Einführung in die Informatik

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Sommersemester 2014

Teil XI

Komplexität

Komplexität

Arten von Komplexität:

- Zeitkomplexität: Wie lange dauert es, ein Programm auszuführen?
- Platzkomplexität: Wie viel Speicherplatz benötigt ein Programm bei seiner Ausführung?

Wir beschränken uns hier auf Zeitkomplexität.

Laufzeitbestimmung – Zeit stoppen

```
(..)
    long time = System.nanoTime();
    containsElement(l, i);
    System.out.println(System.nanoTime() - time);
(..)
static boolean containsElement(int[] l, int i) {
  for (int element : l)
    if (element == i)
      return true;
  return false;
}
```

Art/Größe d. Eingabe	1000	10000	100000	1000000	...
Element am Anfang					
Element in der Mitte					
Element am Ende					

Laufzeitbestimmung – Probleme

Probleme bei der „Tabellenmethode“:

- Tabelle ist groß und unhandlich
- Die Messwerte sind abhängig von Ausführungsumgebung, zB.
  - Hardware (vgl. Moore's Law)
  - Programmiersprache/Interpreter/Compiler/...
  - ...
- Maß für abstrakte Algorithmen gewünscht

Lösung:

- Abstrahiere Eingaben zu ihrer Größe/Länge
- Funktion von Eingabe nach Laufzeit
- Gib Laufzeit als „Anzahl der Ausführungschritte“ an
- Abstrahiere von der konkreten Schrittgröße

Verschiedene Analysen

Letzte Folie: Abstrahiere Eingaben zu ihrer Größe/Länge
Problem: unterschiedliche Eingaben gleicher Länge können zu unterschiedlichen Laufzeiten führen (zB. s.h. Tabelle/Experiment)
Verschiedene Analysen:

- Best Case Analyse
- Average Case Analyse
- Worst Case Analyse
- ...

Definition

Wir schreiben $f \in \mathcal{O}(g)$, wenn

$$\exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f(x)| \leq c \cdot |g(x)|$$

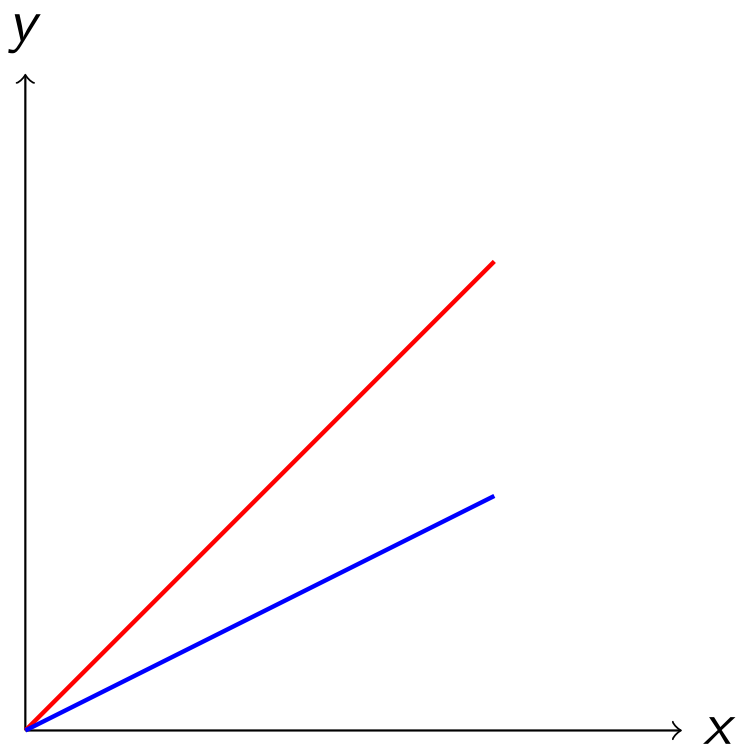
Intuitive Formulierung:
Ab einem bestimmten x-Wert und wenn man g beliebig mit einer Konstante multiplizieren kann, ist f immer durch g nach oben beschränkt.

