Automatentheorie und Formale Sprachen

– LV 4110 –

Problemklassen und Komplexitätstheorie

- Effizienz und Laufzeit von Programmen
- Herausstellen von Komplexitätsklassen
- Laufzeitkomplexität und Problemgröße
- Definitionen der Klassen P und NP
- Erläuterung und Einordnung des Knapsack-Problems
- Definition und Interpretation der NP-Vollständigkeit
- Erläuterung und Einordnung des Erfüllbarkeitsproblems (SAT)
- Lösungsalgorithmen für NP-harte Probleme

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

VII. Problemklassen und Komplexitätstheorie

Gibt es einen Effizienzgewinn
beim Übergang zu
„mächtigeren“ Rechenmodellen als TM oder RAM

bzw.

wie effizient kann ein Problem
grundsätzlich gelöst werden ?

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. **Herausstellen von Komplexitätsklassen**
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

Hintergründe:

- Wir haben **Entscheidungsprobleme**, **Berechenbarkeitsprobleme**, **Approximationsprobleme**, **Erfüllbarkeitsprobleme**, **Aufzählungs-probleme**, **Suchprobleme** bis hin zu **Optimierungsproblemen** zu lösen.
- Dazu stehen uns **Erkennungsprozeduren**, **Such- und Finde-prozeduren**, **Prüf- und Entscheidungsprozeduren** sowie **Verarbeitungsprozeduren** zur Verfügung.
- In diesem Zusammenhang interessieren wir uns im allgemeinen für die **Laufzeit von Algorithmen** (Berechnungen) bzw. für die **„Größe“ von Problemen**.

Fragestellungen:

- In welchem Zusammenhang stehen Sprachen und Probleme? Gibt es überhaupt einen Zusammenhang?
- Gibt es eine Korrespondenz zwischen Entscheidungs-, Berechenbarkeits- und Optimierungsproblemen?
- Gibt es womöglich unterschiedliche Problemklassen?
- Gibt es für jedes Problem einen Lösungsalgorithmus bzw. existiert zu jedem Problem überhaupt eine Lösung?
- Gibt es ein Beweissystem (z. B. aus Axiomen und Schlussregeln), mit dem man feststellen kann, ob ein Problem algorithmisch (un)lösbar ist?

Hinter diesen Fragestellungen verbirgt sich eine der wichtigsten offenen Fragen der theoretischen Informatik:

Ist **P ≠ NP?**

Mit anderen Worten:

- Besitzen deterministische Turing-Maschinen eine andere **Zeitkomplexität** als nicht deterministische Turing-Maschinen?
- Erzielen nicht deterministische Turing-Maschinen eine höhere **Aussagekraft** als deterministische Turing-Maschinen?

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. **Laufzeitkomplexität und O-Notation**
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

Hintergrund:

Ziel ist es, einen möglichst **effizienten Algorithmus** für ein bestimmtes Problem zu finden.

Definition (Rechenzeit):

Sei **TM** eine deterministische Turingmaschine auf dem Eingabealphabet Σ. Dann ist die worst case Rechenzeit $\mathbf{t_{TM}(n)}$ die maximale Anzahl von Rechenschritten, die **TM** auf Eingaben aus Σ^n macht.

Definition (Zeitkomplexitätsfunktion $T_{TM}(n) : \mathbb{N} \rightarrow \mathbb{N}$):

$\mathbf{T_{TM}(n) = \max\{m\}}$, so dass eine deterministische Turing-Maschine **TM** bei Eingabe $\mathbf{x \in \Sigma^n}$ **m** Berechnungsschritte (Übergänge) benötigt, bis ein Endzustand erreicht wird.

Definition (O-Notation):

Für eine Funktion $f : IN \rightarrow IN$ definieren wir

$$f(n) = O(g(n)) = \{\, f : IN \rightarrow IN \mid \exists\, c, n_0 : \forall\, n \geq n_0 : f(n) \leq c \cdot g(n) \,\}$$

