5. Theorie der Algorithmen

5.1 Berechenbarkeit

5.2 Komplexität

5.3 Korrektheit und Verifikation

5.4 Endliche Automaten

5.1 Berechenbarkeit

Frage:

Gilt für jedes beliebige Problem, dass es entweder mit dem Computer lösbar (berechenbar) ist oder als unlösbar erkannt werden kann? (**Hilbertsches Entscheidungsproblem**)

Antwort:

Nein!

Das Hilbertsche Entscheidungsproblem ist nicht berechenbar.

Beispiel:

Gegeben sei irgendeine beliebige Aussage über die natürlichen Zahlen. Dann gibt es keinen Algorithmus, der feststellt, ob diese Aussage wahr oder falsch ist. (Unvollständigkeitstheorem von **Gödel**)

Weitere Beispiele von Church, Kleene, Post, Turing, ca.1930 – 1940.

Präzisierung des Algorithmus-Begriffs

- **Gödel:** Folge von Regeln zur Bildung mathematischer Funktionen aus einfacheren mathematischen Funktionen.
- **Church:** Lambda-Kalkül.
- **Turing:** Turin-Maschine (eine abstrakte Maschine mit sehr einfachen Anweisungen).
- **Church-Turing-These:** Alle Definitionen von Algorithmus sind gleich-wertig.

Äquivalenz der Algorithmus-Begriffe

Man könnte einen Algorithmus auch definieren als eine Verarbeitungsvorschrift, die in Java angegeben werden kann.

Dann vermutet man schon, dass ein äquivalenter Algorithmus auch in **C** oder **FORTRAN** oder **Pascal** oder **LISP** oder **PROLOG** angegeben werden kann. Dies trifft in der Tat zu.

Programmiersprachen unterscheiden sich nicht in der **Mächtigkeit**, sondern in der **Natürlichkeit, Effizienz, Klarheit** usw., in der sie ein bestimmtes Problem ausdrücken können.

Wir erkennen:

Die Berechenbarkeit eines Problems hängt nicht von der Programmiersprache ab.

Das Halteproblem

Gegeben ein **beliebiges** Programm **P** und **beliebige** Daten **D** dazu.

Frage: Wird das Programm jemals anhalten, oder läuft es endlos lange?

Antwort: Das Halteproblem ist nicht berechenbar!

Das bedeutet, dass es keinen Algorithmus gibt, der für **beliebiges P** und **beliebiges D** das Halteproblem lösen kann.

Beispiel: Fermat'sche Vermutung

Es gibt keine natürlichen Zahlen a, b, c, so dass für $n > 2$ gilt:

$$a^n + b^n = c^n$$

(Fortsetzung)

Beispiel: Fermat'sche Vermutung

Algorithmus

```
Modul Fermat (n)
// Prüfe die Fermat'sche Vermutung mit n als Eingabe
Wiederhole für a = 1, 2, 3, …
      Wiederhole für b = 1, 2, 3, …, a
            Wiederhole für c = 2, 3, 4, …, a + b
                  Falls a^n + b^n = c^n
                  dann gib a, b, c und n aus, stoppe
```

Stoppt für $n = 1$ $(a = 1, b = 1, c = 2)$

Stoppt für $n = 2$ $(a = 4, b = 3, c = 5)$

Was geschieht für beliebiges $n > 2$?

Beweis der Nichtberechenbarkeit des Halteproblems

Beweistechnik: Beweis durch Widerspruch

- Angenommen, es gäbe einen Algorithmus, der das Halteproblem löst. Wir nennen ihn `Stop-Tester`.
- Dann konstruieren wir einen Algorithmus "`fun`" derart, dass `Stop-Tester(fun)` die Antwort "JA" liefert, wenn "`fun`" nicht terminiert und die Antwort "NEIN", wenn "`fun`" terminiert.
- Damit haben wir einen immanenten Widerspruch, und es kann `Stop-Tester` somit nicht geben.

Algorithmus "Stop-Tester"

```
Lies P          /* Programm */
Lies D          /* Daten    */

Falls           P(D) stoppt,
dann            gib JA aus
sonst           gib NEIN aus.
```

Stop-Tester-Spezial

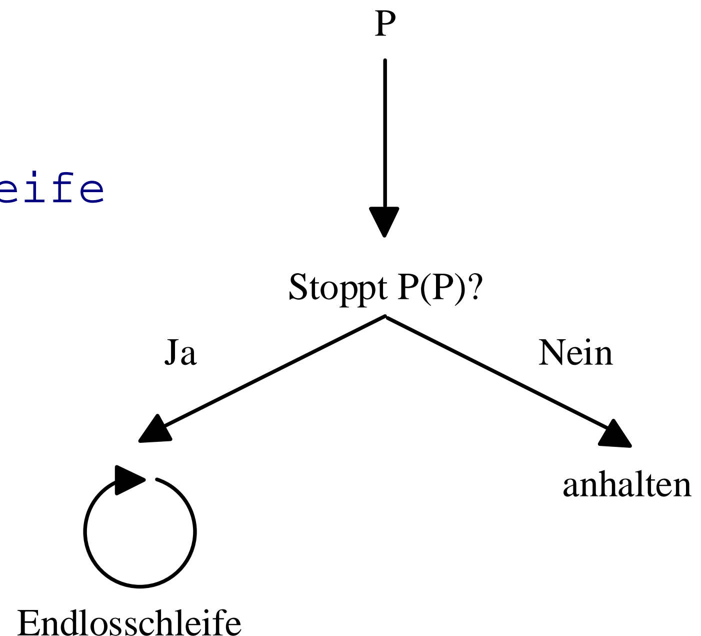
```
Lies P          /* Programm */
Lies P          als Daten

Falls           P(P) stoppt,
dann            gib JA aus
sonst           gib NEIN aus.
```

Algorithmus `fun(P)`

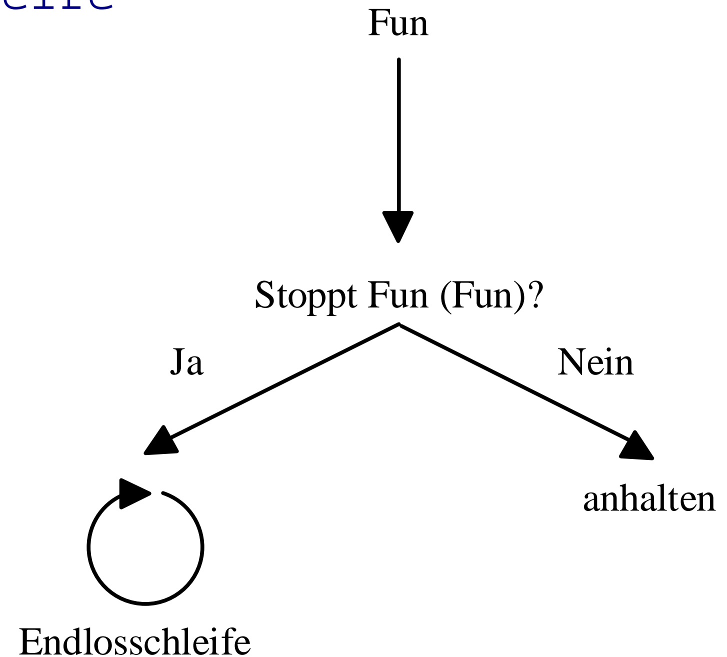
/* Wir nehmen an, dass der Algorithmus **`Stop-Tester`** und damit auch **`Stop-Tester-Spezial`** existiert. */

```
Lies   P
Stop-Tester-Spezial(P)
Falls  Antwort = JA
dann   gehe in eine Endlosschleife
sonst  stoppe.
```



