03-05-H -709.53 Informatik für Nichtinformatiker (13)

Prof. Dr. Udo Frese
Tobias Hammer

Schlaglicht: Grenzen der Berechenbarkeit

Church-Turing These

Russels Antinomie

Halteproblem

P≠NP?

Church-Turing-These

Church-Turing-These

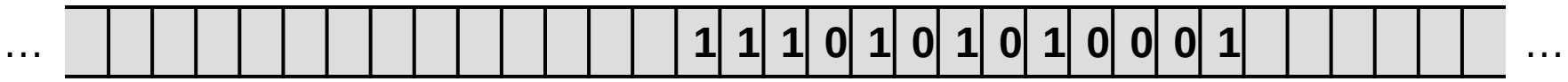
- **Alan M. Turing (1912-1950, London)**
- **„On computable numbers with an application to the Entscheidungsproblem“, 1937**
- **Was ist das Prinzip von mechanischem („algorithmischem“) Rechnen?**

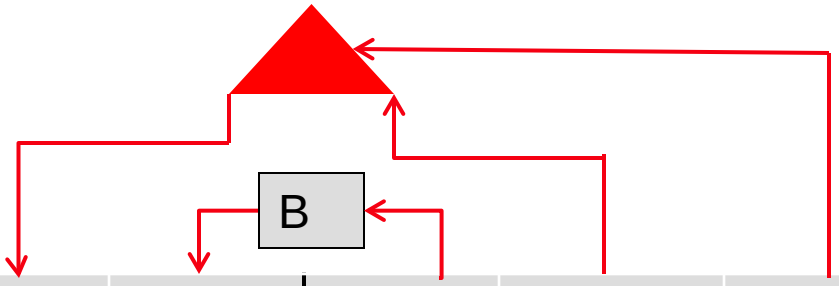
Theorie der Berechenbarkeit

- **Was ist das Prinzip von mechanischem Rechnen?**
- **Rechnung, die in vorgegebene kleinste Einzelschritte zerlegt ist**
- **Beobachtungen:**
  - ein Programm ist endlich (endliche Folge von Zeichen aus endlichem Alphabet) und fest
  - Eingabe-, Ausgabe- und Zwischendaten sind endlich (endliche Folge von Zeichen aus endlichem Alphabet) aber beliebig groß (nicht fest)
  - Speicher unendlich
  - festes Programm muss beliebig große Daten in festen Blöcken bearbeiten
- **Äquivalent:**
  - Blockweise bearbeiten
  - Zeichenweise bearbeiten
  - Bitweise bearbeiten
- **Wie greift ein endliches Programm auf unendlichen Speicher zu?**

Theorie der Berechenbarkeit

Turingmaschine (gedankliche Maschine)





Lesen	Zustand	Zustand	Schreiben	Bewegung
0	A	A	0	➔
1	A	A	1	➔
0	B	A	1	➔
1	B	B	0	➔
	A	HALT		
	B	A	1	➔

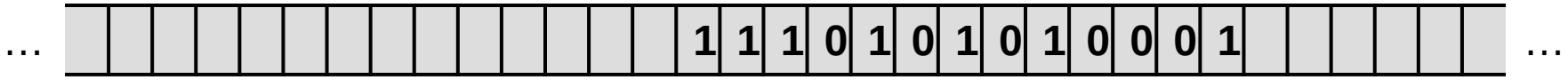
Theorie der Berechenbarkeit

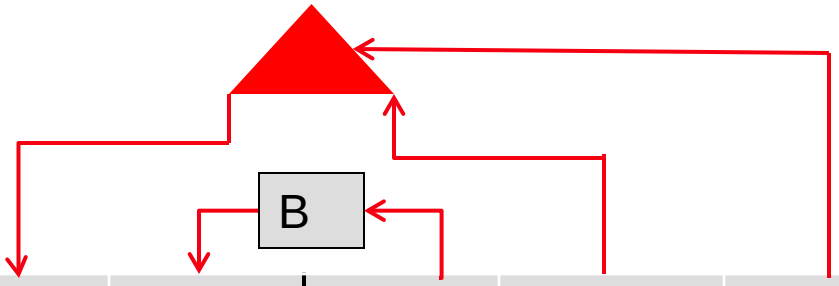
Turingmaschine (gedankliche Maschine)

- **Unendliches Band von Zeichen als Speicher**
- **Schreib- / Lesekopf bewegt sich auf dem Band**
- **enthält anfangs Eingabe**
- **enthält am Ende Ausgabe**
- **endlicher interner Speicher (Zustand)**
- **Programm ist Menge von Anweisungen**
  - wenn Maschine Zeichen x liest, …
  - und in Zustand q ist
  - schreibt sie Zeichen y,
  - geht über in Zustand q‘
  - und bewegt den Kopf nach links/rechts
- **Sonderzustand Halt**

Theorie der Berechenbarkeit

Frage an das Auditorium: Was macht diese Turingmaschine?





