DHBW Lörrach
Duale Hochschule Baden-Württemberg

# Algorithmen und Komplexität
# TIF 21A/B

# Dr. Bruno Becker


## 10. Komplexitätstheorie

## 10.1 Die Komplexitätsklassen *P* und *NP*

# Komplexitätstheorie

- **Komplexität von Problemen**
- Die Komplexitätsklasse $\mathcal{P}$
- Die Komplexitätsklasse $\mathcal{NP}$
- Travelling Salesman Problem

# Komplexität von Problemen

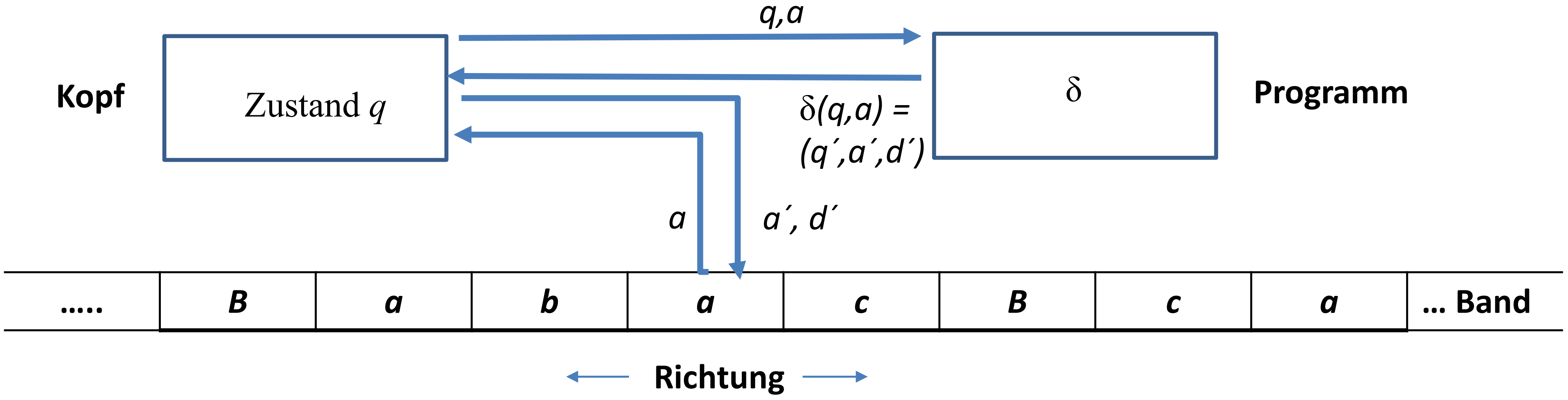
- Vergleich und Klassifikation von Problemen nach *Schwierigkeit einer algorithmischen Lösung*
  - Wie schwierig ist ein gegebenes Problem?
  - Ist *optimaler Algorithmus* hierfür bereits bekannt?
  - Gibt es besonders schwierige Probleme?
- Literaturhinweis für dieses Kapitel:
  - Ingo Wegener, Theoretische Informatik, Teubner-Verlag (Kapitel 2 und 3)
  - Wagner, Pfeiffer-Bohnen, Schmeck, Theoretische Informatik ganz praktisch, De Gruyter-Verlag (Kapitel 7)

# Komplexität von Problemen

- **Komplexität eines Problems:** *Worst-case* Aufwand des *bestmöglichen Algorithmus* zur Lösung des Problems
  - Zeitkomplexität (meistens)
  - Speicherkomplexität
- **Berechnungsmodell**
  - Turingmaschinen
  - Registermaschinen
  - Kostenmaß
  - O-Notation

# Turingmaschine

- *Erfunden von Alan Turing (entschlüsselte die Enigma im 2.Weltkrieg; nach ihm benannt Turing Award und Turing Test)*
- Turingmaschine (*TM*) (1936)
  - Band mit unbegrenzt vielen Feldern; Endliches Bandalphabet
  - Schreibe-Lese-Kopf; Steuerlogik: endlich viele Zustände, Zustandsübergangsfunktion