Algorithmus `fun(fun)`

```
Lies   fun
Stop-Tester-Spezial(fun)
Falls  Antwort = JA
dann   gehe in eine Endlosschleife
sonst  stoppe.
```



Widerspruch in `fun(fun)`

Wenn der Algorithmus “**`fun`**” stoppt, dann gerät er in eine Endlosschleife.

Wenn “**`fun`**” nicht stoppt, dann gerät er auf eine **`STOP`**-Anweisung.

Wir schließen daraus:

Gäbe es einen Algorithmus **`Stop-Tester`**, so würde sich unter ausschließlicher Verwendung dieses Algorithmus ein immanenter Widerspruch zeigen lassen. Daraus folgt, dass es einen Algorithmus **`Stop-Tester`** nicht geben kann.

Das Äquivalenzproblem

Gegeben seien zwei beliebige Programme P_1 und P_2 . Man gebe einen Algorithmus an, der feststellt, ob P_1 und P_2 für beliebige Daten D dieselbe Ausgabe erzeugen (also äquivalent sind).

Behauptung: Das Äquivalenzproblem ist nicht berechenbar.

Partielle Berechenbarkeit

Bisher haben wir verschiedene Probleme betrachtet, die **nicht berechenbar** sind (d.h. keine algorithmische Lösung haben).

Ein Problem heißt **partiell berechenbar**, wenn es einen Algorithmus gibt, der für jeden Eingabewert, für den ein Ergebnis definiert ist, nach endlich vielen Schritten anhält und das korrekte Ergebnis liefert. In allen anderen Fällen, in denen kein Ergebnis definiert ist, bricht der Algorithmus nicht ab.

Beispiel

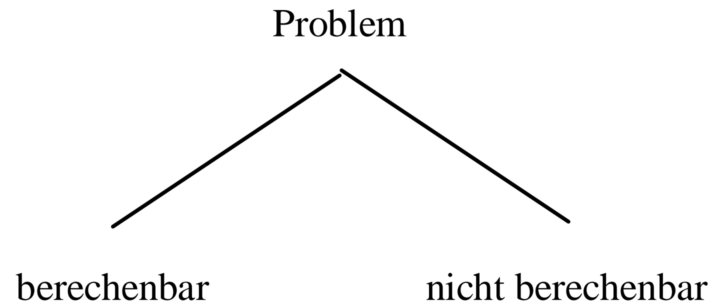
Das Halteproblem ist partiell berechenbar.

Beweis:

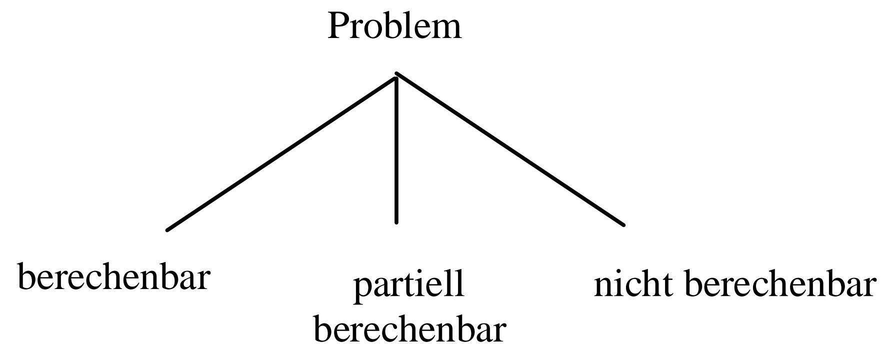
Schrittweise Ausführung des Programms P(D). Falls die schrittweise Ausführung hält, wissen wir, dass das Programm hält. Falls die schrittweise Ausführung nicht hält, können wir keine Aussage machen (vielleicht wird sie irgendwann in sehr ferner Zukunft halten?).

Partielle Berechenbarkeit

Bisher:



Jetzt:



Rekursionssatz

Gegeben sei ein Programm P(D), das eine bestimmte Funktion ausführt. Dann gibt es ein Programm P'(P'), das dieselbe Funktion ausführt, wobei es eine Kopie von sich selbst als Eingabedaten benutzt.

Beispiel 1:

```
Modul Drucke(D)

Gib D aus.
```

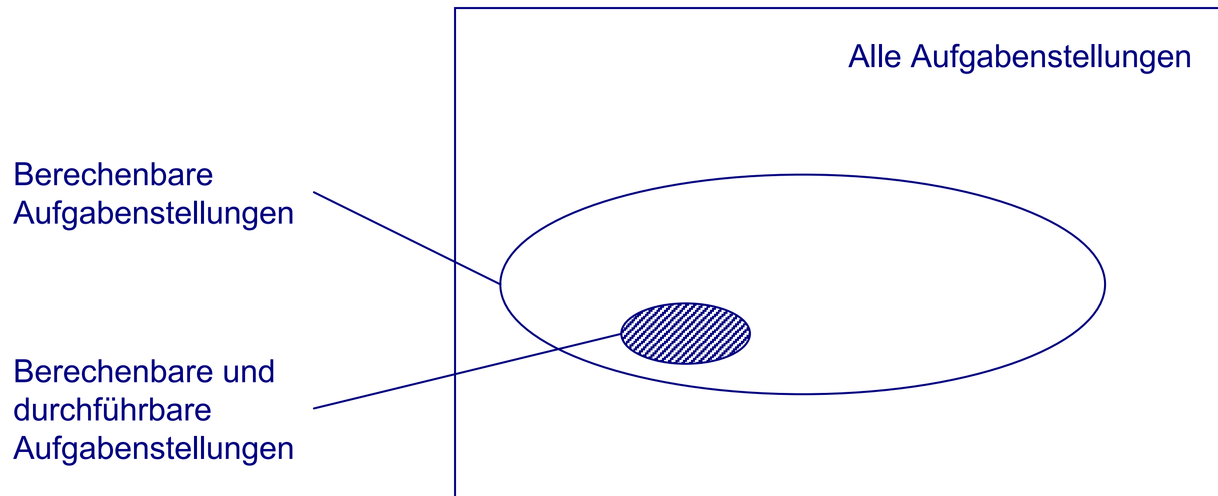
Dann gibt es ein Modul 'Drucke', das seinen eigenen Programmtext ausdruckt.

