in Z_{\text{end}}$
und $a \in \Gamma$:



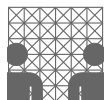
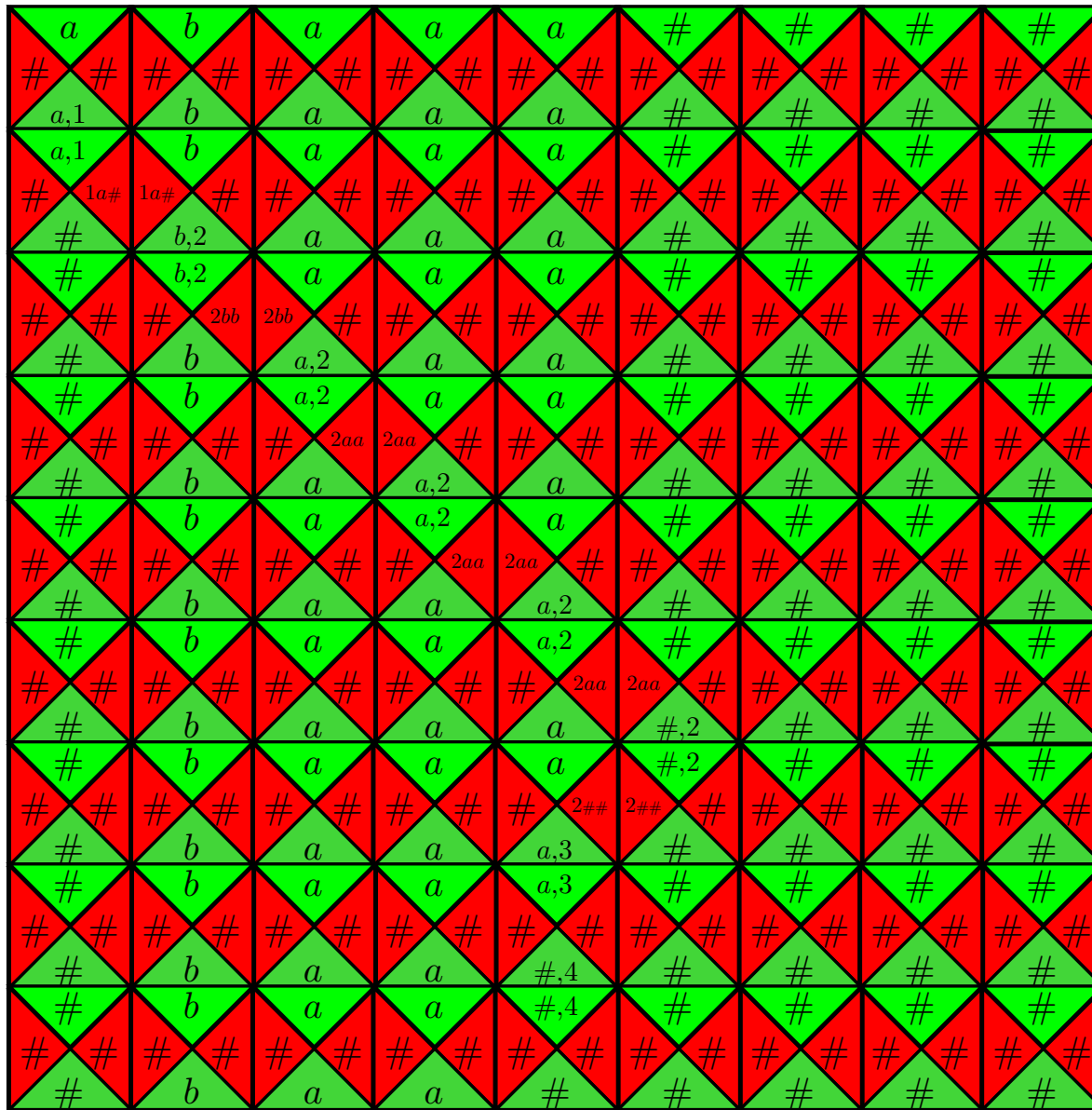
Für Startzustand
 s und alle $a \in \Gamma$:



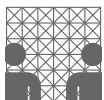
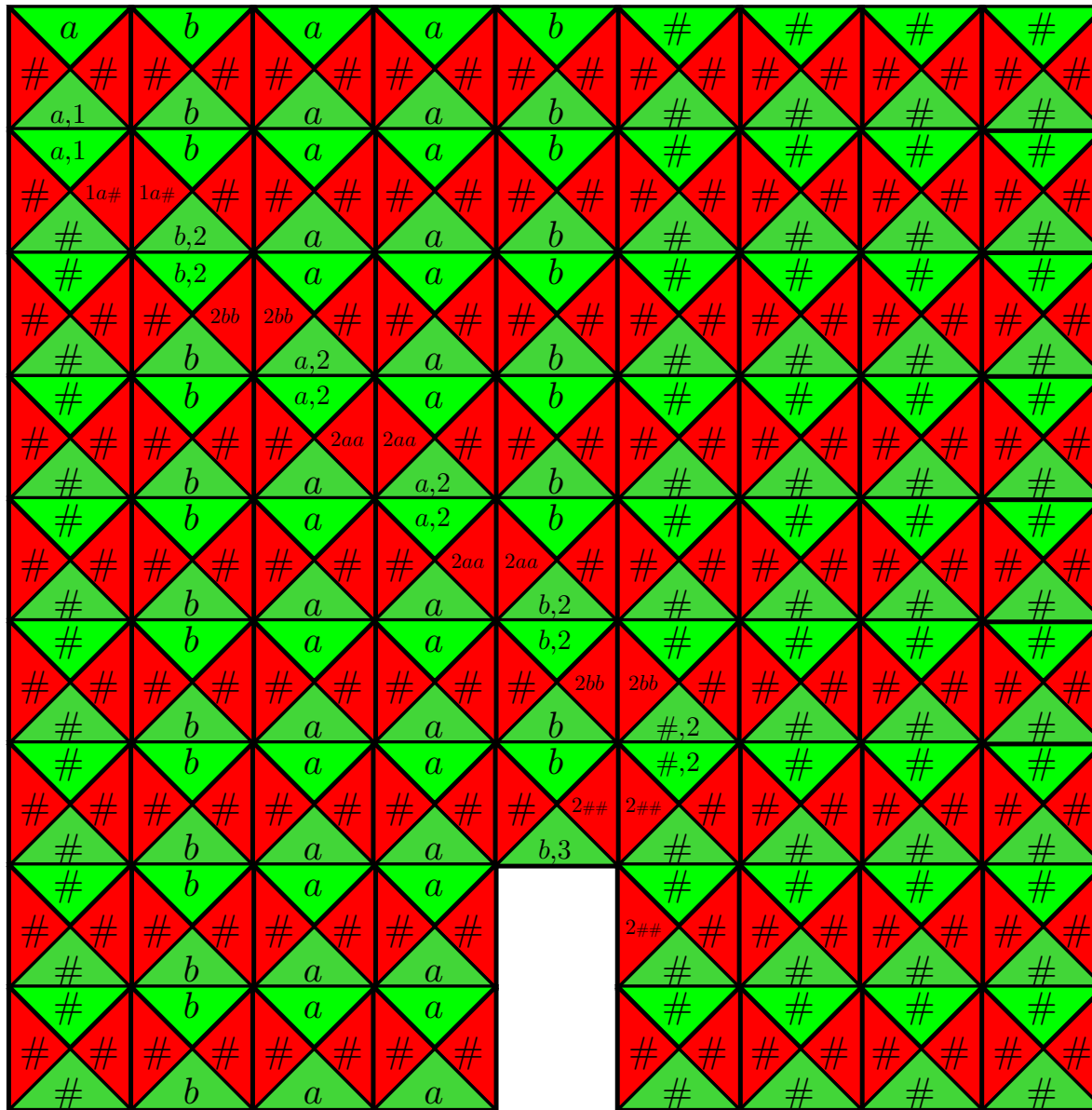
Eine TM M mit $L(M) \in \mathcal{P}$



Beispiel-TM (2)



Beispiel-TM (3)



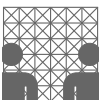
SAT

Definition: $SAT := \{w \in X^* \mid w \text{ ist ein erfüllbarer boolescher Ausdruck}\}$ ist das **Erfüllbarkeitsproblem**:

Gegeben: Eine Menge V von Variablen und eine boolesche (aussagenlogische) Formel $B \in X^*$ mit
 $X := V \cup \{0, 1, \vee, \wedge, \neg, (,)\}$.