# Turingmaschine - Definition

**Definition:**

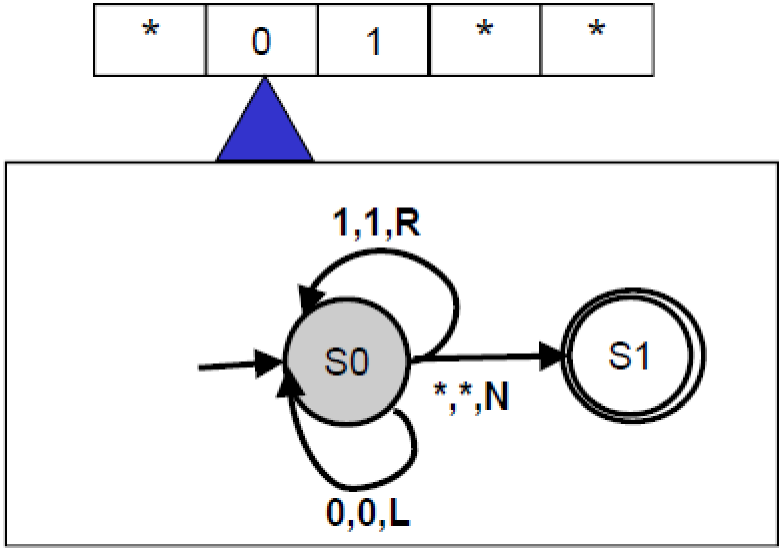
Eine **(deterministische) Turingmaschine** ist ein Tupel T= (E,S, **B**, $\delta$,$s_0$,F) mit

- E= $\{e_1,\ldots, e_r\}$ : Eingabealphabet (Sonderzeichen * ist nicht in E)
- S= $\{s_0,\ldots, s_n\}$ : Zustandsmenge
- **B= $\{b_1,\ldots, k_m\}$ :** **Bandalphabet** (enthält insbesondere **E** und Sonderzeichen * - für unbeschriebene Band-Zelle)
- **$\delta$: S x B→ S x B x {L,R,N} : partielle Überführungsfunktion** (Bandkopf 1 Zelle nach links, rechts oder bleibt)
- $s_0$ : Startzustand
- F $\subseteq$ S : Menge der Endzustände (nichtleer)

# Turingmaschine – Beispiel „Rechts/Links-Maschine“

- E={0,1}, B={0,1,*}, S={$s_0$, $s_1$}, F={$s_1$}
- Übergang:
  - Eingabe 1: S/L-Kopf läuft nach rechts
  - Eingabe 0: S/L-Kopf läuft nach links
  - T bleibt am Ende vom Wort stehen
- w=01

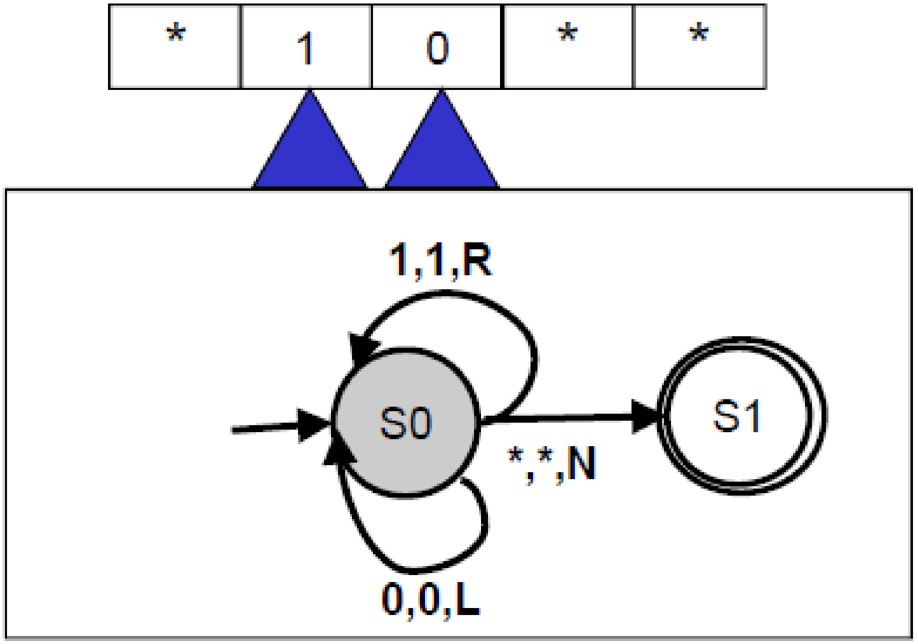
| δ | 0 | 1 | * |
|---|---|---|---|
| $s_0$ | ($s_0$,0,L) | ($s_0$,1,R) | ($s_1$,*,N) |
| $s_1$ | - | - | - |