Lesen	Zustand	Zustand	Schreiben	Bewegung
0	A	A	0	➔
1	A	A	1	➔
0	B	A	1	➔
1	B	B	0	➔
	A	HALT		
	B	A	1	➔

Theorie der Berechenbarkeit

- **<u>Frage an das Auditorium</u>: Was macht diese Turingmaschine?**
- **Sie ersetzt am Anfang eine Folge von 1 gefolgt von einer 0 durch entsprechend viele 0en gefolgt von einer 1**
- **D.h., sie addiert 1 zu einer Binärzahl**
- **Begründung:**
  - B heißt, noch einen Übertrag addieren
  - A heißt, nichts mehr addieren
  - Bei Übertrag: 0→1 + Übertrag, 1→0 ohne Übertrag

Church-Turing-These

- **Die im intuitiven Sinne algorithmisch berechenbaren Funktionen sind genau die von Turingmaschinen berechenbaren.**
- **Argument 1: Turingmaschinen treffen den Kern von „algorithmisch“**
  - Algorithmus heißt: In kleine Schritte zerlegte Handlungsvorschrift. Der einzelne Schritt betrachtet nur einen beschränkten Teil der Gesamtinformation.
  - Genau das macht die Turingmaschine, weil jeder Schritt nur von Zustand und gelesenem Zeichen abhängt
  - Intuitives Argument, nicht beweisbar
- **Argument 2: Alle bisher vorgeschlagenen, plausiblen Formalismen sind äquivalent zur Turingmaschine**
  - Registermaschinen (sozusagen theoretischer Assembler)
  - Ersetzungsregeln auf Zeichenketten
  - Sog. μ-rekursiven Funktionen
  - Beweis: Simuliere komplexere Maschine auf Turingmaschine

Church-Turing-These

Die Rolle der Schleifen

- **Frage an das Auditorium: Was ist der fundamentale Unterschied zwischen `.times` und `.each` Schleifen im Gegensatz zu `while` Schleifen?**

Church-Turing-These

Die Rolle der Schleifen

- Frage an das Auditorium: Was ist der fundamentale Unterschied zwischen `Integer.times` und `Array.each` Schleifen im Gegensatz zu `while` Schleifen?
- Bei `.times` und `.each` steht vor dem Start der Schleife fest, wie oft sie durchlaufen wird.
- Bei `while` Schleifen ergibt sich der Abbruch durch das Kriterium während der Schleife
- Dieser Unterschied ist wichtig!
- Programme mit `.times` und `.each` kommen stets zu einem Ende
- Bei allgemeinen while Schleifen ist das unklar ($\Rightarrow$Halteproblem)
- Es gibt Funktionen, die sich nur berechnen lassen, wenn die Anzahl der Schleifendurchläufe nicht von vornherein fest stehen muss.
- Church-Turing-These erfordert allgemeine Schleifen!

Russels Antinomie

Russels Antinomie

- **Entdeckt 1903 von Bertrand Russel (1872-1970)**
- **Widerspruch in der damaligen naiven Mengenlehre nach Frege**
- **Frege: Zu jeder Eigenschaft gibt es die Menge von allem das diese Eigenschaft erfüllt**
  - Z.B. Menge aller natürlichen Zahlen, die nur durch 1 und sich selbst teilbar sind.
  - $X = \{n \in N | t \text{ teilt nicht } n \text{ für alle } 1<t<n\}$
  - Z.B. Menge aller Teilmengen von $\{\{\},\{\{\}\}\}$
  - $X=\{Y|Y \subset \{\{\},\{\{\}\}\}\}$

Russels Antinomie

- **Frage an das Auditorium: Welche Elemente hat {{},{{}}}?**

Russels Antinomie

- **Frage an das Auditorium: Welche Elemente hat {{},{{}}}?**
- **Die leere Menge {} und die Menge, die die leere Menge enthält {{}}.**
- **Frage an das Auditorium: Was ist die Menge der Teilmengen von {{},{{}}}?**