Gesucht: Gibt es eine Belegung der Variablen mit TRUE (1) und FALSE (0), so dass B zu TRUE (1) evaluiert?

Antwort: JA oder NEIN

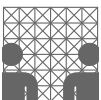


\mathcal{NP} -Vollständigkeit von SAT

Theorem: SAT ist \mathcal{NP} -vollständig.

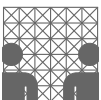
Beweisidee:

- Es ist einfach zu sehen, dass $SAT \in \mathcal{NP}$ gilt.
 - Zu einer gegebenen Formel B mit den Variablen x_1, x_2, \dots, x_n wird in Linearzeit nichtdeterministisch eine Belegung der Variablen mit TRUE oder FALSE geraten.
 - Es wird geprüft, ob B mit dieser Belegung den Wert TRUE bekommt. Dies ist in Polynomzeit möglich.
- Bleibt zu Zeigen, dass jedes andere Problem aus \mathcal{NP} in Polynomialzeit auf SAT reduzierbar ist.



Reduktion auf SAT

- $\text{FELD}(i, j, t) \hat{=} \text{In Konfiguration } k_t \text{ steht das Zeichen } x_j \text{ in Feld } i.$
- $\text{ZUSTAND}(r, t) \hat{=} \text{In der Konfiguration } k_t \text{ befindet sich die TM } A_L \text{ im Zustand } z_r.$
- $\text{KOPF}(i, t) \hat{=} \text{In der Konfiguration } k_t \text{ steht der LSK auf dem Feld } i.$
- Anzahl dieser Variablen in $O(p(n)^2)$, wenn $p(n)$ die Zeitschranke ist. Wir definieren die Formel
$$\oplus(a_1, a_2, \dots, a_r) := (a_1 \vee a_2 \vee \dots \vee a_r) \wedge \bigwedge_{i \neq j} (\neg a_i \vee \neg a_j).$$
Die Länge dieser Formel ist von der Ordnung $O(r^2)$.
- Die Formel F_w hat die Form
$$F_w := A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G.$$



Reduktion auf SAT (2)

Die einzelnen Teilformeln bedeuten:

$A \hat{=} \text{In jeder Konfiguration } k_t \text{ steht der LSK auf genau einem Feld.}$

$B \hat{=} \text{In } k_t \text{ enthält jedes Feld genau ein Zeichen.}$

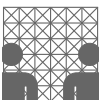
$C \hat{=} \text{In } k_t \text{ befindet sich } A_L \text{ in genau einem Zustand.}$

$D \hat{=} \text{Bei jedem Übergang wird genau das Feld verändert, auf das der LSK zeigt.}$

$E \hat{=} \text{Jeder Übergang entspricht der Turing-Tafel.}$

$F \hat{=} \text{Die erste Konfiguration ist } k_0 = q_0 w \# \dots \#.$

$G \hat{=} \text{Der Zustand in der letzten Konfiguration } k_{p(n)} \text{ ist Endzustand aus } Z_{\text{end}}.$



Beispiel einer Teilformel

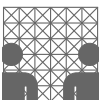
- Es ist $A := A_1 \wedge A_2 \wedge \dots \wedge A_{p(n)}$ und
 $\forall t \leq p(n) : A_t := \bigoplus(\text{KOPF}(1, t), \text{KOPF}(2, t), \dots, \text{KOPF}(p(n), t))$
- Auch B und C sind zusammengesetzte Formeln:

$$B := \bigwedge_{1 \leq i, t \leq p(n)} B(i, t) \text{ mit}$$

$$B(i, t) := \bigoplus(\text{FELD}(i, 1, t), \dots, \text{FELD}(i, m, t)), \quad m := |Y|$$

$$C := \bigwedge_{1 \leq t \leq p(n)} C_t \text{ mit}$$

$$C_t := \bigoplus(\text{ZUSTAND}(1, t), \dots, \text{ZUSTAND}(s, t)), \\ s := |Z|$$



Einfache Folgerungen

Lemma: Sei L \mathcal{NP} -vollständig, $L \leq_{pol} M$ und $M \in \mathcal{NP}$.
Dann ist auch M \mathcal{NP} -vollständig

Beweis: Das Lemma folgt direkt aus der Definition von \mathcal{NP} -Vollständigkeit und der Transitivität der Polynomzeitreduktionen.

- Beispiel im Skript: KNF ist \mathcal{NP} -vollständig.
- $KNF := \{w \in X^* \mid w \text{ ist eine erfüllbare boolesche Formel in konjunktiver Normalform}\}$



andere \mathcal{NP} -vollständige Probleme

Definition: Das Erfüllbarkeitsproblem boolescher Formeln in konjunktiver Normalform dargestellt als Sprache:

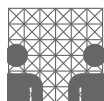
$KNF := \{w \in X^* \mid w \text{ ist eine erfüllbare boolesche Formel in konjunktiver Normalform}\}$

Theorem: KNF ist \mathcal{NP} -vollständig.

Definition: Die Sprache $3\text{-SAT} \subsetneq KNF \subsetneq SAT$ ist gegeben durch:

$3\text{-SAT} := \{w \in X^* \mid w \text{ ist erfüllbare boolesche Formel in konjunktiver Normalform mit genau 3 Literalen in jeder Klausel}\}$.

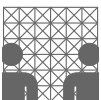
Theorem: 3-SAT ist \mathcal{NP} -vollständig.



\mathcal{NP} -vollständige Probleme (2)

Theorem:

- Das Hamilton-Kreis Problem H_c ist \mathcal{NP} -vollständig.
- Das Problem L_w („Längster Weg zwischen zwei Knoten“) ist NP-vollständig.
- Das Rucksackproblem ist \mathcal{NP} -vollständig.
- Das Partitionierungsproblem von Graphen ist \mathcal{NP} -vollständig.
- Das Problem $CLIQUE$ ist \mathcal{NP} -vollständig.

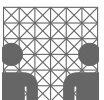


\mathcal{NP} -vollständige Probleme in \mathcal{P} ?