Beispiel: 01: T bleibt links von w stehen

# Turingmaschine – Beispiel „Rechts/Links-Maschine“

| δ | 0 | 1 | * |
|---|---|---|---|
| $s_0$ | $(s_0,0,L)$ | $(s_0,1,R)$ | $(s_1,*,N)$ |
| $s_1$ | - | - | - |




Beispiel: 10: T bleibt gar nicht stehen
→ Endlosschleife.

# Turingmaschine – Mögliche Situationen

- T hält nie an
- T hält im Endzustand an
- T hält an, ist aber nicht im Endzustand
- T liest Wort w nur teilweise
- T verändert Eingabewort w.

# Halteproblem

- Sei <*M*> ein Programm auf einer Turingmaschine *T* und *w* Eingabe auf *M*
  - Frage: Hält die Turingmaschine *M* auf Eingabe *w* an? (d.h. gerät nicht in Endlosschleife)
  - **$H := \{\langle M\rangle w \mid M \text{ hält auf } w\}$ ist nicht entscheidbar.**

Übersetzt auf allgemeines Maschinenmodell:
Kann man einen Algorithmus entwickeln, mit dem man testen kann, ob ein (in geeigneter Weise codierter) übergebener Algorithmus bei der Verarbeitung übergebener Daten hält oder nicht?

# Bedeutung Turingmaschine

- **Reduktion auf ein sehr einfaches Maschinenmodell**
  - Ermöglicht Einsatz mathematischer Methoden zur Analyse der Berechenbarkeit
  - Trotz der Einfachheit ist prinzipiell „jede Rechenmaschine“ auf Turingmaschine abbildbar
- **Churchsche These:**

**„Die durch die formale Definition der Turing-Berechenbarkeit erfasste Klasse von Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein“**

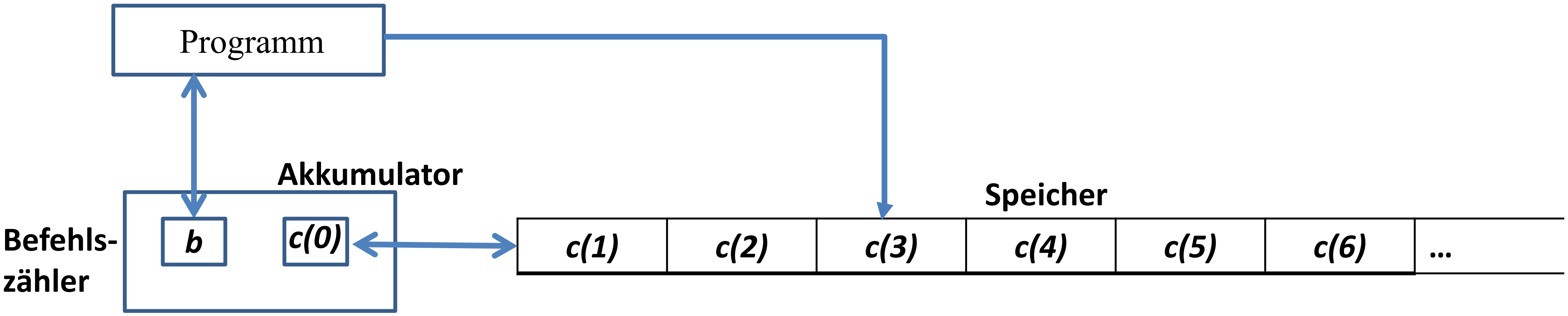
- **Klartext:** *Jeder Algorithmus ist durch eine geeignete Turingmaschine darstellbar.*
  *→ Jede Aussage über Berechenbarkeit auf Turingmaschine gilt auch für jedes andere Maschinenmodell*