Beispiel 2:

Man kann einen Compiler in einer Programmiersprache so schreiben, dass er sich selbst kompilieren kann!

5.2 Komplexität von Algorithmen

Berechenbarkeit, wie zuvor eingeführt, bedeutet nur, dass ein Problem **prinzipiell** algorithmisch lösbar ist. Es heißt aber nicht, dass es **mit vernünftigem Aufwand** praktisch berechenbar ist.



Ressourcen

Bei der Ermittlung der Komplexität eines Algorithmus betrachtet man in der Regel die folgenden Ressourcen:

Zeit und **Speicherplatz**.

Häufig geht eine Optimierung des Bedarfs an einer Ressource auf Kosten der anderen Ressource (z. B. weniger Zeit, dafür mehr Speicher).

In der Komplexitätstheorie wird meist die **Anzahl der Operationen in Abhängigkeit von der Anzahl der Eingabedaten** als Komplexitätsmaß betrachtet.

Beispiel 1: Komplexität der Multiplikation

Es seien zwei vierstellige Zahlen zu multiplizieren. Der übliche Schul-Algorithmus wird angewendet. Jede Zeile kann in vier Operationen errechnet werden. Es gibt vier Zeilen, die anschließend zu addieren sind.

Insgesamt ist die Anzahl der Operationen proportional zu n^2, wenn n die Anzahl der Ziffern ist.

```
1984  x 6713
------------
 11904
   1 3888
       1984
         5952
------------
  13318592
```

Verbesserter Multiplikationsalgorithmus (1)

Zerlegung der vierstelligen Multiplikation in zwei zweistellige Multiplikationen:

A	B		C	D
19	84	x	67	13

$$(100*A + B) * (100*C + D)$$

$$= 10000*AC + 100*AD + 100*BC + BD$$

$$= 10000*AC + 100*(AD + BC) + BD$$

$$= 10000*AC + 100*[(A + B)(C + D) - AC - BD] + BD$$

(Fortsetzung) →

Verbesserter Multiplikationsalgorithmus (2)

AC = 19 x 67 = **1273**

(A+B)(C+D)-AC-BD = (103x80)-1273-1092 = **5875**

BD = 84 x 13 = **1092**

```
1273
  5875
    1092
--------
13318592
```

Alle Multiplikationen sind nur noch zweistellig. Es sind nur noch drei Zeilen mit je vier Ziffern zu addieren.

Man kann generell zeigen, dass die Anzahl der Operationen proportional zu $n^{1,59} < n^2$ ist.

Beispiel 2: Sortieren durch Auswahl (1)

Gegeben seien **n** Zahlen, die zu sortieren sind.

Algorithmus `Auswahlsortieren`

```
Wiederhole n-mal
   Markiere die erste Zahl der Eingabemenge.
   Vergleiche sie mit der zweiten, dritten usw.
   Wenn eine kleinere als die erste gefunden wird,
       markiere diese.
   Wenn das Ende der Eingabemenge erreicht ist,
       übertrage die markierte kleinste Zahl in den
         Ausgabe-Datenstrom und sperre sie für die
         weitere Bearbeitung.
```

Beispiel 2: Sortieren durch Auswahl (2)

Komplexität: proportional zu n^2

(n Durchläufe mit jeweils größenordnungsmäßig n Vergleichen)

Anmerkung: Es gibt Sortieralgorithmen mit Komplexität n log n.

Asymptotisches Verhalten

In der Praxis interessiert oft nicht so sehr die genaue Anzahl der Operationen, sondern das **asymptotische Wachstum für große n**.

Beispiel:

Sei $3n^2 + 5n$ die Anzahl der Operationen. Dann ist für großes n der quadratische Anteil dominierend. Man sagt, die Komplexität wächst mit n^2.

Größe n der Daten	**$\log_2 n$ s**	**n s**	**n^2 s**	**2^n s**
10	0.000003 s	0.00001 s	0.0001 s	0.001 s
100	0.000007 s	0.0001 s	0.01 s	10^{14} Jahrhunderte
1,000	0.00001 s	0.001 s	1 s	astronomisch
10,000	0.000013 s	0.01 s	1.7 min	astronomisch
100,000	0.000017 s	0.1 s	2.8 h	astronomisch

O-Kalkül

Berechnung der Komplexität unter Abstraktion

- von Konstanten (die maschinenabhängig sind),
- von weniger dominanten Termen, die für große Werte des Komplexitäts-parameters n nicht mehr signifikant sind.

Asymptotisches Wachstum (1)

Definition 1

Seien f und g zwei positive, reellwertige Funktionen, die auf einem gemeinsamen Bereich D definiert sind. Dann bedeutet

$$\mathbf{f = O(g)},$$

dass es eine Konstante $c > 0$ und einen Anfangswert x_0 gibt derart, dass

$$f(x) \leq cg(x) \quad \text{für alle } x > x_0, \quad x, x_0 \in D$$

gilt.

Man sagt, **f wächst asymptotisch höchstens wie g**.

Asymptotisches Wachstum (2)

Definition 2

Es bedeutet

$$\mathbf{f = \Omega(g)},$$

dass es eine Konstante $c > 0$ und einen Anfangswert x_0 gibt derart, dass

$$f(x) \geq cg(x) \quad \text{für alle } x > x_0,\ x, x_0 \in D$$

gilt.

Man sagt, **f wächst asymptotisch mindestens wie g**.

Beispiel für den O-Kalkül

Sortieren durch Auswahl

Eingabeparameter: n Anzahl der Elemente

Der Vergleich zweier Zahlen ist in konstanter Zeit c_1 möglich.

Innere Schleife: $t_1 = nc_1$

Äußere Schleife: $t_2 = nt_1 = n^2c_1$

Gesamter Rechenaufwand: $t = t_2 = n^2c_1 = O(n^2)$

Einfache Rechenregeln des O-Kalküls

```
(1) f        = O(f)
(2) O(O(f))= O(f)
(3) cO(f)    = O(f)
(4) O(f+c)   = O(f)
(5) O(f) + O(f) = O(f)
(6) O(f) * O(g) = O(fg)
(7) O(f) + O(g) = O(f+g)
(8) O(fg) = O(f)*O(g)
```

Konvention: Diese Gleichungen werden von links nach rechts gelesen.