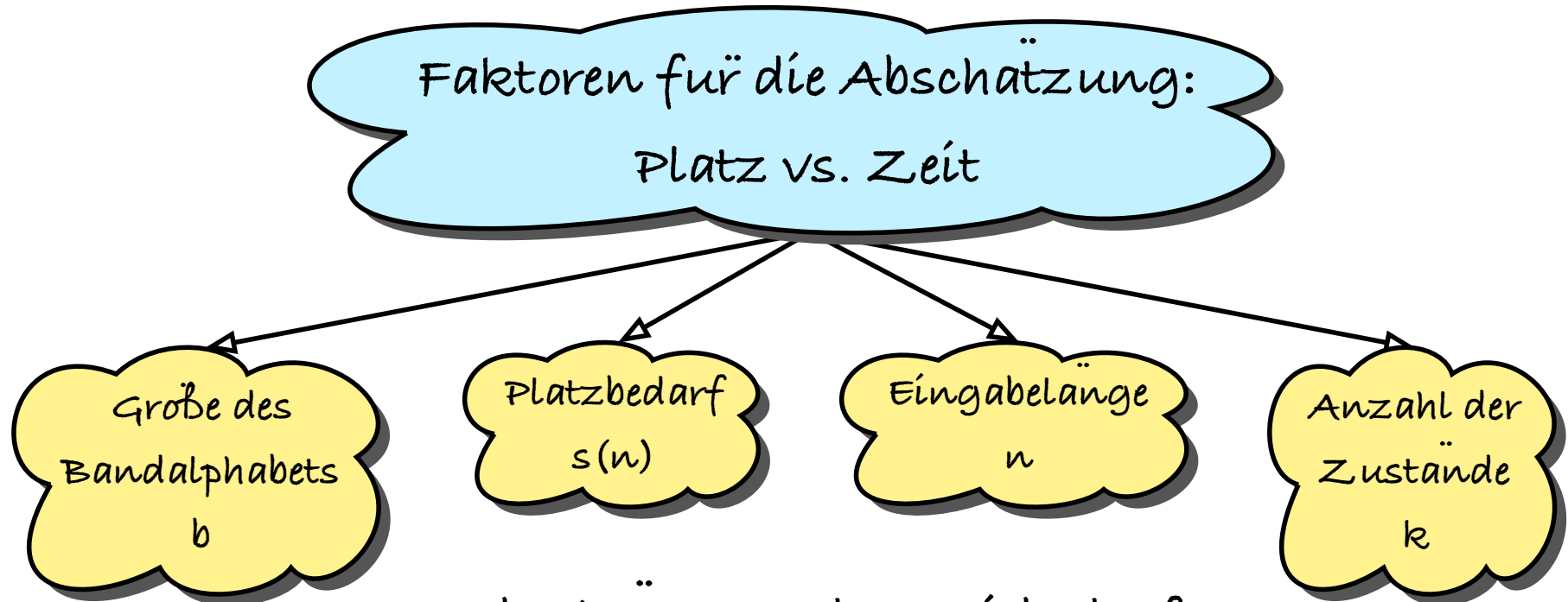
- Daß $\mathcal{P} \subseteq \mathcal{NP}$ gilt ist klar!
- Könnte auch das Gegenteil, also $\mathcal{P} \subseteq \mathcal{NP}$, gelten?
 - Man bräuchte nur von einem einzigen \mathcal{NP} -vollständigen Problem A zeigen, dass es in \mathcal{P} liegt!
 - Dann würde nämlich sofort folgen:

$$\forall B \in \mathcal{NP} : B \leq_{\text{pol}} A \quad \Rightarrow \quad \forall B \in \mathcal{NP} : B \in \mathcal{P}$$

- Dies ist bislang noch niemandem gelungen!!!



Band- vs. Zeitbedarf



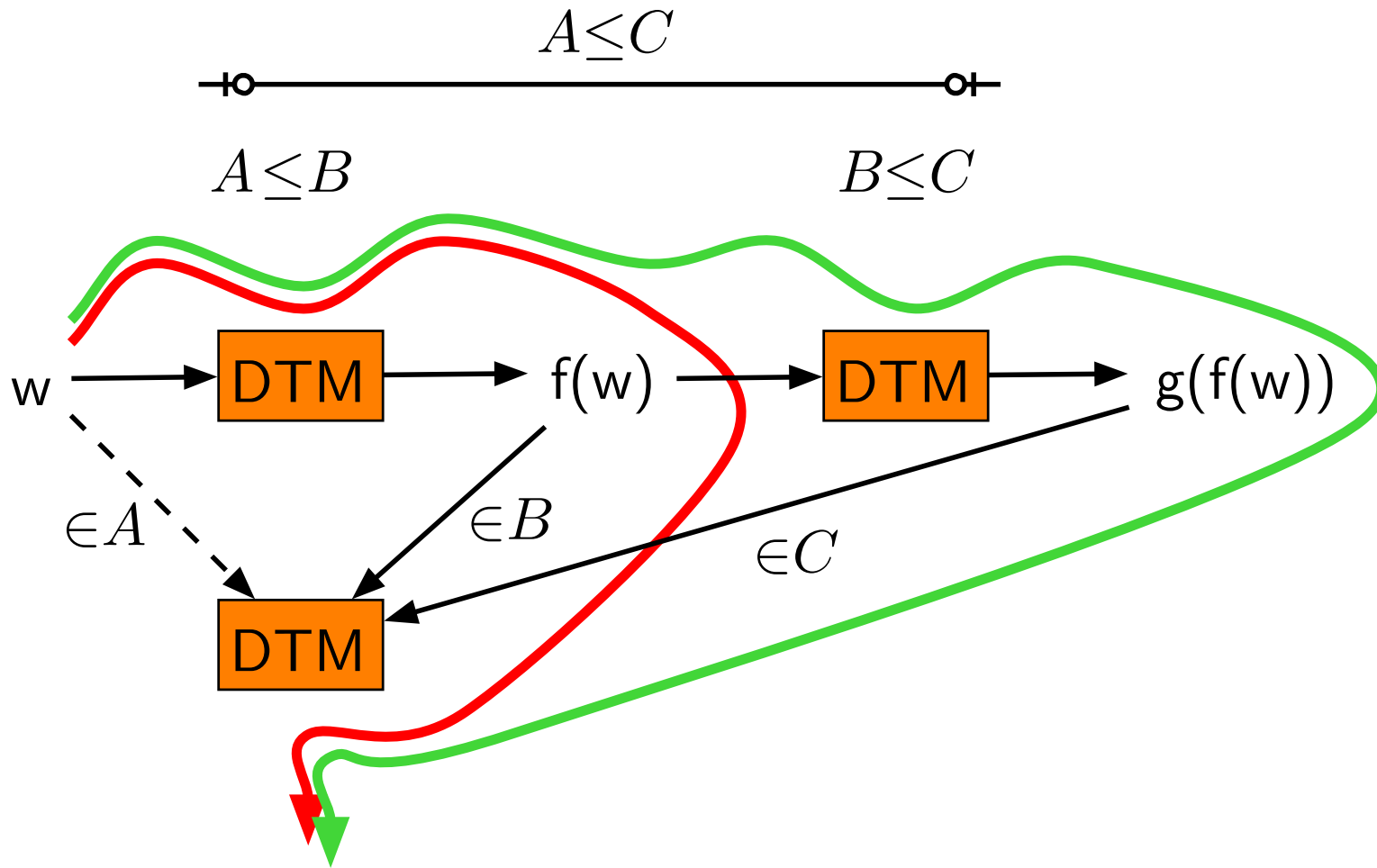
Abschätzung des Zeitbedarfs:

$$n \cdot s(n) \cdot k \cdot b^{s(n)} \in O(c^{s(n)})$$



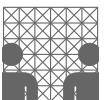
Transitivität von Reduktionen

Allgemein:



Transitivität von Reduktionen

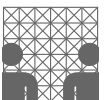
- generelles Problem bei Komplexitätsbeschränkung für die Reduktion:
 - Wie groß wird das Zwischenergebnis?
- **Polynomialzeitreduktion:**
 - $|f(w)|$ ist maximal $p(|w|)$ für ein Polynom $p \in O(t_f)$.
 - Platz auf Arbeitsbändern ist **nicht** begrenzt.
 - $g(f(w))$ wird in Polynomzeit (in Abhängigkeit von $|f(w)|$) berechnet, also existiert Polynom $q \in O(t_g)$ und insgesamt wird $q(p(|w|))$ Zeit benötigt.
 - Dies ist wiederum ein Polynom!