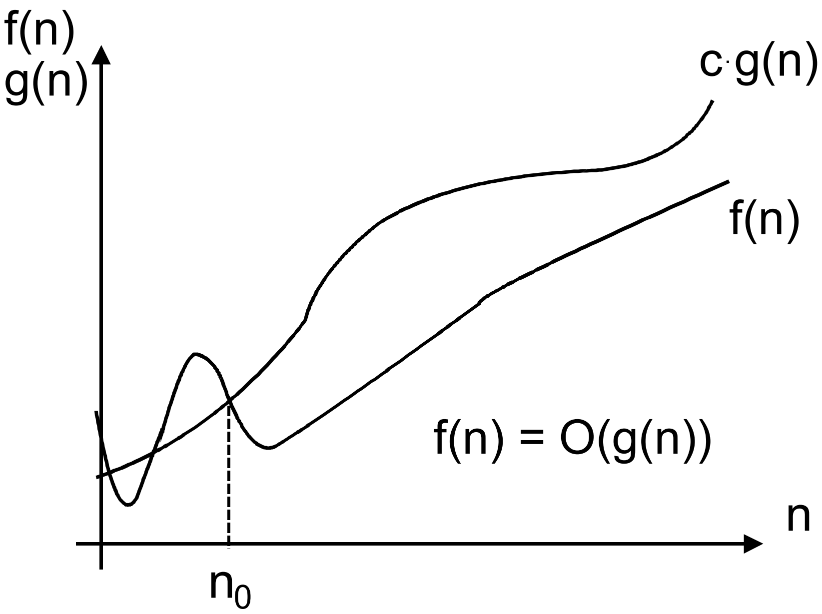
und können so die Laufzeitfunktion f(n) durch eine asymptotische obere Schranke g(n) begrenzen $\Rightarrow$ in Worten: f(n) wächst nicht schneller als g(n)

Die **Komplexität** eines Algorithmus ist O(g(n)), wenn die Laufzeit $T_{TM}(n)$ in O(g(n)) ist, beispielsweise geschrieben als: $f(n) = 2 \cdot n^2 + 4 \cdot n \in O(n^2)$

Dabei bezeichnen:

f(n), g(n)	Zeitkomplexitätsfunktionen (positiv)
n	Eingabelänge
c, n_0	positive Konstanten

Verläufe und Beispiele:



Fehlerterm bei der Approximation:

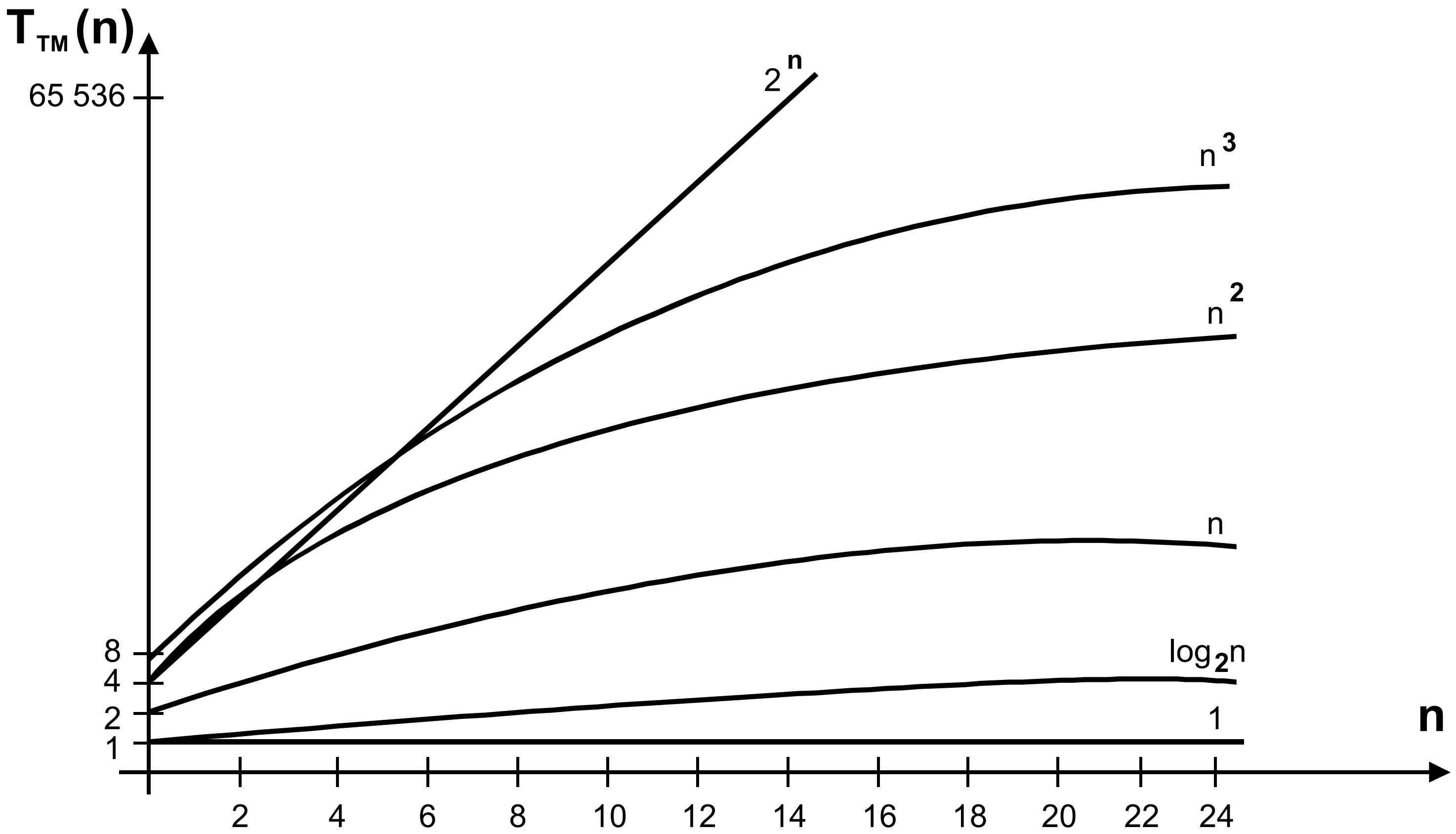
$e^x = 1 + x + x^2 / 2 + O(x^3)$ für $\forall\, x \to 0$ drückt aus, dass die vernachlässigten Summanden höherer Ordnung kleiner sind als der konstante Wert x^3, wenn x nur klein genug ist.