Russels Antinomie

- **Frage an das Auditorium: Welche Elemente hat {{},{{}}}?**
- **Die leere Menge {} und die Menge, die die leere Menge enthält {{}}.**
- **Frage an das Auditorium: Was ist die Menge der Teilmengen von {{},{{}}}?**
- **Die Menge bestehend aus a) der leeren Menge, b) der Menge, die die leere Menge enthält, c) der Menge, die die Menge enthält ,die die leere Menge enthält und d) der Menge die sowohl die Menge enthält, die die leere Menge enthält als auch die Menge, die die Menge enthält, die die leere Menge enthält.**
- **{{}, {{}}, {{{}}}, {{},{{}}}}**

Russels Antinomie

- **<u>Frage an das Auditorium</u>: Der Frisör rasiert alle, die sich nicht selbst rasieren.**

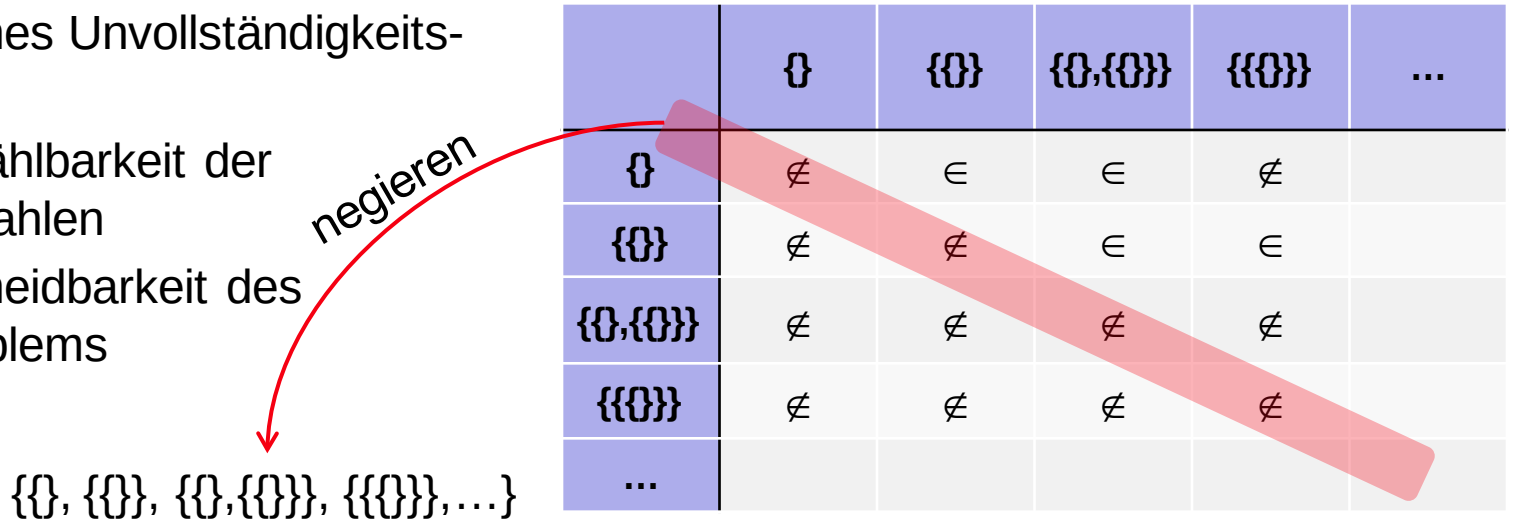
Russels Antinomie

- **Menge aller Mengen, die nicht in sich selbst enthalten sind**
  - $R = \{M | M \notin M\}$
- **Ist $R \in R$?**
- **Angenommen ja: $R \in R$**
  - $\Rightarrow$ R muss definierende Eigenschaft erfüllen
  - $\Rightarrow R \notin R$
  - $\Rightarrow$ Widerspruch
- **Angenommen nein: $R \notin R$**
  - $\Rightarrow$ R darf definierende Eigenschaft nicht erfüllen
  - $\Rightarrow R \in R$
  - $\Rightarrow$ Widerspruch
- **Folgerung: Die Menge R kann nicht existieren.**

Russels Antinomie

Prinzip hinter der Argumentation

- **Selbstbezüglichkeit (Diagonalisierung)**
- **Negation**
- **Mehrfach auftretendes Phänomen**
  - Der Frisör rasiert alle, die sich nicht selbst rasieren
  - „Ich lüge jetzt.“
  - Gödelsches Unvollständigkeits-theorem
  - Überabzählbarkeit der reellen Zahlen
  - Unentscheidbarkeit des Halteproblems