Anmerkungen:

- Wir schreiben abkürzend $f \in \mathcal{O}(n)$ statt $f \in \mathcal{O}(g), g(n) = n$.
- Man schreibt manchmal auch $f = \mathcal{O}(g)$ statt $f \in \mathcal{O}(g)$.
- Es gibt auch entsprechende Definitionen für die untere Schranke.
- Wir versuchen immer die kleinste obere Schranke anzugeben.
- Man spricht auch von der *asymptotischen Laufzeit* eines Programms/Algorithmus'

Einige Funktionen

$$f \in \mathcal{O}(g) \text{ gdw. } \exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f(x)| \leq c \cdot |g(x)|$$

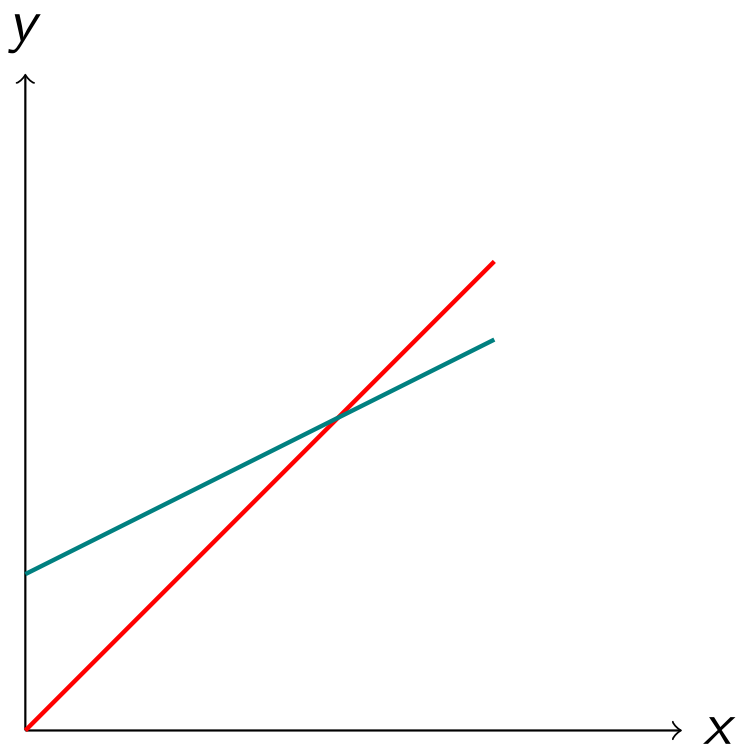


$g(x) = x$

$f_1(x) = \frac{1}{2}x$

Einige Funktionen

$$f \in \mathcal{O}(g) \text{ gdw. } \exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f(x)| \leq c \cdot |g(x)|$$