→ *Es gibt daher kein Super-Algorithmus, der entscheiden kann, ob gegebenes Programm mit konkretem Input hält oder nicht.*

→ *Das liegt nicht am Unvermögen der Informatiker, sondern an den Grenzen der algorithmischen Problemlösemethode.*

# Registermaschine

- **Registermaschine (*Random Access Maschine, RAM*)** ist Abstraktion von CPU:
  - Eingabeband (read-only), Ausgabeband (write-only)
  - Unbegrenzte Anzahl *Register (Speicherzellen)*, beliebig große ganze Zahlen
- **Einfache CPU-Befehle:**
  - *LOAD i: c(0)=c(i), STORE i c(i)=c(0), CLOAD, CSTORE mit Konstante i statt c(i)*
  - *ADD i: c(0)=c(0)+ c(i), SUB i, MULT i, DIV i, CADD,…,*
  - *INDLOAD,… Indirekte Adressierung c(i) durch c(c(i)) ersetzen (Pointer)*
  - *GO TO j : b= j, IF c(0) <,≤, >,≥, = GO TO*



# Bedeutung Registermaschine

- **Reduktion auf ein einfaches Maschinenmodell**
  - Ist näher dran an realen Programmen als Turing-Maschinen
  - Jeder Algorithmus und jedes Programm lässt sich auf Registermaschinen übertragen – Anzahl Rechenschritte unterscheidet sich nur um konstanten Faktor gegenüber konkretem Rechnermodell → Für O-Notation unerheblich
- **Logarithmisches Kostenmass**
  - Zeit und Platz proportional zur Länge der Operanden in Binärcode

# Komplexitätstheorie

- Komplexität von Problemen
- **Die Komplexitätsklasse $\mathcal{P}$**
- Die Komplexitätsklasse $\mathcal{NP}$
- Travelling Salesman Problem

# Die Komplexitätsklasse $\mathcal{P}$

**Polynomialzeit-Algorithmus:** Algorithmus mit worst-case Laufzeit $O(n^k)$ auf Inputs der Länge $n$ für ein festes $k$

- Maschinenmodell TM oder RAM mit *logarithmisches Kostenmaß*
- Abstraktion für *machbare/effiziente* Algorithmen
- Im Gegensatz zu *exponentieller Laufzeit* $(\Omega(2^n))$

**Polynomialzeit ist unabhängig vom Berechnungsmodell**

- TM und RAM können sich gegenseitig mit *polynomialen Zeitverlust* simulieren
- Gilt auch zwischen RAM und modernen Sprachen und Rechnermodellen
- Aufwand steigt also höchstens um Faktor $n^c$

# Die Komplexitätsklasse $\mathcal{P}$

**Polynomialzeit = machbar/effizient??** z.B. *$1000 * n^3$, $10^9 * n^3$, $n^{50}$*

*Ja, denn:*

- „skalierbare" Lösung
- Passt zur Intuition
- Erfahrungswert: Wenn man überhaupt einen *Polynomialzeit*-Algorithmus findet,  gibt es auch einen mit praktikabler Potenz von n und nicht zu hohen Konstanten