Halteproblem

Halteproblem

Ein selbstbezügliches algorithmisches Problem

- **Programme, die etwas über Programme sagen**
- **Formal: Turingmaschinen, intuitiv: Rubyprogramme**
- **Halteproblem: Gegeben ein Programm P und eine Eingabe x, würde P auf x gestartet irgendwann einmal anhalten?**
- **Ein Programm H, das das Halteproblem löst, würde**
  - (P,x) als Eingabe erhalten,
  - eine endliche Zeit rechnen
  - True ausgeben, falls P auf x zu einem Ende kommt
  - False ausgeben, falls P auf x in eine Endlosschleife geraten würde
- **Schwierigkeit: Irgendwann einmal feststellen, dass P endlos weiterlaufen wird und dann False ausgeben.**
- **Zu zeigen: Es gibt kein Programm, das das Halteproblem löst.**

Halteproblem

Halteverhalten

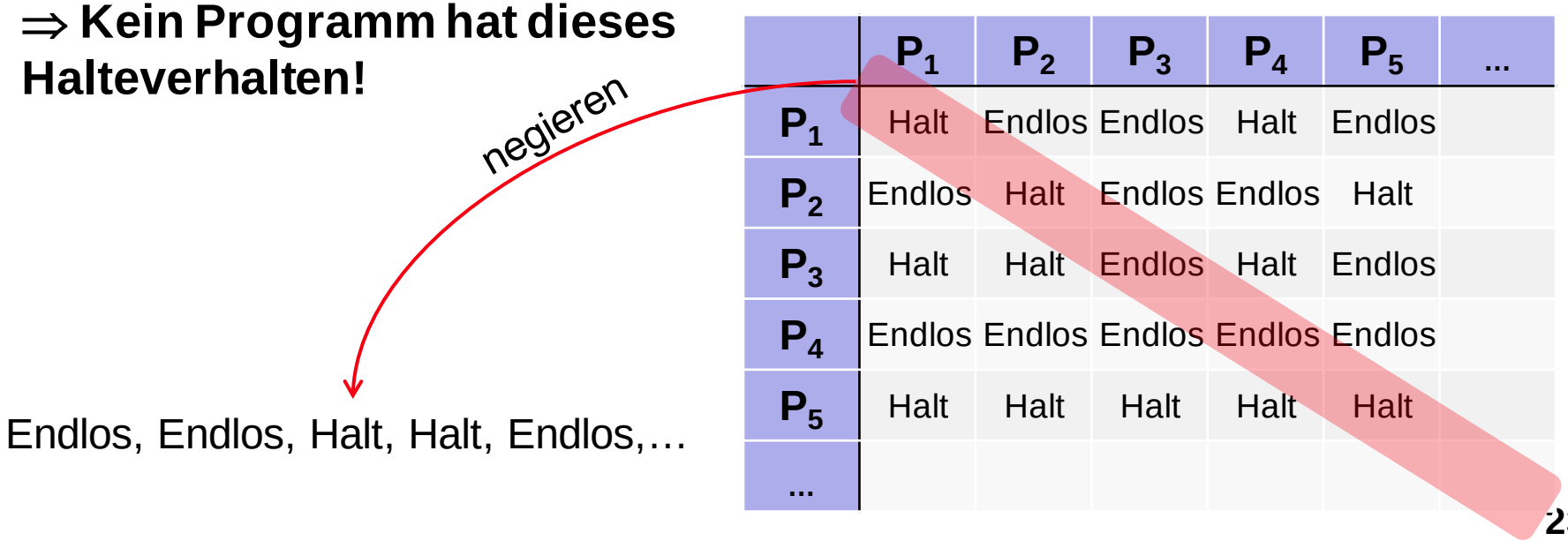
- **Selbstbezüglichkeit: Wie verhält sich Programm P, wenn es ein anderes Programm P‘ als Eingabe x erhält?**
- **Für welche Programme P‘=x als Eingabe hält das Programm P an, wann gerät es in eine Endlosschleife?**
- **Frage an das Auditorium: Könnt ihr ein unmögliches Halteverhalten konstruieren?**

	P_1	P_2	P_3	P_4	P_5	...
P_1	Halt	Endlos	Endlos	Halt	Endlos	
P_2	Endlos	Halt	Endlos	Endlos	Halt	
P_3	Halt	Halt	Endlos	Halt	Endlos	
P_4	Endlos	Endlos	Endlos	Endlos	Endlos	
P_5	Halt	Halt	Halt	Halt	Halt	
...						