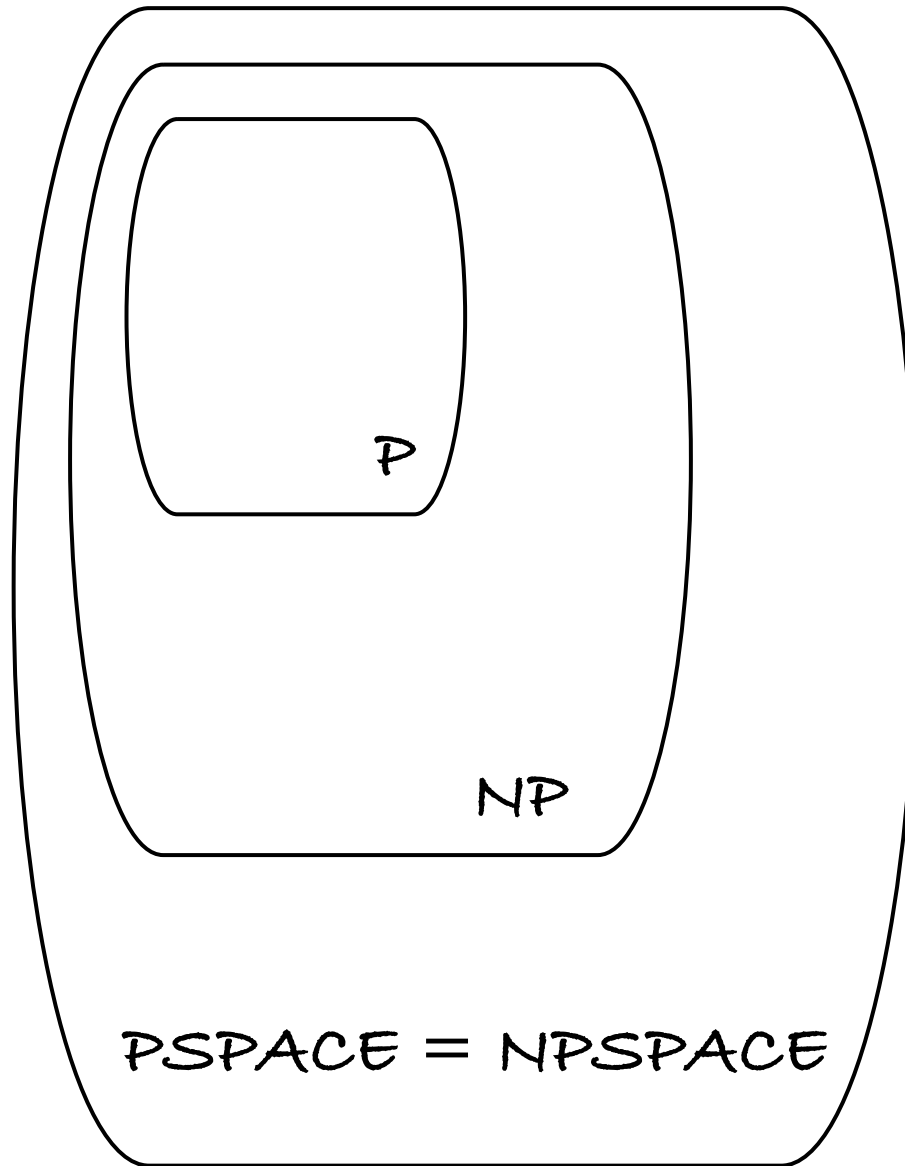
Transitivität von Reduktionen

■ logarithmische-Platz-Reduktion:

- Zwischenergebnis $f(w)$ kann die logarithmische Länge bzgl. der Eingabelänge $|w|$ überschreiten!
- $c^{\log(n)} \notin O(\log(n))$
- $f(w)$ ist somit nicht mehr auf einem Arbeitsband speicherbar!!
- Es muss **grundlegend anders** gearbeitet werden:
 - Nur das gerade benötigte Symbol der Ausgabe generieren!
 - Speicherung der Position des Symbols in log-Platz (binär)
 - Rechenzeit ist unbegrenzt!



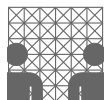
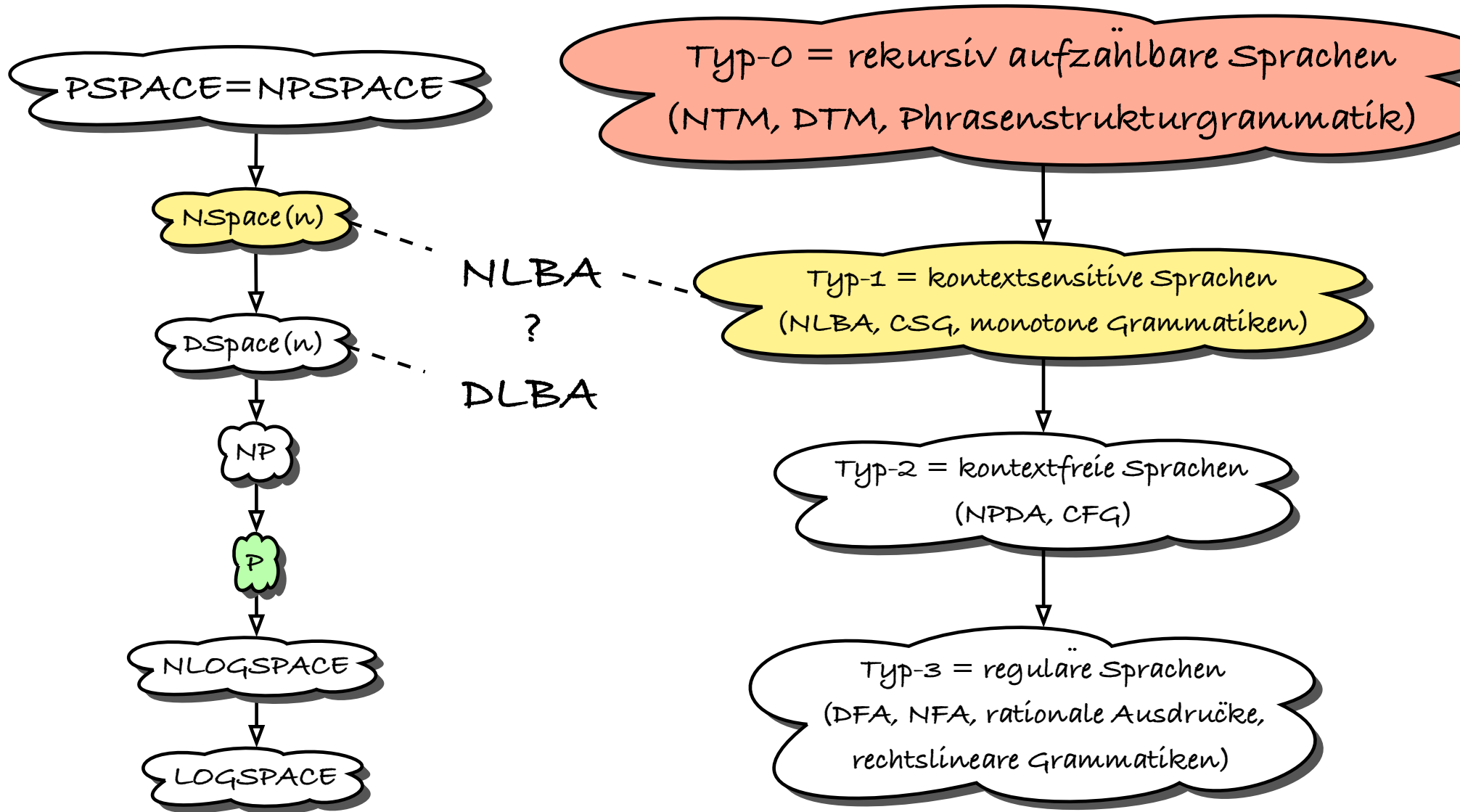
Zusammenhänge