# Entscheidungsprobleme

**Input:** Binärer Input – Folge von *n* Bits

**Output:** JA / NEIN (TRUE / FALSE 1 / 0)

- Ein **Entscheidungsproblem** ist durch eine Menge von JA-Instanzen eindeutig definiert, d.h. diese Inputs werden *akzeptiert*
- **Entscheidungsproblem:** Menge *M* = $\{0,1\}^*$ (Alphabet : {0,1}, jedes Wort aus *M* besteht aus Buchstaben aus diesem Alphabet z.B: 0111011110101 )

Warum betrachtet man in Komplexitätstheorie gerne Entscheidungsprobleme:

- Outputgröße kein Faktor
- Einfacher zu handhaben
- Oft Verwandschaft zwischen Entscheidungsproblem und Nicht-Entscheidungsproblem

# Die Komplexitätsklasse $\mathcal{P}$

$\mathcal{P}$ ist die Klasse der Entscheidungsprobleme, für die es einen Polynomialzeit-Algorithmus gibt.

Beispiele von Problemen in $\mathcal{P}$ :

- Ist ein gegebenes Element in einer Liste enthalten?
- $a+b=c$ ?
- Hat ein gewichteter Graph $G$ einen *MST* mit Gewicht $\leq w$ ?
- Ist das lineare Gleichungssystem $Ax = b$ lösbar?

# Probleme außerhalb von $\mathcal{P}$

## Spezielles Halteproblem:

- **Input:** Eine Turingmaschine *M*, ein Input *w* und eine Zahl *k*
- **Output:** Hält *M* auf Input *w* nach höchstens *k* Schritten an?

- Das spezielle Halteproblem ist berechenbar – Im Gegensatz zum allgemeinen Halteproblem
- Aber nicht in Polynomialzeit lösbar: Im Worstcase *k* Schritte

# Komplexitätstheorie

- Komplexität von Problemen
- Die Komplexitätsklasse $\mathcal{P}$
- **Die Komplexitätsklasse $\mathcal{NP}$**
- Travelling Salesman Problem

# Lösung eines Problems verifizieren

## Beispiel: Zahlenkombination eines Schlosses *n* Stellen knacken

- Lösung *finden: 10* $^{n}$ Möglichkeiten → Exponentieller Aufwand
- Lösung *verifizieren*: Einstellen, ausprobieren → *O(n)*

Typisch für *kombinatorische Probleme*: Lösung ist schwer zu finden aber einfach zu verifizieren

# Nichtdeterministische Algorithmen

**Nichtdeterministischer Algorithmus:**

- ***rät*** einen möglichen Beweis (*Zertifikat)* dafür, dass ein gegebener Input *w* eine JA-Instanz des Problems ist (Zertifikat ist Bitfolge abhängig vom Input)
- ***verifiziert*** das Zertifikat mit deterministischen Algorithmus

  → Output JA: Zertifikat beweist, dass *w* JA-Instanz ist

  → Output NEIN: Ungültiges Zertifikat (beweist nichts)

**Der Algorithmus arbeitet korrekt, wenn**:

- Für jede JA-Instanz *w* gibt es *mindestens ein* akzeptiertes Zertifikat
- Für jede NEIN-Instanz w gibt es *kein* akzeptiertes Zertifikat

# Die Komplexitätsklasse $\mathcal{NP}$

$\mathcal{NP}$ ist die Klasse der Entscheidungsprobleme, für die es einen *nichtdeterministischen* Polynomialzeit-Algorithmus gibt.