CPU Time eines Algorithmus:

$$T_{TM}(n) = f(n) = 10 \log(n) + 5 (\log(n))^3 + 7 n + 3 n^2 + 6 n^3$$

$\Rightarrow f(n) = O(n^3)$

Verläufe (Wachstumsraten):



Beispiel:

Ein Beispiel für die Größenordnung bzw. Hierarchie ist die folgende Anordnung:

1	$\log n$	$n \quad n^2 \quad n^3 \quad \ldots \quad n^k$	2^n
↑	↑	↑	↑
konstant	logarithmisch	polynomiell	exponentiell

Satz:

Sei **k** eine Konstante und **p** ein Polynom. Ein Algorithmus ist **polynomiell**, wenn seine Zeitkomplexität $T_{TM}(n) \in O(n^k)$ ist. Ein Algorithmus ist dagegen **exponentiell**, wenn seine Laufzeit $T_{TM}(n) \in O(2^{p(n)})$ ist.

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. **Definitionen der Klassen P und NP**
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

Definition (Klasse P):

P ist eine Klasse der Probleme, für die es eine deterministische Turing-maschine **TM** gibt, deren worst case Rechenzeit $\mathbf{t_{TM}(n)}$ polynomiell beschränkt ist, d. h. es existiert ein Polynom **p** mit

$$\mathbf{T_{TM}(n) \leq p(n)}$$

Definition (Klasse NP):

NP (nicht deterministisch polynomiell) ist die Klasse der Entscheidungs-probleme, für die es eine nicht deterministische Turingmaschine **NTM** gibt, deren worst case Rechenzeit $\mathbf{t_{NTM}(n)}$ polynomiell beschränkt ist, d. h.

$$\mathbf{T_{NTM}(n) \leq p(n)}$$

<u>Beispiel</u>:

Das Entscheidungsproblem **PRIMES** besteht darin, zu entscheiden, ob es sich bei einer gegebenen natürlichen Zahl $\mathbf{z} > 1$ um eine Primzahl handelt. Dabei sei die Zahl $\mathbf{z}$ zur Basis $\mathbf{b} \in \mathbf{IN}$ dargestellt.

Die dazugehörige Sprache sei mit $\mathbf{L_b}$ = **L [PRIMES, b]** bezeichnet.

<u>Satz</u>:

Sei $\mathbf{L_1}$ **:= L [PRIMES, 1]**. Erst 2002[1)] konnte gezeigt werden, dass gilt:

$\mathbf{L_1}$ **liegt in P**

d. h. es gibt eine DTM, deren Laufzeit von der Ordnung $\mathbf{O(n^3)}$ und damit polynomial beschränkt ist.

1) Drei indische Mathematiker: M. Agrawal, N. Kayal und N. Saxena

Beispiel:

Ein bedeutender Algorithmus ist der **Euklidische Algorithmus** zur Berechnung des größten gemeinsamen Teilers (kurz **ggT**(n, m)).

Satz:

Der Euklidische Algorithmus zur Berechnung des **ggT** zweier natürlicher Zahlen n und m ist **polynomial** und damit **effektiv berechenbar**.