O(f) ist, mathematisch gesehen, eine Menge von Funktionen.

Mengenoperationen:

$M_1 + M_2 := \{ x + y \mid x \in M_1 \text{ und } y \in M_2\}$

$M_1 * M_2 := \{ x * y \mid x \in M_1 \text{ und } y \in M_2\}$

Beweis von Regel (7) als Beispiel

```
(7)O(f) + O(g) = O(f+g)
```

$h_1(n) = O(f)$, also: Es gibt c_1, n_1 mit $h_1(n) \leq c_1 f(n)$ für alle $n > n_1$

$h_2(n) = O(g)$, also: Es gibt c_2, n_2 mit $h_2(n) \leq c_2 g(n)$ für alle $n > n_2$

Addition liefert

$$h_1(n) + h_2(n) \leq c_1 f + c_2 g \text{ für alle } n > \max(n_1, n_2).$$

Setzen wir $h_3(n) = h_1(n) + h_2(n)$, $c_3 = \max(c_1, c_2)$ und $n_3 = \max(n_1, n_2)$, so gilt

$$h_3(n) \leq c_3 (f + g), \text{ für alle } n > n_3,$$

was nach Definition $h_3(n) = O(f + g)$ bedeutet.

Multiplikation im O-Kalkül (1)

Beispiel:

```
1984  x 6713
 11904
   1 3888
       1984
         5952
  13318592
```

Eingabeparameter: n_1, n_2 Länge der Faktoren

- Multiplikation zweier Ziffern sei in konstanter Zeit c_1 möglich.
- Addition zweier Ziffern sei in konstanter Zeit c_2 möglich.

Multiplikation im O-Kalkül (2)

Berechnung einer Zeile: $t_1 = n_1 * c_1$

Berechnung von n_2 Zeilen: $t_2 = n_2 * t_1 = n_2 * n_1 * c_1$

Addition von n_2 Zeilen der Länge $\leq n_1 + 2$:

$t_3 = n_2 * (n_1 + 2) * c_2$

$$
\begin{aligned}
t = t_2 + t_3 &= n_1 n_2 c_1 + (n_1 + 2) n_2 c_2 \\
&= n_1 n_2 c_1 + n_1 n_2 c_2 + 2 n_2 c_2 \\
&= n_1 n_2 (c_1 + c_2) + 2 n_2 c_2 \\
&= O(n_1 n_2 + n_2) \\
&= O(n_1 n_2)
\end{aligned}
$$

Mit $n := \max \{n_1, n_2\}$ ist $t = O(n*n) = O(n^2)$

Zwei Arten von Schranken (1)

Für ein gegebenes Problem lässt sich oft eine untere Schranke für die Komplexität einer Lösung aus der Theorie angeben, **unabhängig vom Algorithmus**.

Beispiel 1:

Suche das Minimum aus n unsortierten Zahlen.

Jede Zahl muss mindestens einmal angesehen (verglichen) werden. Daher gilt: Das Problem der Minimumsuche hat Komplexität $\mathbf{\Omega(n)}$.

Zwei Arten von Schranken (2)

Darüber hinaus lässt sich **für einen gegebenen Algorithmus** die Abhängigkeit der Anzahl der Operationen von n ermitteln und dadurch eine **für diesen Algorithmus** spezifische Schranke ermitteln.

Beispiel 2:

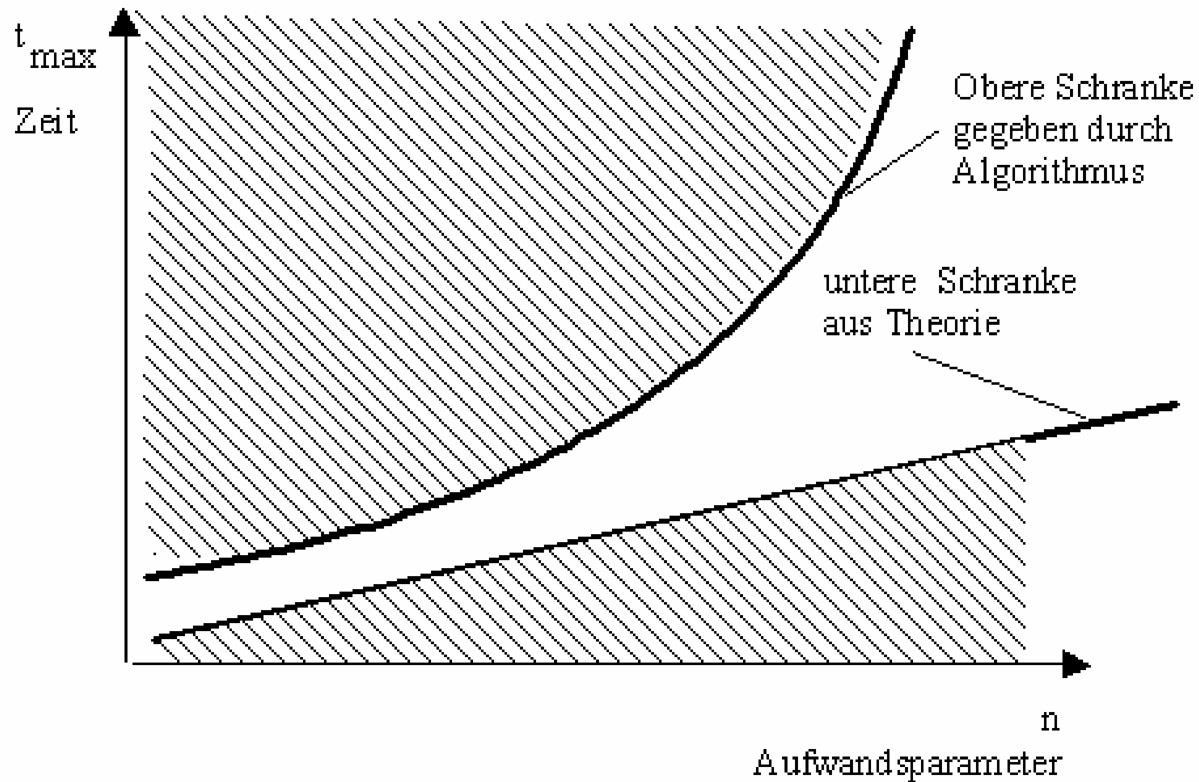
Ein dummer Algorithmus A_1 sortiert die n Zahlen und erhält dann das Minimum als erste Zahl des Ergebnisses.