Halteproblem

Halteverhalten

- **Frage an das Auditorium: Könnt ihr ein unmögliches Halteverhalten konstruieren?**
- **Auf Eingabe P genau dann halten, wenn P auf Eingabe P nicht hält.**
- **Hätte Programm U dieses Halteverhalten, dann müsste es auf U als Eingabe genau dann halten, wenn U auf U nicht hält. Widerspruch!**
- **⇒ Kein Programm hat dieses Halteverhalten!**



	P_1	P_2	P_3	P_4	P_5	...
P_1	Halt	Endlos	Endlos	Halt	Endlos	
P_2	Endlos	Halt	Endlos	Endlos	Halt	
P_3	Halt	Halt	Endlos	Halt	Endlos	
P_4	Endlos	Endlos	Endlos	Endlos	Endlos	
P_5	Halt	Halt	Halt	Halt	Halt	
...						

Halteproblem

Reduktion des Halteproblems auf unmögliches Halteverhalten

- **Lösen eines Problems mit Hilfe einer angenommenen Lösung für ein anderes Problem**
- **Hier: Wenn es ein Programm H gäbe, das das Halteproblem löst, dann gäbe es auch ein Programm U mit unmöglichen Halteverhalten.**
- **Definition von H**
  - Immer anhalten
  - Auf Eingabe (P,x) True liefern, wenn P auf x hält und sonst False.
- **Unmögliches Halteverhalten von U**
  - Auf Eingabe P genau dann halten, wenn P auf Eingabe P nicht hält.
- **Frage an das Auditorium: Könnt Ihr aus dem hypothetischen Programm H das unmögliche Programm U konstruieren?**

Halteproblem

- Frage an das Auditorium: Könnt Ihr aus dem hypothetischen Programm H das unmögliche Programm U konstruieren?
- Wenn P auf P hält, dann liefert H(P,P) True und U(P) bleibt in der Endlosschleife
- Wenn P auf P endlos bleibt, dann liefert H(P,P) False und U(P) hält.
- ⇒ U hält auf P, genau dann, wenn P nicht auf P hält
- ⇒ Widerspruch!
- ⇒ Halteproblem ist unlösbar!

```
def H(P,x)
  … # hypothetisches Programm H
    # das das Halteproblem löst
end

def U(P)
  if H(P,P) then
    while true do end
  end
end
```

Halteproblem

Postsches Korrespondenzproblem (Emil Leon Post)

- **Gegeben: Eine Menge von Dominosteinen mit Text auf oberer und unterer Hälfte. Gesucht: Gibt es eine Folge von Steinen, das der Text auf den oberen und unteren Hälften gleich ist. Jeder Typ Stein ist beliebig oft vorhanden.**
- **Unlösbar!**

AXTXT	XT
A	XTXT

AXTXT	XT	XT
A	XTXT	XTXT

P≠NP?

P≠NP?

- **Größtes offenes Problem der theoretischen Informatik**
- **Ist das Finden einer Lösung (P) fundamental schwieriger, als das Prüfen einer Lösung (NP)?**
- **Vermutung: Ja, P≠NP. Aber nicht bewiesen.**

P≠NP?

- Größtes offenes Problem der theoretischen Informatik
- Was ist schnell berechenbar?
- Ist das Finden einer Lösung (P) fundamental schwieriger, als das Prüfen einer Lösung (NP)?
- Vermutung: Ja, P≠NP. Aber nicht bewiesen.
- Es geht um Rechenzeit in Abhängigkeit von Eingabelänge n
- Problem ist Funktion Eingabe→Ausgabe
- Programm löst ein Problem, bzw. berechnet eine Funktion, wenn auf der Eingabe gestartet, es die Ausgabe berechnet.

P≠NP?

- P ist Menge aller Probleme, die mit Rechenzeit $<n^k$ berechnet werden kann, also Lösung in $<n^k$ *finden*.
- NP ist Menge aller Probleme, deren Lösung mit Rechenzeit $<n^k$ *überprüft* werden kann
- Alle möglichen Lösungen zu prüfen braucht Zeit $>2^n>n^k$
- P≠NP heißt: Es gibt Probleme, bei denen man die Lösung schnell prüfen, aber nur langsam finden kann.

P≠NP?