- Ein Problem ist als in $\mathcal{NP}$, wenn alle JA-Instanzen in Polynomialzeit verifizierbare Beweise (Zertifikate) haben
- Assymetrische Definition: Keine Aussage über NEIN-Instanzen

Formale Behandlung über *nichtdeterministische Turingmaschine:*

- Aktueller Zustand und aktuelles Bandsymbol legen Übergang nicht fest, d.h. keine Übergangsfunktion, sondern *Übergangsrelation*

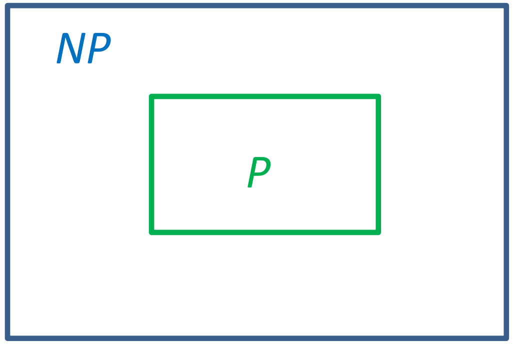
# Beispiele für Probleme in $\mathcal{NP}$

Viele kombinatorische Such-/Optimierungsprobleme liegen in $\mathcal{NP}$

- Ist eine gegebene Zahl eine Primzahl?
- SAT: Ist eine in Klauselform gegebene aussagenlogische Formel erfüllbar?
- In einem Graphen: Gibt es einen Zyklus, der jeden Knoten genau einmal durchläuft?

# Ist $\mathcal{P} = \mathcal{NP}$ ?

**Vermutung:**




Klar ist: $\mathcal{P} \subset \mathcal{NP}$.

- Wichtigstes offenes Problem der Theoretischen Informatik !
- Falls $\mathcal{P} = \mathcal{NP}$, hätte das gravierende Auswirkungen

# Reduktion von Problemen

Eine **Reduktion von Problem *A* auf Problem *B*** ist eine Funktion *f*, für die gilt:

- Es gibt einen Polynomialzeit-Algorithmus zur Berechnung von *f*
- Für alle Inputs *w* gilt: $w \in A \Leftrightarrow f(w) \in B$

Das heißt: ***Wenn wir einen Algorithmus zur Berechnung von B haben, können wir damit auch A lösen:***

1. Berechne für Input *w* den Wert *f(w)*
2. Löse *B* für *f(w)*

**Reduktion ist sehr wichtige Methode in der Komplexitätstheorie!**

# Reduktion von Problemen (2)

Wenn sich Problem *A* auf Problem *B* reduzieren lässt, dann schreibt man

$\boldsymbol{A \leq B}$ **:** *A ist (bis auf polynomiale Faktoren) höchstens so schwierig wie B*

Aus $\boldsymbol{A \leq B}$ folgt:

- Falls *B leicht* ist → *A* ist auch *leicht*
- Falls *A schwierig* ist → *B* ist auch *schwierig*

# $\mathcal{NP}$-harte und $\mathcal{NP}$-vollständige Probleme

Ein Entscheidungsproblem $B$ heißt ***NP-hart,*** wenn für jedes $A \in$ ***NP*** gilt:

$$A \leq B$$

Das bedeutet: ***Jedes Problem aus NP lässt sich auf B reduzieren!***

Ein Entscheidungsproblem $B$ heißt ***NP-vollständig,*** wenn es:

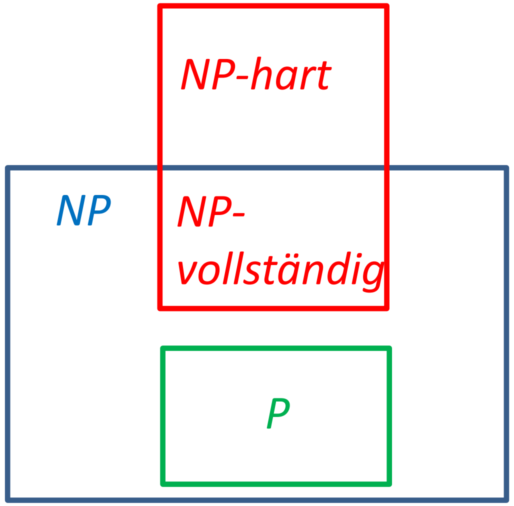
- ***NP-hart*** ist, und
- selbst in ***NP*** liegt.