Sortieren durch Auswahl: $O(n^2)$

Bestimmung der ersten Zahl: $O(1)$

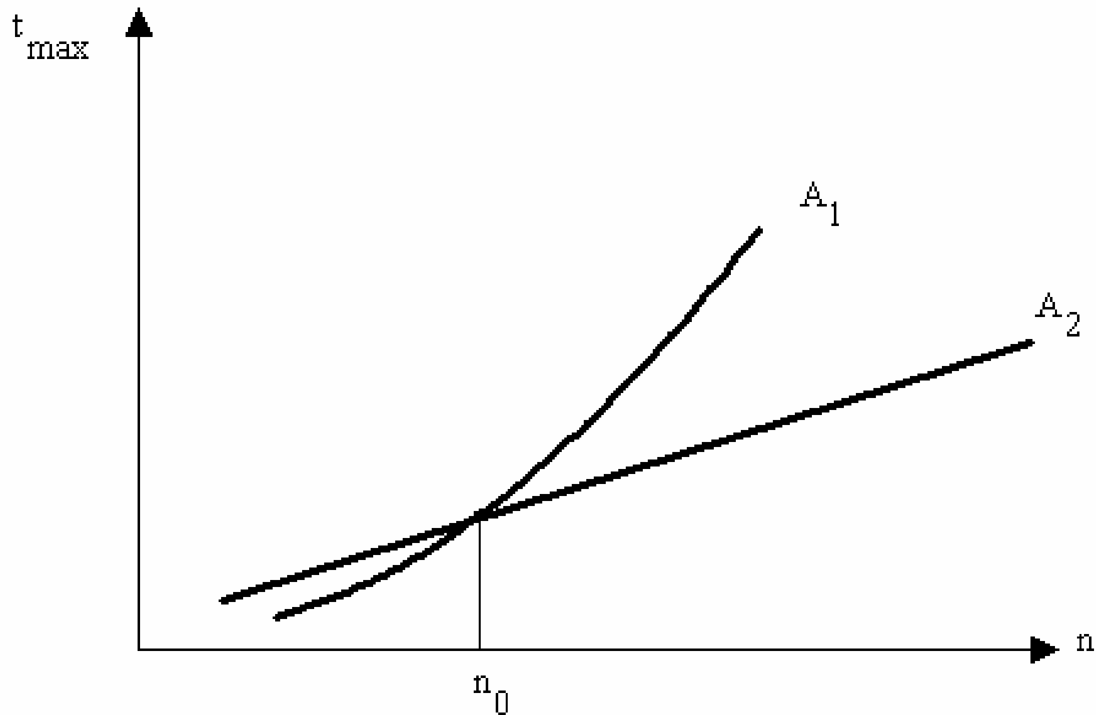
Der Algorithmus A_1 hat also die Komplexität $\mathbf{O(n^2)}$.

Obere und untere Schranken



Die Erforschung solcher Schranken wird in der **Komplexitätstheorie** behandelt, einem wichtigen Zweig der Theoretischen Informatik.

Vergleich zweier Algorithmen



Wir sehen:

Der asymptotisch optimale Algorithmus ist nicht unbedingt der bessere für alle n; es kann sich lohnen, für kleine n einen anderen Algorithmus zu wählen als für große n.

Rekurrenzrelation: "Teile und herrsche"

Das Verfahren "Teile und Herrsche" erweist sich oft als gutes Entwurfsprinzip für effiziente Algorithmen.

Idee
Aufteilen des Gesamtproblems in kleinere Teilprobleme derart, dass die Summe der Aufwände für die Teilprobleme und des Aufwands für das Zusammenfügen geringer ist als der Aufwand für das direkte Lösen des Gesamtproblems.

Dies ist der Trick bei allen **effizienten** Sortieralgorithmen.

"Teile-und-herrsche": Sortieren (Beispiel 1)

```
Modul TH-Sortiere(Liste)
/* Sortiert eine gegebene Liste aus n Namen alphabetisch */

Falls n > 1
dann TH-Sortiere(erste Listenhälfte)
     TH-Sortiere(zweite Listenhälfte)
     Mische die zwei Hälften zusammen
```

Aufwand für TH-Sortier-Algorithmus (1)

- $T(n)$ sei der Aufwand für das Sortieren von n Zahlen.
- Sortieren von n/2 Zahlen erfordert $T(n/2)$.
- Mischen der beiden Hälften ist proportional zu n.

Der Aufwand für das TH-Sortieren durch Halbieren und Mischen ist also

$$T(n) = 2\,T(n/2) + cn$$

$$T(1) = k$$

Aufwand für TH-Sortier-Algorithmus (2)

Diese **Rekurrenzrelation** (rekursives Gleichungssystem) hat die geschlossene Lösung:

$$T(n) = cn \log n + kn$$

Wir erkennen:

$T(n) = O(n \log n)$

Das ist besser als $O(n^2)$!

Das Verfahren "Teile und herrsche" hat also zu einem Algorithmus mit **geringerer Komplexität** geführt.

Entwicklung von T(n) (1)

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

$T(n) = 2T(n/2) + cn$

$= 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn$

$= 4(2T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn$

...

Behauptung

$T(n) = 2^i T(n/2^i) + icn$ (i ist die aktuelle Rekursionsstufe)

Entwicklung von T(n) (2)

Beweis durch vollständige Induktion über i:

i = 1:

$2^1T(n/2^1) + 1cn = 2T(n/2) + cn$

Schluss von i auf i + 1:

$T(n) = 2^iT(n/2^i) + icn$

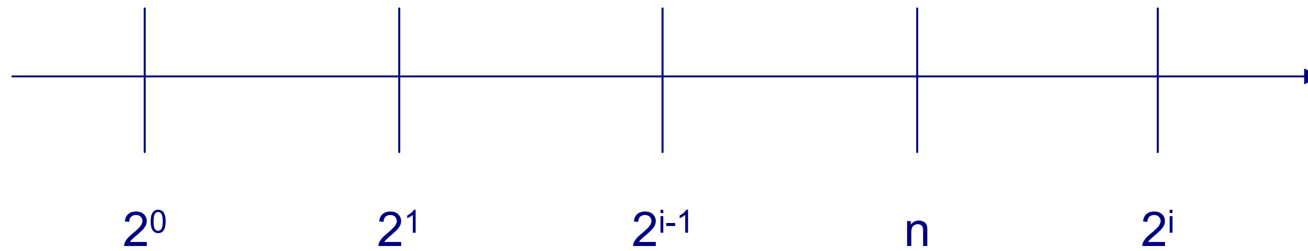
$= 2^i(2T(n/2^i+1) + cn/2^i) + icn$

$= 2^{i+1}T(n/2^{i+1}) + (i+1)cn$

$$T(n) = \begin{cases} k & \text{falls } n = 1 \\ 2^iT(n/2^i) + icn & \text{falls } n > 1 \text{ mit } i \geq 1 \end{cases}$$

Für Zweierpotenzen (1)

Betrachte nur solche n, die Zweierpotenzen sind:



Für Zweierpotenzen (2)

Denn da der Zeitaufwand des Algorithmus mindestens linear wächst ($\Omega(n)$), gilt für alle $n \leq n_2$:

$$T(n) \leq T(n_2)$$

mit $n_2 := 2^i$, so dass $2^{i-1} < n \leq 2^i = n_2$.