Beispiel: Faktorisieren großer Zahlen

- **Seien p, q große (n-stellige, n>1000) Primzahlen**
- **Zahlen p, q können multipliziert werden zu p·q**
- **Produkt p·q ist eindeutig in die Primfaktoren p, q zerlegbar**
- **Multiplizieren, also (p,q)→p·q braucht n^2 Rechenzeit**
- **Faktorisieren, also p·q→(p,q) braucht sehr viel Rechenzeit**
  - vermutlich $>n^k$
  - Faktorisieren vermutlich $\notin$P
- **Prüfen der Lösung einer Faktorisierung heißt multiplizieren (p,q)→p·q, braucht n^2 Rechenzeit**
  - Faktorisieren $\in$NP
- **An der Schwierigkeit des Faktorisierens hängt die Sicherheit des RSA Verschlüsselungssystems, wo p·q öffentlich gemacht wird, aber nur der rechtmäßige Empfänger p und q kennt.**

P≠NP?

Beispiel: Faktorisieren großer Zahlen

- **Seien p, q große (n-stellige, n>1000) Primzahlen**
- **Zahlen p, q können multipliziert werden zu p·q**
- **Produkt p·q ist eindeutig in die Primfaktoren p, q zerlegbar**
- **Multiplizieren, also (p,q)→p·q braucht n^2 Rechenzeit**
- **Faktorisieren, also p·q→(p,q) braucht sehr viel Rechenzeit**
  - vermutlich $>n^k$
  - Faktorisieren vermutlich $\notin$P
- **Prüfen der Lösung einer Faktorisierung heißt multiplizieren (p,q)→p·q, braucht n^2 Rechenzeit**
  - Faktorisieren $\in$NP
- **An der Schwierigkeit des Faktorisierens hängt die Sicherheit des RSA Verschlüsselungssystems, wo p·q öffentlich gemacht wird, aber nur der rechtmäßige Empfänger p und q kennt.**

P≠NP?

Weitere NP Probleme, die vermutlich ∉P sind

- Viele Optimierungsprobleme
- Kürzeste Rundreise
- SAT (Auflösung Gleichungssysteme über und, oder, nicht)
- Kürzestes Netzwerk, das eine Menge von Punkten verbindet
- Optimale Bewegung für einen Roboter
- Kleinste rechteckige Fläche aus der eine Menge an Teilen geschnitten werden kann
- Wertvollsten „Rucksack“ aus einer Menge von Objekten mit Gewicht und Wert auswählen

P≠NP?

NP-Vollständigkeit

- **Ein Problem ist NP-vollständig, wenn sofern es in P liegt, alle NP Probleme in P liegen.**
- **NP-vollständigen Probleme sind „schwersten“ Probleme in NP**
- **Satz von Cook: SAT ist NP-vollständig**
- **SAT: Lösung logischer Gleichungssystemen über und, oder, nicht**
- **Beweisidee:**
  - Angenommen, wir hätten eine P-Lösung für SAT
  - Problem liegt in NP $\Rightarrow$ Es gibt ein Programm Q, das eine Lösung überprüft
  - Kodiere das Programm Q als logische Gleichung, die auf der unbekannten Lösung x operiert
  - Gleichungssystem sagt: „Gibt es eine Lösung x, die Q akzeptieren würde?“
  - Finde Lösung der Gleichung mit Hilfe der P-Lösung für SAT

P≠NP?

NP-Vollständigkeit

- **Fast alle praktisch relevanten NP-Probleme ohne bekannten P-Algorithmus sind NP-Vollständig**
- **Faktorisierung ist eines der wenigen Probleme, die weder bekanntermaßen in P liegen, noch bekanntermaßen NP-vollständig sind.**

Zusammenfassung

- **Church-Turing-These**
  - Intuitive Algorithmen entsprechen den Turingmaschinen
  - Operieren schrittweise auf einem beschränkten Teil der Daten
- **Halteproblem**
  - Es gibt Probleme, die sich prinzipiell nicht berechnen lassen
  - Halteproblem: Hält Programm P auf Eingabe x?
  - Besonders selbstbezügliche Probleme, wie das Halteproblem
  - Aber auch „unschuldig“ aussehende kombinatorisches Probleme, wie das Postsche Korrespondenzproblem
- **NP≠P**
  - NP: Probleme für die sich eine Lösung in Zeit n^k überprüfen läst
  - P: Probleme für die sich eine Lösung in Zeit n^k finden lässt
  - Ca. 3000 relevante, NP-vollständige Probleme. Ist eines davon in P, so NP=P
  - Die meisten Informatiker vermuten NP≠P, aber bis heute nicht bewiesen