Genauer gesagt gilt für seine Laufzeit:

$$T(n, m) = O\left((\log_2 n + \log_2 m)^2\right)$$

Beispiel:

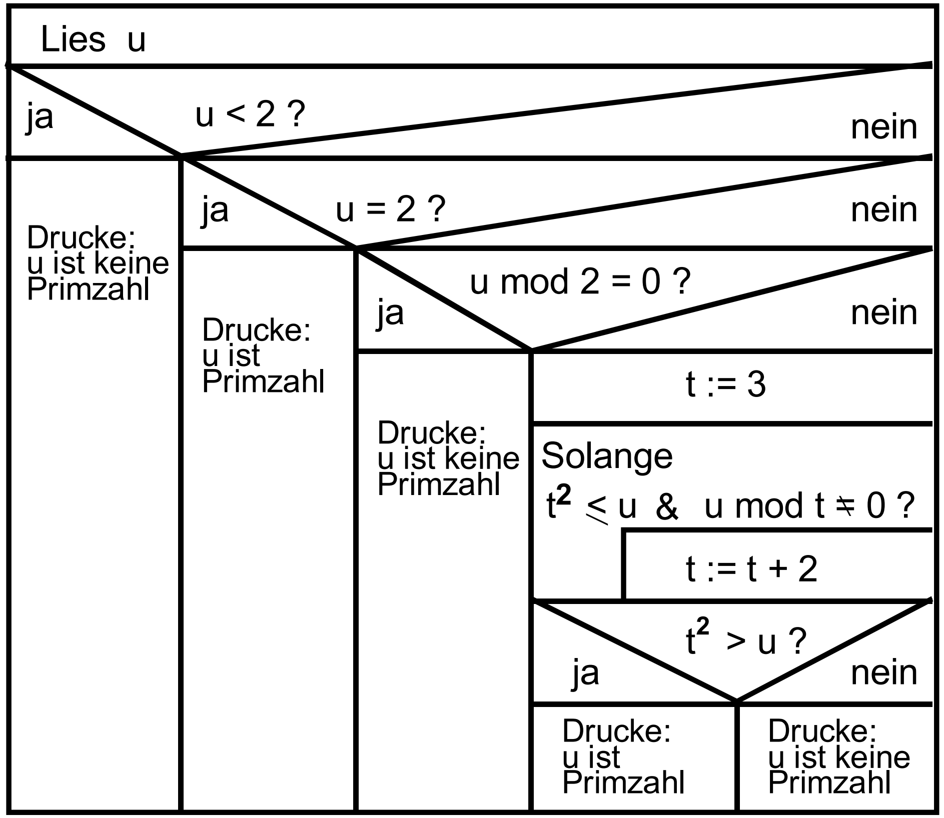
Liefert ein Entscheidungsalgorithmus zu der Frage, ob n zusammengesetzt (also beispielsweise $n = p \cdot q$) ist, als Beleg der Antwort „JA“ einen Faktor q von n, so lässt sich in polynomialer Zeit nachprüfen, ob $q \mid n$.

Es gilt nämlich: $q \mid n \quad \text{gdw} \quad (\exists\, k \in \mathbf{Z}) : n = k \cdot q \quad \Leftrightarrow \quad \text{ggT}(q, n) = |\, q \,|$