Dann gilt:

$$n = 2^i \qquad \text{bzw.} \qquad i = \log_2 n = \text{ld } n$$

Sei $n > 1$. Dann ist

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + icn \\ &= nT(1) + cn \text{ ld } n \\ &= nk + cn \text{ ld } n \\ &= O(n \text{ ld } n) \end{aligned}$$

"Teile-und-herrsche": Multiplikation (Beispiel 2) (1)

Multiplikation von vierstelligen Zahlen durch Zurückführung auf die Multiplikation von zweistelligen Zahlen.

Es sei X * Y zu berechnen.

Dann trenne man X und Y auf in Zahlen mit halber Länge.

Beispiel:

X = 1984; A = 19 B = 84

Y = 6713; C = 67 D = 13

"Teile-und-herrsche": Multiplikation (Beispiel 2) (2)

Nun gilt:

$X * Y = A * C * 10^4 + ((A + B) * (C + D) - AC - BD) * 10^2 + BD$

Die Zehnerpotenzen bedeuten nur ein Schieben der Zahlen nach links beim Addieren. Es sind nun nur noch drei Multiplikationen mit Zahlen halber Länge nötig:

$A * C$, $B * D$ und $(A + B) * (C + D)$.

Der Additionsaufwand ist proportional zu n, n = Anzahl der Ziffern.

Aufwand für die TH-Multiplikation

$T(n) = 3T(n/2) + cn$

$T(1) = k$

Lösung der Rekurrenzrelation ergibt:

$$T(n) = (2c + k)n^{ld\,3} - 2cn$$
$$= O(n^{ld\,3})$$
$$= O(n^{1,59})$$

Also ist auch hier der Aufwand geringer als $O(n^2)$ beim Schul-Algorithmus.

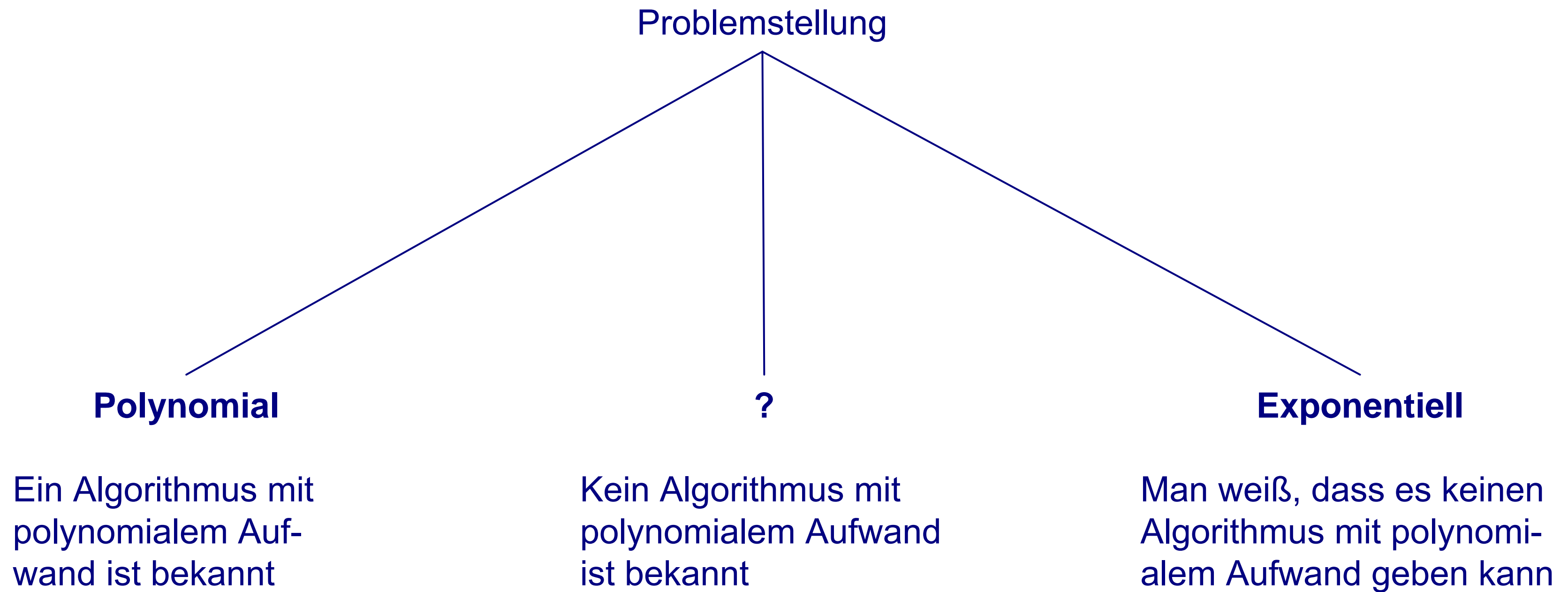
Polynomialer und exponentieller Aufwand

Eine gute Näherung für die Durchführbarkeit von Algorithmen ist die Annahme, dass

- Algorithmen mit polynomialem Aufwand ($O(n^c)$) durchführbar sind
- Algorithmen mit exponentiellem Aufwand ($O(c^n)$) nicht durchführbar sind

Daher ist es eine zentrale Fragestellung der Theoretischen Informatik, ob es für ein gegebenes Problem einen Algorithmus mit polynomialem Aufwand geben kann.

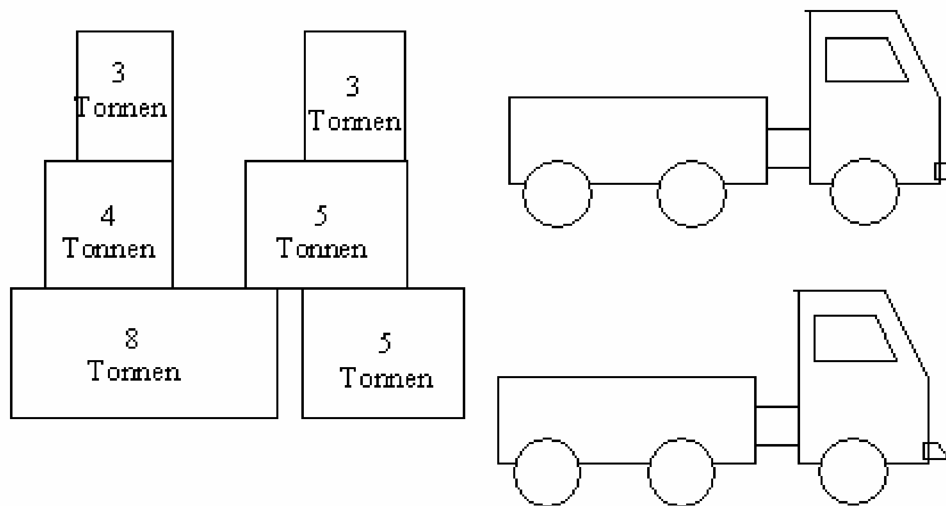
Klassifikation der Komplexität von Problemen