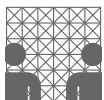
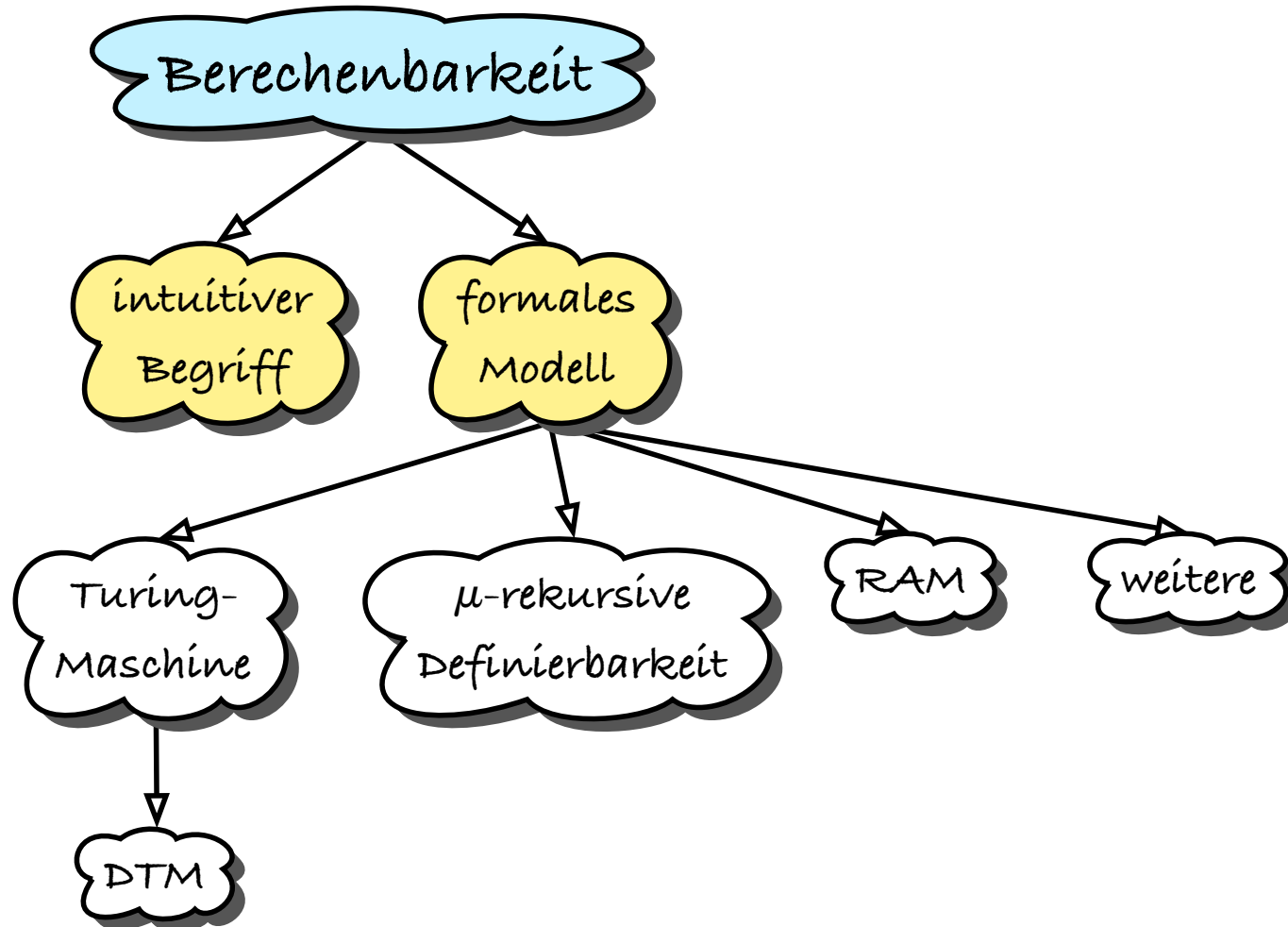
$\text{NSpace}(s(n))$
 $\in \text{DSPACE}(s(n)^2)$



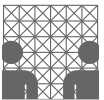
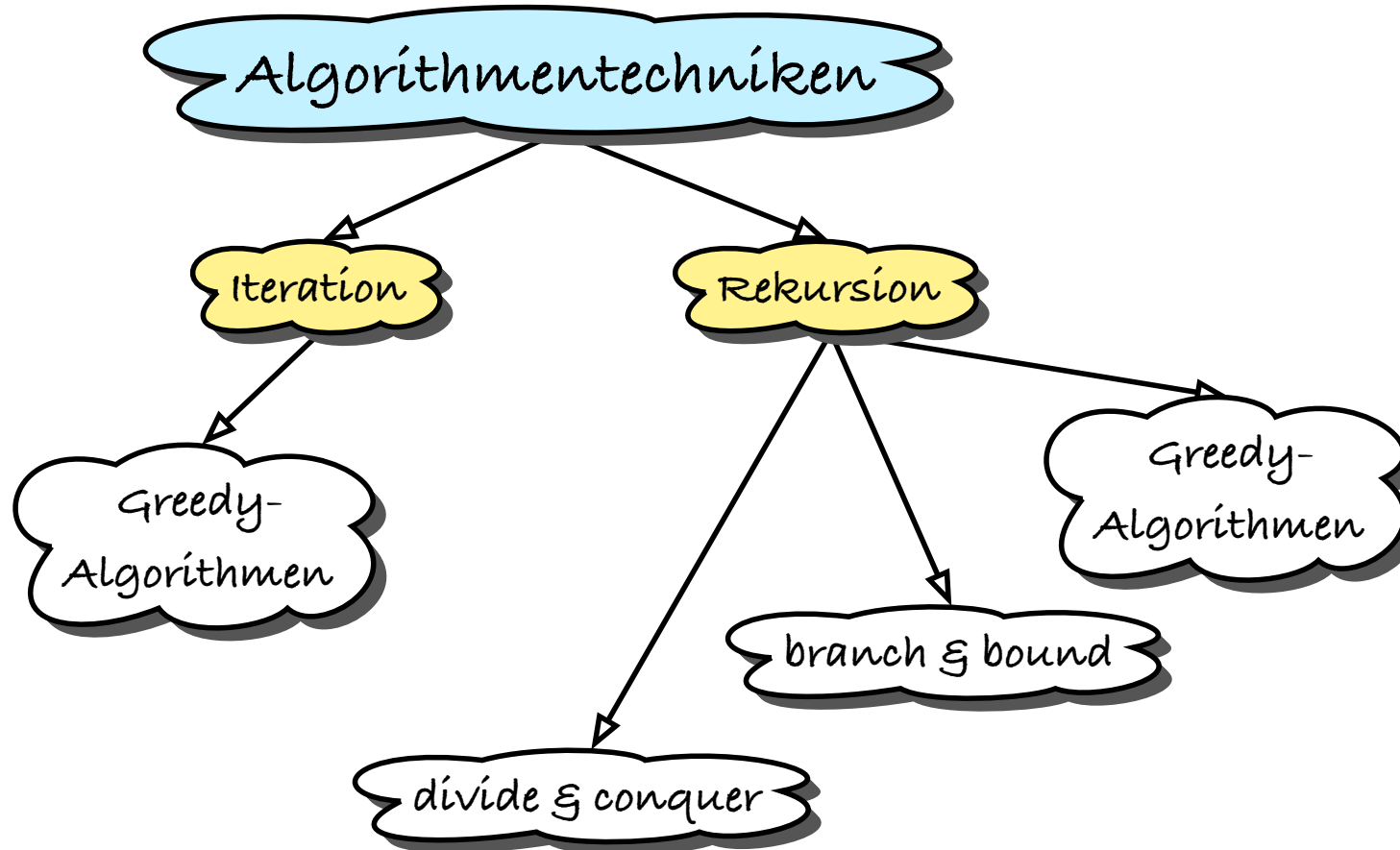
Zusammenhänge