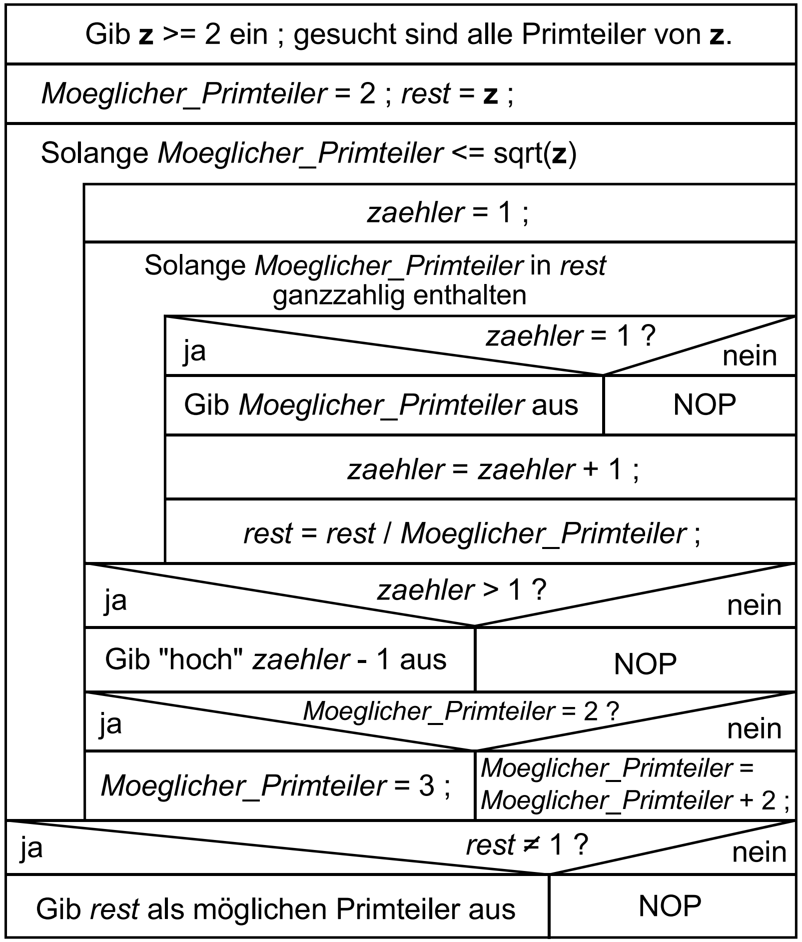
Das **Faktorisierungsproblem** (d. h. die Zerlegung von n in ihre Primfaktoren) ist also in **NP**, aber man weiß nicht, ob es in **P** ist.

Satz:

Die Klasse **NP** besteht aus denjenigen Entscheidungsalgorithmen, bei denen es einen Algorithmus gibt, der im Falle einer **JA**-Antwort durch den Algorithmus des Entscheidungsproblems die Korrektheit der Antwort in polynomialer Zeit testet (d. h. Antwort plus Beleg).



Lies u
u < 2 ?
ja
nein
Drucke: u ist keine Primzahl
u = 2 ?
ja
nein
Drucke: u ist Primzahl
u mod 2 = 0 ?
ja
nein
Drucke: u ist keine Primzahl
t := 3
Solange t² ≤ u & u mod t ≠ 0 ?
t := t + 2
t² > u ?
ja
nein
Drucke: u ist Primzahl
Drucke: u ist keine Primzahl



Gib z >= 2 ein ; gesucht sind alle Primteiler von z.
Moeglicher_Primteiler = 2 ; rest = z ;
Solange Moeglicher_Primteiler <= sqrt(z)
zaehler = 1 ;
Solange Moeglicher_Primteiler in rest ganzzahlig enthalten
zaehler = 1 ?
ja
nein
Gib Moeglicher_Primteiler aus
NOP
zaehler = zaehler + 1 ;
rest = rest / Moeglicher_Primteiler ;
zaehler > 1 ?
ja
nein
Gib "hoch" zaehler - 1 aus
NOP
Moeglicher_Primteiler = 2 ?
ja
nein
Moeglicher_Primteiler = 3 ;
Moeglicher_Primteiler = Moeglicher_Primteiler + 2 ;
rest ≠ 1 ?
ja
nein
Gib rest als möglichen Primteiler aus
NOP

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. **Erläuterung und Einordnung des Knapsack-Problems**
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

Definition (KNAPSACK **KP**):

Gegeben sei ein Rucksack und **n** Objekte mit Gewichten $g_1, g_2, ..., g_n \in \mathbf{IN}$ sowie eine Gewichtsschranke **G**. Zusätzlich seien $a_1, a_2, ..., a_n \in \mathbf{IN}$ die Nutzenwerte für die Objekte. Bei $x_i = 1$ wird ein Objekt i eingepackt und bei $x_i = 0$ nicht.

Variante 1: Gibt es zu einem gegebenen Nutzenwert **A** eine Bepackung des Rucksackes, die das Gewichtslimit **G** respektiert und mindestens den Nutzen **A** erreicht?

Also ob: $a_1 \cdot x_1 + a_2 \cdot x_2 + ... + a_n \cdot x_n \geq \mathbf{A}$

unter der Nebenbedingung: $g_1 \cdot x_1 + g_2 \cdot x_2 + ... + g_n \cdot x_n \leq \mathbf{G}$

Variante 2: Berechne den **größten erreichbaren Nutzen A**!

Variante 3: Berechne eine **optimale Bepackung** des Rucksackes!

<u>Satz:</u>

Die Entscheidungsvarianten 1 bis 3 von KP sind in **NP**.