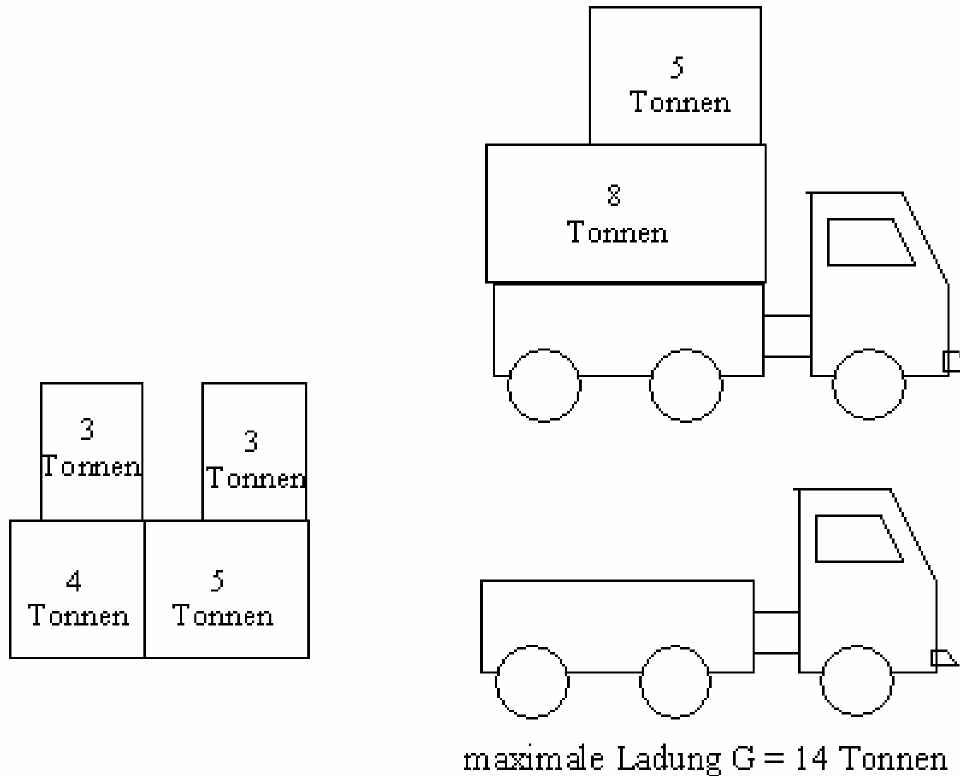
Besonders die Probleme aus der mittleren Klasse sind für Informatiker inter-essant!

Beispiel 1: Bin-packing Problem

Gegeben sind n Kästen mit unterschiedlichem Gewicht, T Lastwagen mit maximaler Zuladung G. Ist es möglich, alle Kästen zu verladen?



Algorithmus „Schwerster Kasten zuerst“ (1)



Nachdem ein Lastzug mit dem Algorithmus „Schwerster Kasten zuerst“ beladen wurde, müssen die verbleibenden Kästen nicht unbedingt auf den nächsten Wagen passen.

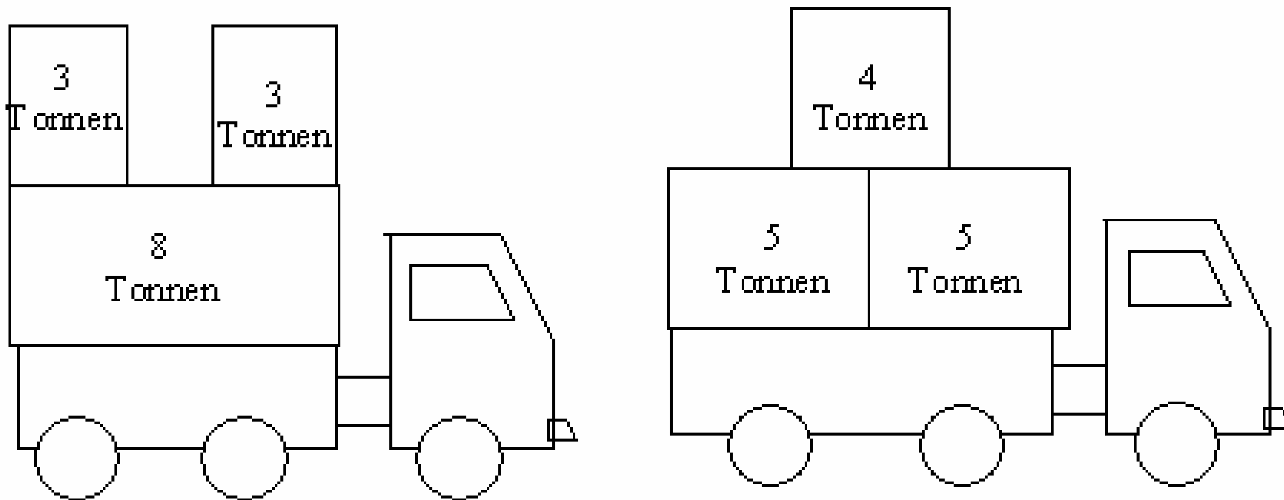
Algorithmus „Schwerster Kasten zuerst“ (2)

Führt nicht in allen Fällen zu einer Lösung!

Beachte:
Viele Probleme der Informatik sind analog zum bin-packing problem, z.B. das Laden von n Programmen unterschiedlicher Größe in den Speicher von T parallelen Prozessoren.

Lösung

Eine Lösung des Kastenproblems könnte folgendermaßen aussehen:



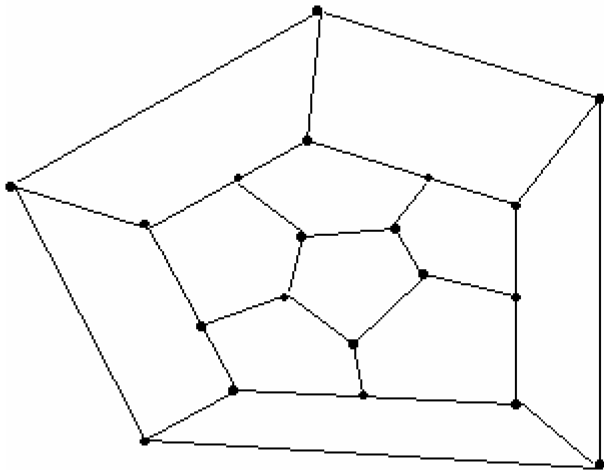
Beispiel 2: Travelling Salesman

Gegeben ist ein Straßennetz zwischen n Städten mit Entfernungen.