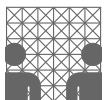
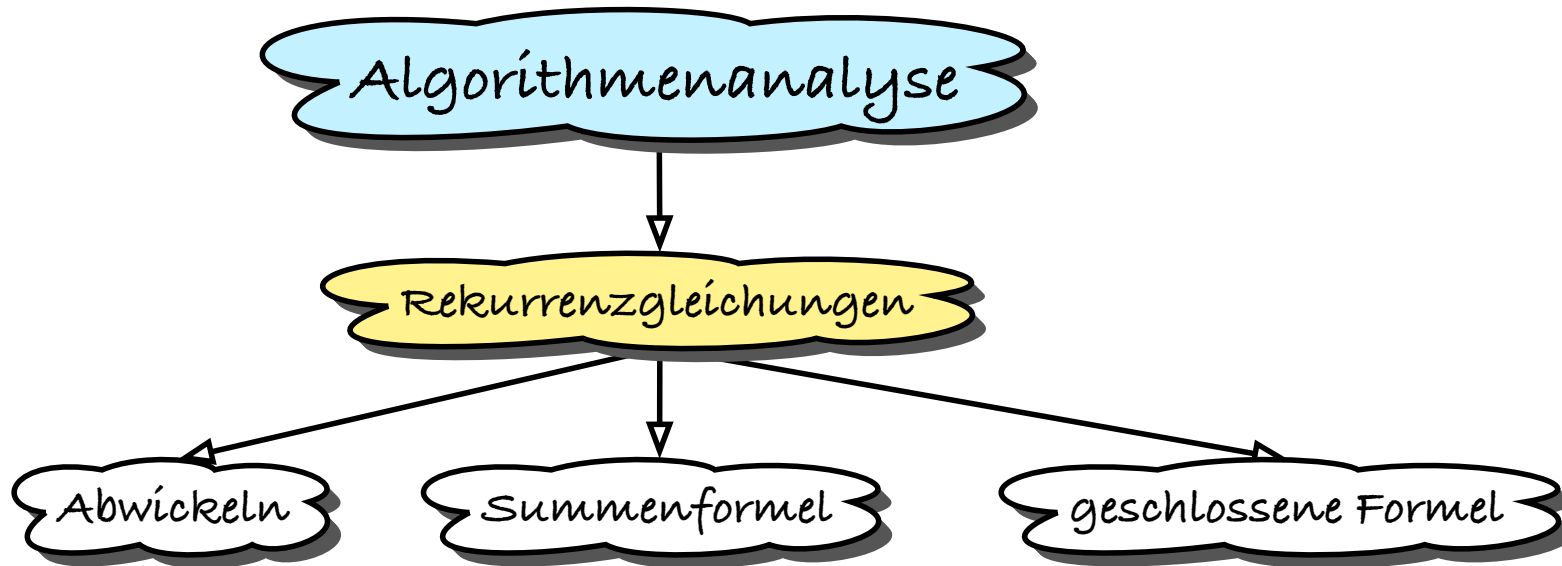
Berechenbarkeit



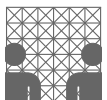
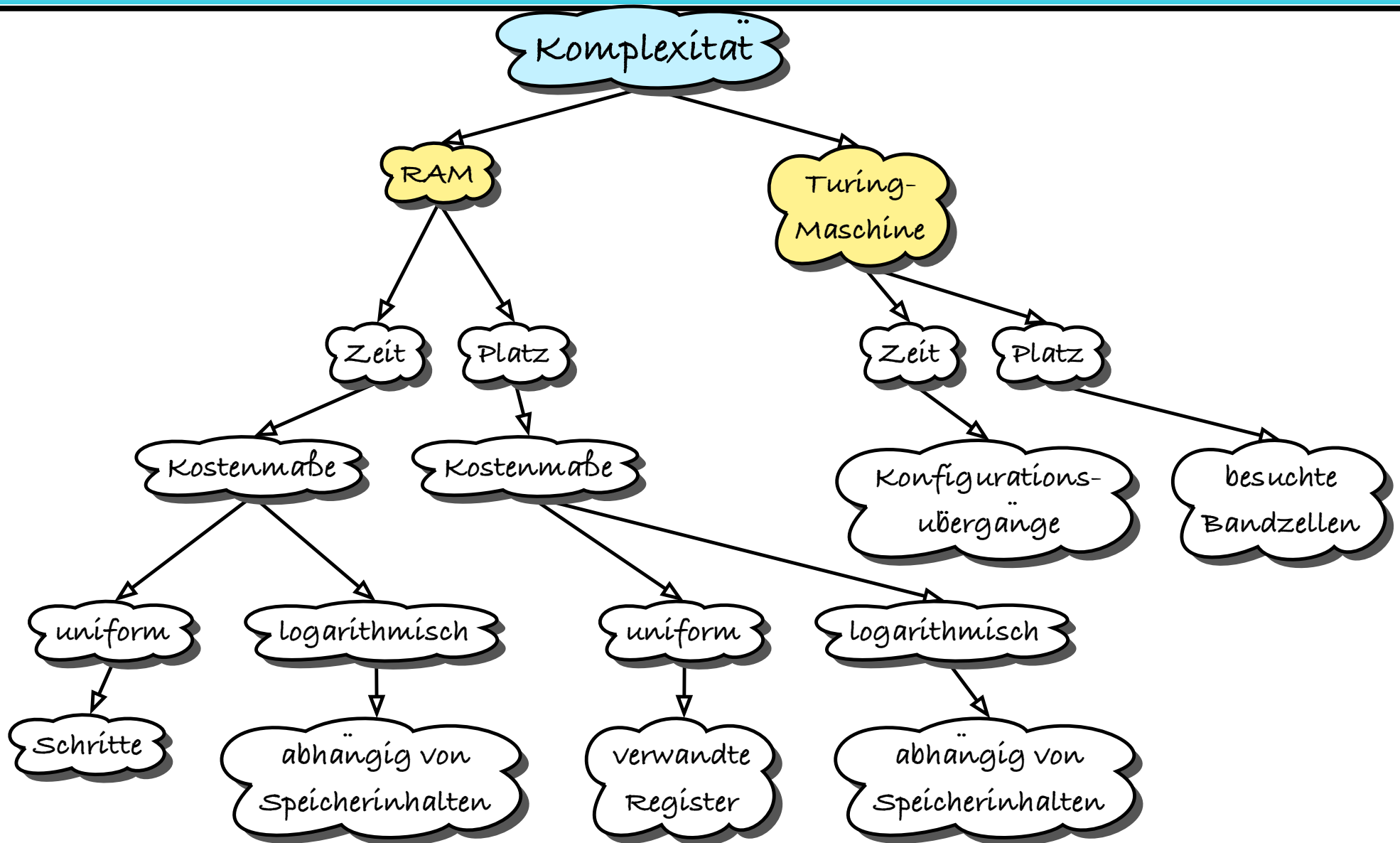
Algorithmentechniken