<u>Satz:</u>

Trivialerweise gilt:

$$\mathbf{P} \subseteq \mathbf{NP}$$

<u>Satz:</u>

Für jede Sprache **L** $\in$ **NP** gibt es ein Polynom **p** und eine **deterministische** Turingmaschine **TM**, so dass **TM** die Sprache **L** in

exponentieller Zeit $2^{p(n)}$ akzeptiert.

Dabei ist:

n = Länge der Eingabesymbole (aus Σ^n)

Satz:

KP ist NP-vollständig[1)].

1) also mindestens so komplex, wie jedes andere Problem in **NP**.

Algorithmus:

Die Lösung des KP-Problems führt auf die sog. Bellmansche Optimalitäts-gleichung.

Satz:

Das Rucksack-Problem (KP) kann in der Zeit (→ pseudopolynomiell)

O(nG)

gelöst werden (**n** = Anzahl der Objekte, **G** = Gewichtsschranke).

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. **NP-Vollständigkeit**
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

Definition (Polynomiell reduzierbar):

Es seien L_1 und L_2 Sprachen über Σ_1 bzw. Σ_2. Dann heißt L_1 polynomiell auf L_2 reduzierbar,

Notation: $\mathbf{L_1} \leq_p \mathbf{L_2}$ (**p** steht für **polynom**iell)

wenn es eine polynomielle Transformation von L_1 nach L_2 gibt, d. h. wenn es eine von einer deterministischen Turingmaschine **TM** in polynomieller Zeit **berechenbare** Funktion

$$\mathbf{f} : \Sigma^*_1 \rightarrow \Sigma^*_2$$

gibt, so dass für alle $w \in \Sigma^*_1$ gilt:

$$w \in \mathbf{L}_1 \Leftrightarrow \mathbf{f}(w) \in \mathbf{L}_2$$

Beispiel:

Seien $\mathbf{L_1}$ die Sprache der ungeraden Zahlen in Dezimaldarstellung über $\Sigma_1 = \{0, 1, ..., 9\}$ und $\mathbf{L_2}$ die Sprache der Wörter gerader Länge über dem Alphabet $\Sigma_2 = \{a, b\}$.

Dann kann $\mathbf{L_1}$ polynomial in $\mathbf{L_2}$ transformiert werden, d. h. $\mathbf{L_1} \leq_p \mathbf{L_2}$.

Beweisidee:

Sei $f : \Sigma_1^* \rightarrow \Sigma_2^*$ definiert als

$$f(z) := \begin{cases} \mathbf{aa}, & \text{falls } z \text{ ungerade} \\ \mathbf{a}, & \text{sonst} \end{cases}$$

Dann kann f von einer DTM berechnet werden.

Fortsetzung der Beweisidee:

Eine solche DTM erhält als Eingabe eine Dezimalzahl z. Zunächst löscht sie alle Zeichen der Eingabe bis auf das Letzte. Wenn das letzte Zeichen der Eingabe ungerade ist, schreibt sie **aa**, ansonsten **a** auf das Band. Die Laufzeit der DTM ist hierbei im Wesentlichen **logarithmisch** in der Größe der Eingabe. Damit ist die Laufzeit polynomial beschränkt.

Sei $z \in \Sigma_1^*$. Dann ist $f(z) \in \mathbf{L_2} \Leftrightarrow f(z) = aa \Leftrightarrow$ z ist ungerade $\Leftrightarrow z \in \mathbf{L_1}$.

Damit ist f eine polynomiale Transformation von $\mathbf{L_1}$ in $\mathbf{L_2}$.

Definition (NP-Vollständigkeit):

Eine Sprache **L** heißt NP-vollständig, wenn **L** $\in$ **NP** ist und für alle **L'** $\in$ **NP** gilt:

$$L' \leq_p L$$

Interpretation:

L ist also **NP**-vollständig, wenn **L** selber zu **NP** gehört und jedes Problem in **NP** bzgl. $\leq_p$ nicht schwieriger als **L** ist. **NP**-vollständige Probleme müssen also selber in **NP** enthalten sein.

$$\textbf{NP}\text{-vollständig} \subseteq \textbf{NP}$$

NP-vollständige Probleme sind also **mindestens so komplex**, wie jedes andere Problem in **NP**.

VII. Problemklassen und Komplexitätstheorie

Bemerkungen:

Falls irgendwann jemand einen polynomiellen Lösungsalgorithmus für ein **NP**-vollständiges Problem **L** finden würde, dann hätte man einen polynomiellen Lösungsalgorithmus für **jedes** Problem in **NP**.

Es würde dann gelten, dass

P = NP

(eines der z. Z. größte offene Problem der theoretischen Informatik!)