Kann ein Handelsreisender mit Kilometerbeschränkung eine Rundreise machen, bei der er jede Stadt **genau einmal** besucht?

Beispiel 3: Hamilton-Zyklus

Gegeben ist ein Straßennetz als Graph. Gibt es eine Rundreise, bei der jede Stadt **genau einmal** besucht wird?



Der einfache Fall eines Hamilton-Zyklus

- Knoten bedeuten Städte
- Linien bedeuten Straßen

Die Lösung heißt **Hamilton-Zyklus.**

Es ist kein Algorithmus mit polynomialem Aufwand bekannt!

Lösung zum Beispiel-Graphen

Eine Lösung zum Beispiel sieht folgendermaßen aus:

Beispiel 4: Stundenplanproblem

Gegeben ist eine Liste von Fächern, zu unterrichtende Schüler und ein Lehrplan (Anzahl der Stunden in jedem Fach in jedem Schuljahr).

Gibt es einen Stundenplan, bei dem kein Schüler eine Freistunde hat?

Antwort: Es ist kein Algorithmus mit polynomialem Aufwand bekannt!

Ansätze in der **Praxis**:

- Suche eine Näherungslösung statt der perfekten Lösung (z. B. Stundenplan mit kleinen Lücken).
- Suche einen Algorithmus, der für durchschnittliche Eingabefälle schnell arbeitet, selbst wenn der ungünstigste Fall exponentiell bleibt.

Die Klassen P und NP

Alle Probleme, die mit polynomialem Aufwand berechnet werden können, für die es also einen Algorithmus gibt, der durch ein Polynom nach oben beschränkt ist, bilden die **Klasse P**. Wie schon erwähnt, bedeutet das nicht immer, dass die Berechnung praktikabel ist, denn das Polynom kann sehr groß sein.

Alle Probleme, zu denen eine (wie auch immer ermittelte) **gegebene** Lösung in polynomialer Zeit **verifiziert** (als korrekt bewiesen) werden kann, bilden die **Klasse NP**.

Die Klasse P ist in der Klasse NP enthalten.

Die vorangegangenen vier Beispiele sind alle in NP (eine wie auch immer ermittelte Lösung ist mit polynomialem Aufwand verifizierbar).

Man weiß bis heute nicht, ob es Probleme gibt, die in NP, aber nicht in P sind!

NP-Vollständigkeit

Ein Problem heißt **NP-vollständig** (NP-complete), wenn gilt:

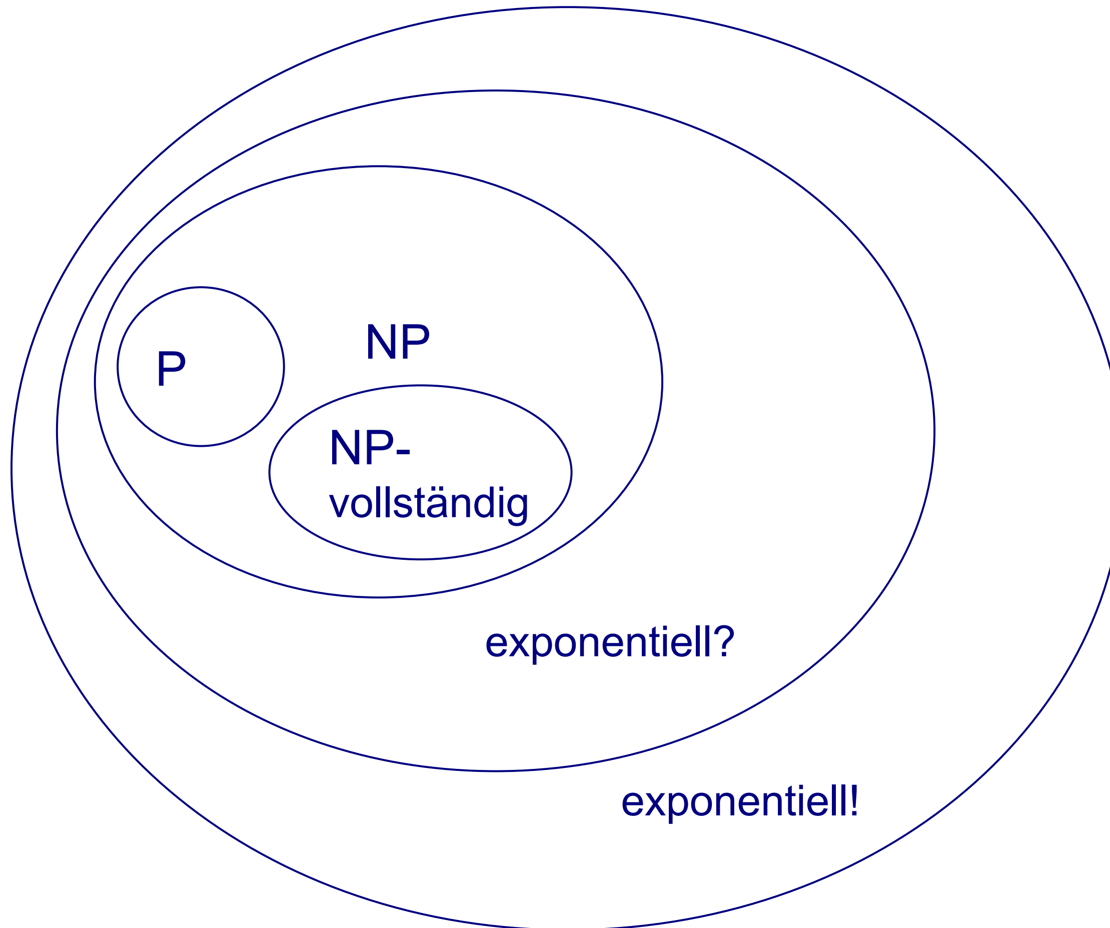
Wenn es für dieses Problem einen polynomialen Algorithmus gibt, dann gibt es für alle Probleme in NP einen polynomialen Algorithmus.

NP-vollständige Probleme sind die "schwersten" Probleme in NP.

Die Beispiele 1 bis 4 (bin-packing, travelling salesman, Hamilton-Zyklus, Stundenplan) sind NP-vollständig.

Viele Informatiker glauben, dass $P \neq NP$ ist, dass also P eine echte Teilmenge von NP ist.

Zusammenfassung Komplexität (1)



$$P \stackrel{?}{=} NP$$

Zusammenfassung Komplexität (2)