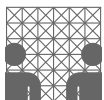
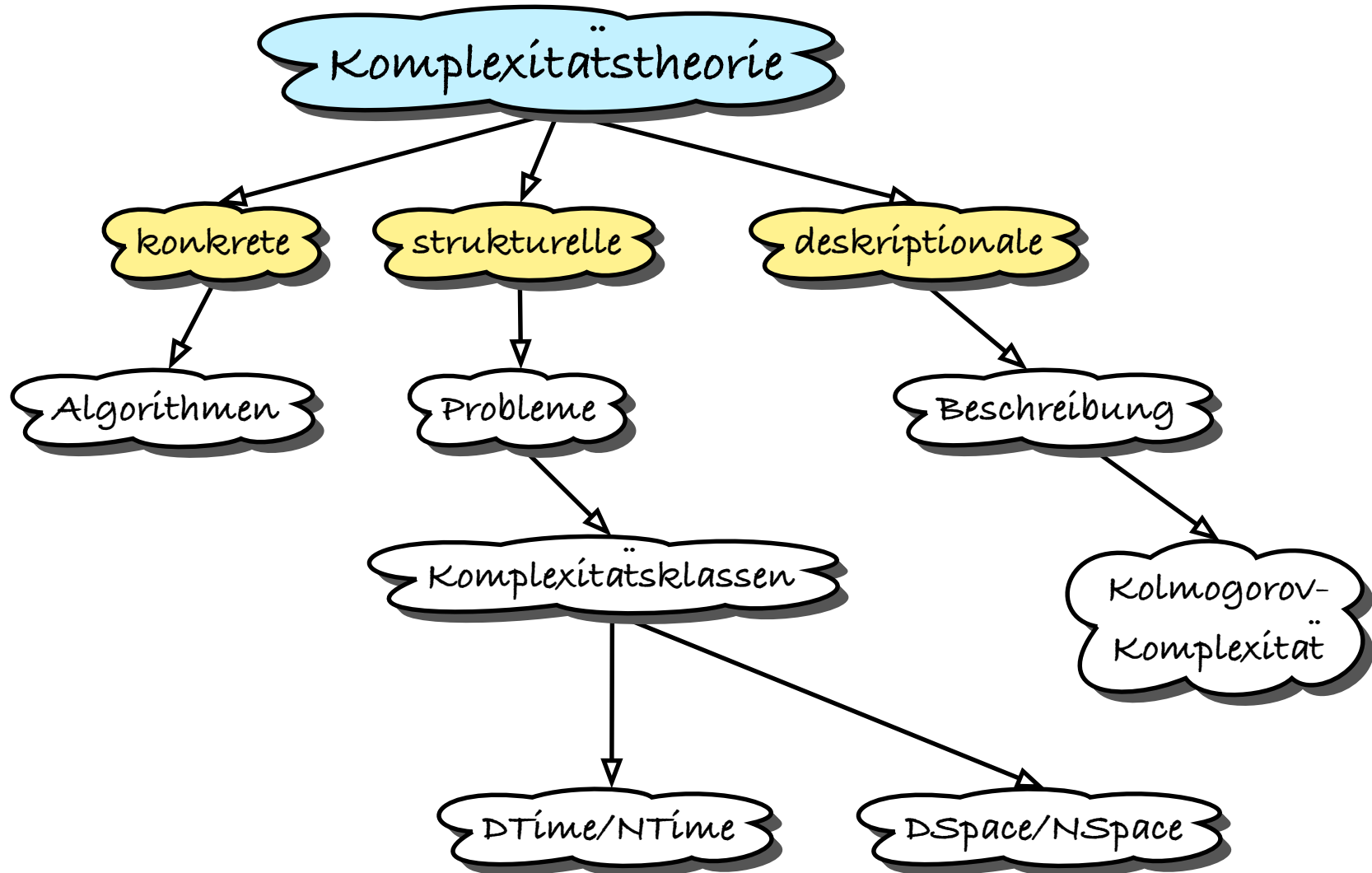
Algorithmenanalyse



Komplexität

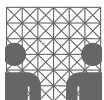


Überblick: Komplexitätstheorie

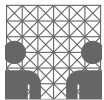
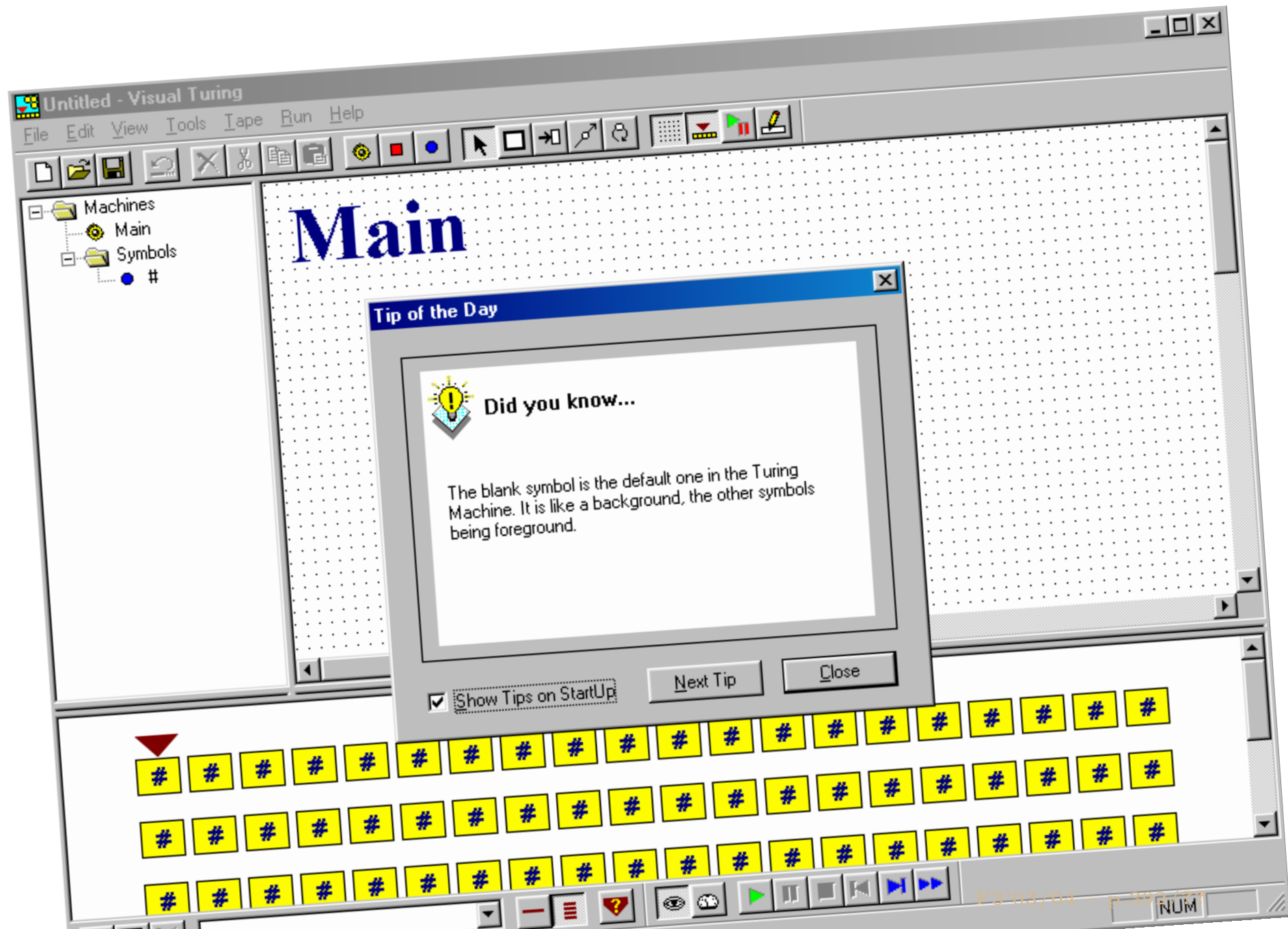


Was fehlt noch?!