Obwohl man bisher noch **nicht** nachweisen konnte, dass es keinen polynomiellen Lösungsalgorithmus für **NP**-vollständige Probleme gibt, konnte man für eine Reihe wichtiger Probleme zeigen, dass sie **NP**-vollständig sind.

Das bekannteste Beispiel ist das **Erfüllbarkeitsproblem**, kurz **SAT**.
(**SAT** = **Sat**isfiability)

Definition (SAT):

Sei **X** = { $x_1, x_2, ..., x_m$ } eine Menge von **booleschen** Variablen (x_i und $\bar{x}_i$ heißen auch **Literale**). Eine **Wahrheitsbelegung** von **X** ist eine Funktion

$$\mathbf{f} : \mathbf{X} \rightarrow \{ \text{wahr}, \text{falsch} \}.$$

Eine **Klausel** k ist ein **boolescher** Ausdruck der Form

$$y_{k1} \vee y_{k2} \vee \dots \vee y_{ks}$$

(d. h. Disjunktion von Literalen) mit

$$y_{ki} \in \{ x_1, x_2, ..., x_m \} \cup \{ \bar{x}_1, \bar{x}_2, ..., \bar{x}_m \} \cup \{ \text{wahr}, \text{falsch} \}.$$

Dann ist **SAT** wie folgt definiert:

Existiert eine **Wahrheitsbelegung** von **X**, so dass **alle Klauseln** (k = 1, 2, ... n) den Wahrheitswert **wahr** annehmen?

Beispiel:

Gegeben seien eine Menge von Variablen **X** = { x_1, x_2 } und eine Menge von Klauseln **C** = { c_1, c_2 } über **X** gegeben mit

$$c_1 = x_1 \vee \overline{x}_2 \quad \text{und} \quad c_2 = \overline{x}_1 \vee x_2 \ .$$

Mit der Wahrheitsbelegung

$$f(x_1) = f(x_2) = \text{wahr}$$

(d. h. $x_1 = x_2 = 1$) wird **C** erfüllt.

Satz (von Steven Cook, 1971):

SAT ist **NP**-vollständig.

Definition (3SAT):

Gegeben: Eine Menge **X** von Variablen und eine Menge **C** von Klauseln **C**, wobei jede Klausel **genau drei Literale** enthält.

Frage: Existiert eine erfüllende Wahrheitsbelegung für **C**?

Satz:

Das Problem **3SAT** ist **NP**-vollständig.

Anmerkung:

Um zu zeigen, dass ein Problem **NP**-vollständig ist, wird häufig **3SAT** auf das Problem reduziert.

Beispiel (3SAT):

Gegeben seien eine Menge von Variablen **X** = { x_1, x_2, x_3 } und eine Menge von Klauseln **C** = { c_1, c_2 , c_3, c_4 } über **X**, wobei m = | **X** | = 3 und n =| **C** | = 4.

$$c_1 = x_1 \vee x_2 \vee x_3$$

$$c_2 = x_1 \vee \overline{x}_2 \vee x_3$$

$$c_3 = x_1 \vee \overline{x}_2 \vee \overline{x}_3$$

$$c_4 = \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3$$

Mit der Wahrheitsbelegung

$$f(x_1) = f(x_2) = \text{wahr} \;\text{ und }\; f(x_3) = \text{falsch},$$

d. h. (x_1 , x_2 , x_3) = (1 , 1 , 0), wird **C** erfüllt.

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. **NP-harte Probleme**
9. Problemstellung der PARTITION

Vorbemerkung:

Wenn auch bislang der Versuch eines Beweises für **N = NP** nicht gelungen ist, so konnte man in der Vergangenheit für manche Probleme jedoch schon zeigen, dass man damit jedes andere Problem in **NP** lösen kann. Offensichtlich gelang es dabei aber noch nicht zu zeigen, dass diese Probleme selbst auch in **NP** sind.

Für diese Klasse von Problemen führte man die Bezeichnung **NP** hart ein.