# Bedeutung $\mathcal{NP}$-vollständiger Probleme

***NP*-vollständige Probleme sind die *härtesten* Probleme in *NP-hart:***

- Wenn ***ein einziges NP-***vollständiges Problem in ***P*** liegt, folgt ***P=NP***
- Wenn ***ein einziges NP-***vollständiges Problem ***nicht*** in ***P*** liegt, gilt das für **alle *NP-***vollständiges Probleme (d.h. die Lösung benötigt exponentielle Zeit)

**Vermutung:**



# Satz von Cook (1971)

**SAT (Erfüllbarkeitsproblem der Aussagenlogik) ist *NP*-vollständig**

*SAT-Problem (von satisfiability): Ist eine in Klauselform gegebene aussagenlogische Formel erfüllbar?*

**Beweis-Idee:**

- SAT $\in$ ***NP*** : Zertifikat = erfüllende Interpretation raten, in Formel einsetzen und auswerten.
- SAT ist ***NP***-hart: Gegeben ein Problem $A \in$ ***NP*** und eine nichtdeterministische TM hierfür mit polynomialer Laufzeit.
  - Spezifiziere Verhalten der TM in jedem Rechenschritt als große aussagenlogische Formel (mit Bandinhalt, Zustand und Zustandsübergängen)
  - Länge der resultierenden Formel ist polynomial in Inputlänge, da TM polynomiale Laufzeit hat
  - Formel erfüllbar $\Leftrightarrow$ es gibt ein akzeptiertes Zertifikat

# Beweis der *NP*-Vollständigkeit

Beweis des Satzes von Cook ist kompliziert, aber der Beweis der ***NP***-Vollständigkeit für ein erstes Problem erleichtert den Nachweis für andere Probleme, denn:

**Sei $L_2 \in NP$ und $L_1 \leq L_2$ für *ein NP-vollständiges* Problem $L_1$.**
**Dann ist $L_2$ *NP-vollständig*.**

Beweis: Es ist $L_2 \in$ ***NP.*** Sei $L' \in$ ***NP*** . Da $L_1$ ***NP***-vollständig ist, gilt $L' \leq L_1$.

Nach Voraussetzung ist $L_1 \leq L_2$. Aus der Transitivität von „ $\leq$“ folgt $L' \leq L_2$

Da $L'$ beliebig aus ***NP*** gewählt, gilt das für jedes Problem aus ***NP.***

# Weitere *NP*-vollständige Probleme

Weitere *NP*-vollständige Probleme

- **3-SAT:** Wie SAT, aber Klauselform mit höchstens 3 Literalen je Klausel
- **Hamilton-Zyklus:** Gegeben Graph. Gibt es Zyklus, der jeden Knoten genau einmal durchläuft?
- **Knappsack-Problem:** Gegeben ein Rucksack und *n* Objekte mit Gewichten und Nutzenwerten. Gibt es zu gegebenem Nutzenwert *A* eine Bepackung des Rucksacks, die Gewichtslimit respektiert und mindestens Nutzenwert *A* liefert?
- https://en.wikipedia.org/wiki/List_of_NP-complete_problems

# Umgang mit $\mathcal{NP}$-harten Problemen

Was tun bei ***NP-***harten Problemen in der Praxis?

- Exponentielle Worst-Case-Laufzeit akzeptieren (z.B. Backtracking)
- Probleme einschränken auf „gutartige“ Fälle, so dass Problem in ***P*** liegt
- Heuristiken/Näherungslösungen verwenden, die in der Praxis gut funktionieren
- Randomisierte Algorithmen: Verwendung von Zufallszahlen, um eine akzeptable *erwartete Laufzeit* zu erreichen

# Komplexitätstheorie

- Komplexität von Problemen
- Die Komplexitätsklasse $\mathcal{P}$
- Die Komplexitätsklasse $\mathcal{NP}$
- **Travelling Salesman Problem (-> in 10.2)**