- parallele und verteilte Berechnungsmodelle
- parallele Komplexität
- probabilistische Berechnungsmodelle (probabilistische TM, probabilistischer endlicher Automat)
- formale Ansätze zur Verifikation (unter Verwendung von speziellen Logiken)
- zu finden in F4, PNL, AUK, ...



... übrigens



... übrigens

The screenshot displays the 'Visual Turing' software interface. The title bar reads 'Step 2 Copying strings - Visual Turing'. The menu bar includes 'File', 'Edit', 'View', 'Tools', 'Iape', 'Run', and 'Help'. A toolbar with various icons is located below the menu bar.

On the left, a 'Machines' tree view shows a folder named 'Main' containing 'alpha', 'Left #', and 'Right #'. Below it, a 'Symbols' list includes '# Blank', 'a', and 'b'.

The main workspace, titled 'Main', contains a state transition diagram for a Turing Machine. The diagram starts with a state 'R' (represented by a hand icon) on a tape containing 'Left #'. Transitions lead to states 'alpha' (green square with a pencil icon), '# Blank' (green square with a pencil icon), and 'Right #' (blue square with a pencil icon). The diagram shows the machine moving the head to the right and copying the input string onto a blank tape.

Below the diagram, a text box explains the machine's function: 'This program shows how can a Turing Machine copy a string. It performs the following function : having an input string like #,s1,s2,...,sn,# with the head on the latter blank, it copies the string and the tape become like that : #s1,s2,...,sn,#s1,s2,...,sn,#. Formally, this machine transforms #w# into #w#w#, where w is a string. This copy machine can be found in the machines library - the "Machines Library.tur" file. Resolution recommended : 800 x 600 - maximized.'

At the bottom, a tape simulation shows three rows of yellow cells. The first row contains the string '# a b b b a b b # # # # # # # # # # # # # # # #'. A red arrow points to the first blank cell after the input string. The second and third rows consist of blank cells.

The bottom status bar shows 'Demo tape 1' and a series of control buttons. The system tray in the bottom right corner displays 'E8'03/04' and 'NUM'.



Know-how  Optimierung

Dr. Armin Scholl, Gabriela Krispin,
Robert Klein, Prof. Wolfgang Domschke

Besser beschränkt

Clever optimieren mit *Branch and Bound*

Ob schnellste Route,
profitabelste
Investition oder
geringster Verschleiß:
Bäume und
Schranken helfen,
optimale Lösungen
zu finden.

 Optimierung

Bei der Fahrt ins Wochenende ist die Lage noch übersichtlich. Viele Wege führen zum Ziel, aber nur wenige kommen ernsthaft in Frage. Eine zumindest nahezu optimale Route findet der Ausflügler ohne Mühe auf der Karte.

Leider sind viele Optimierungsaufgaben aus der Geschäftswelt wesentlich komplexer, sei es bei der Steuerung von Produktionsanlagen oder bei der Auswahl von gewinnträchtigen Investitionen. Die Anzahl der Alternativen bei solchen kombinatorischen Optimierungsproblemen ist riesig und wächst exponentiell mit zunehmendem Umfang der Aufgabe. Das vollständige Enumerieren (Aufzählen) kommt dann nicht mehr in Frage. Meistens begnügt man sich mit einer nicht optimalen Lösung der Aufgabe. Das muß aber nicht sein – Branch and Bound zeigt, wie es besser geht.

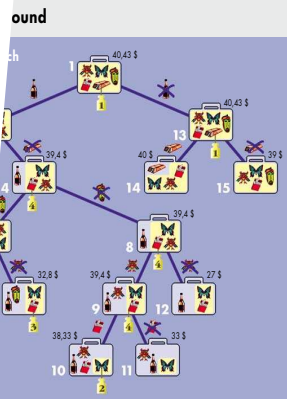
Eine geschickte Anwendung kann zu sehr schnellen Optimierungsalgorithmen führen. In vielen Bereichen ist Branch and Bound daher zur Standardtechnik geworden, etwa in der Logistik bei der Transport-, Touren- und Standortplanung [1]. Andere wichtige Anwendungsfelder sind Produktions-, Projekt- und Investitionsplanung [2].

Geheimdiplomatie

Dankbares Beispiel ist wieder der deutsche Diplomat Herbert B., der schon bei der Vorstellung des Optimierungsverfahrens *Tabu Search* [3] gute Dienste leistete. Da das Auswärtige Amt den Fall totschweigend, gehen wir davon aus, daß er bei Dienstreisen ins Ausland immer noch gerne Nebengeschäfte tätigt. Jüngst war er wieder in Rio und stand vor der Entscheidung, seinen Zweitkoffer möglichst profitabel mit Schmuggelwaren zu füllen –

ke vielversprechend erscheinen. Einen sehr einfachen logischen Test hat Kuno intuitiv angewendet, als er die Knoten 6 und 10 nicht mehr verzweigt hat. Obwohl sie einen guten Schrankenwert aufweisen, konnte er feststellen, daß kein noch nicht fest eingepackter Gegenstand mehr dazugepackt werden kann – der Koffer ist voll.

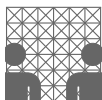
Wenn sich eine Teilaufgabe nicht ausloten läßt, kann man versuchen, eine zulässige Lösung für diese Teilaufgabe zu bestimmen. Dazu verwendet man sogenannte *Heuristiken*, die einer Aufgabe mit einfachen Faustregeln zu Leibe rücken. Wenn man Glück hat, ergibt sich dabei eine neue bisher beste Lösung. Dies hilft beim



en könnte. Masker setzt dieses Vorgehen fort und besucht die Koffer in der Reihenfolge ihrer Nummern. Bei Koffer 6 ergibt sich zwar eine Schranke, die höher als der bisher höchste Gewinn von 35 Dollar ist. Doch Kuno stellt fest, daß zu den bereits eingepackten Gegenständen keiner mehr paßt. Diese Kombination bringt 34 Dollar Gewinn, also keine Verbesserung. Koffer 7 kann er auch ausloten, weil die Schranke kleiner als 35 Dollar ist. Koffer 10 ist durch die eingepackten Gegenstände bereits voll. In Koffer 11 und 12 finden sich beim Beschränken

zulässige Kombinationen. In allen Fällen ist der Gewinn zu gering, und es wird ausgelotet. Erst in Koffer 14 ergibt sich bei Berechnung der Schranke die Kombination *Amulett, Schmetterling, Zigaretten, Kupfer* mit einem Gewinn von 40 Dollar. Diese Kofferbeladung merkt sich Kuno als neue bisher beste. Nach Betrachten von Koffer 15 sind sämtliche Teilaufgaben voll verzweigt oder ausgelotet. Daher weiß Kuno, daß die zuvor ermittelte Lösung mit einem Gewinn von 40 Dollar optimal ist, und beendet völlig erschöpft seine Bemühungen.

© 1997, Heft 10



Zum Abschluß
kommt ihr zu Wort ...



- Waren die Folien lesbar?
- Zu viel / zu wenig Inhalt?
- Kritik an den Aufgaben?
- sonstiges zu kritisieren ...