Definition (NP hart):

Eine Sprache **L** heißt NP-hart, wenn für alle **L'** $\in$ **NP** gilt:

$$L' \leq_p L$$

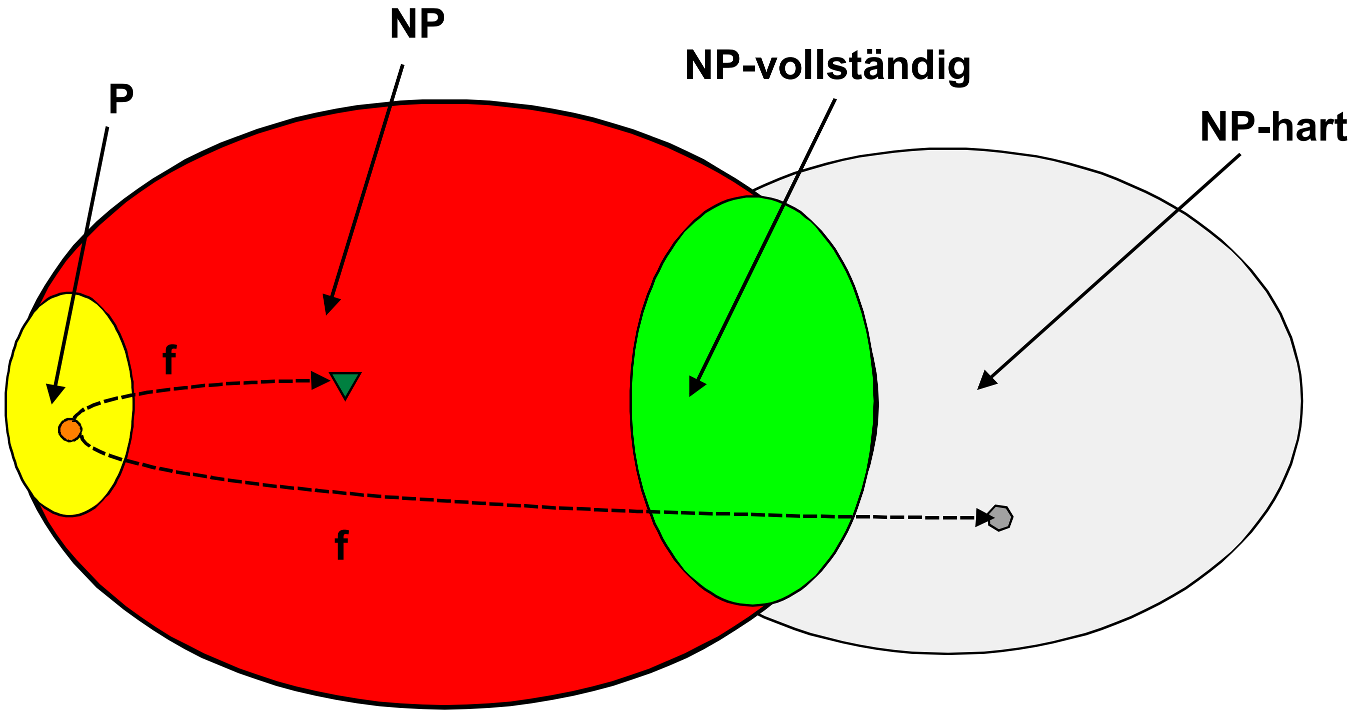
Interpretation:

Ein Problem, das **NP** hart ist, muss selbst nicht notwendigerweise in **NP** enthalten sein. Wir nennen ein Problem **NP** hart, wenn es **mindestens** so schwer ist, wie alle **NP**-vollständigen Probleme.

Klar ist dagegen, dass ein **NP**-vollständiges Problem immer auch **NP** hart ist.

NP vollständig $\subseteq$ **NP** hart

Zusammenhänge: **P** $\subseteq$ **NP** und **NP**-vollständig $\subseteq$ **NP**-hart



$\xrightarrow{f}$ = polynomial transformierbar

f ist transitiv

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. **Problemstellung der PARTITION**

Definition (PARTITION):

Gegeben sind $b_1, b_2, ..., b_n \in$ **IN**. Gibt es eine Teilmenge **K** $\subseteq \{1, 2, ..., n\}$, so dass die Summe aller b_k, $k \in$ **K**, gleich der Summe aller b_j, $j \notin$ **K** ist?

Satz:

PARTITION ist NP-vollständig.

Beweis:

PARTITION $\in$ **NP**, da wir **K** raten können. Das vollständige Raten (vollständige Aufzählung) entspricht dabei dem **Nicht-Determinismus**.

Beispiel (PARTITION):

Gegeben sei die Zahlenmenge ($b_1, b_2, \ldots, b_8$) = (2, 4, 5, 6, 7, 9, 10, 13).

Die gesuchte Teilmenge **K** $\subseteq \{ 1, 2, \ldots, 8 \}$ ist dann gleich $\{ 1, 2, 4, 5, 6 \}$, weil:

$$b_1 + b_2 + b_4 + b_5 + b_6 = b_3 + b_7 + b_8 = 28$$