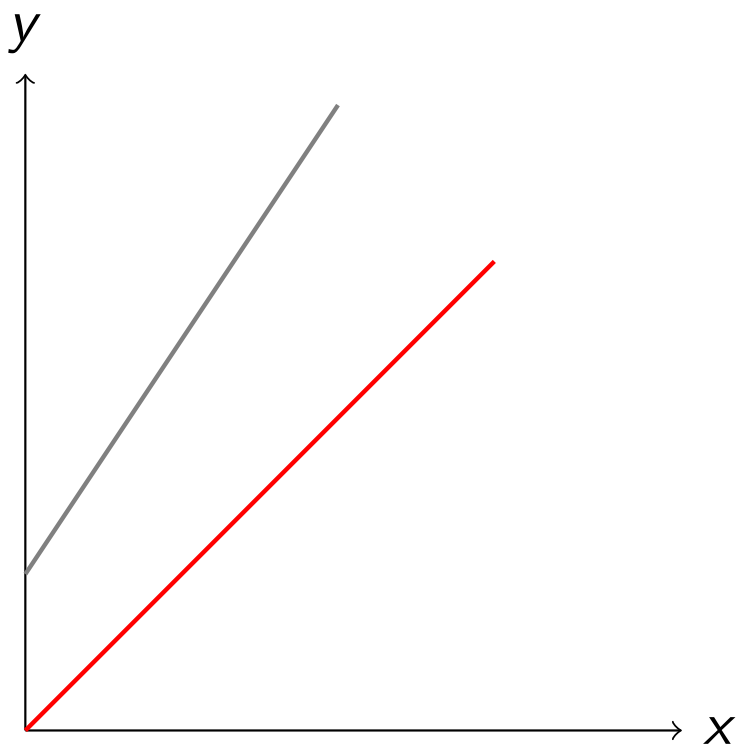


$$g(x) = x$$

$$f_2(x) = \frac{1}{2}x + 1$$

Einige Funktionen

$$f \in \mathcal{O}(g) \text{ gdw. } \exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f(x)| \leq c \cdot |g(x)|$$

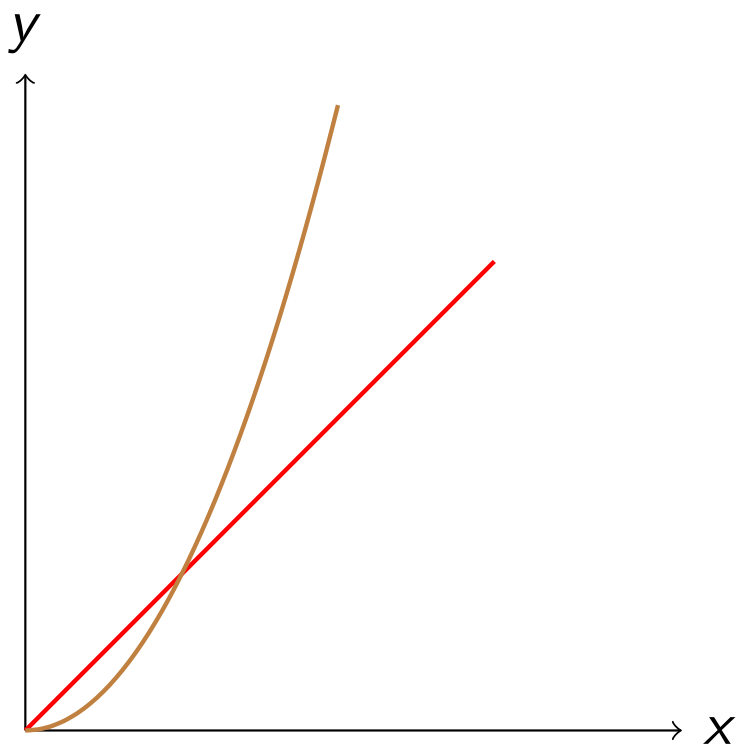


$$g(x) = x$$

$$f_3(x) = \frac{3}{2}x + 1$$

Einige Funktionen

$$f \in \mathcal{O}(g) \text{ gdw. } \exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f(x)| \leq c \cdot |g(x)|$$



$$g(x) = x$$

$$f_4(x) = x^2$$

Rechnung $f_3 \in \mathcal{O}(g)$

$$f_3(x) = \frac{3}{2}x + 1, g(x) = x$$

Zu zeigen:
$\exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f_3(x)| \leq c \cdot |g(x)|$

Wähle $c = 2$, dann ergibt sich als Schnittpunkt der beiden Geraden bei $x = 2$, wähle $\varepsilon = 2$.

Es gilt $\forall x > 2 . \frac{3}{2}x + 1 \leq 2x$.

□

Rechnung $f_4 \notin \mathcal{O}(g)$

$$f_4(x) = x^2, g(x) = x$$

Zu zeigen:

$$\neg(\exists c \in \mathbb{R}_+ . \exists \varepsilon \in \mathbb{N} . \forall x > \varepsilon : |f_4(x)| \leq c \cdot |g(x)|)$$
$$\Leftrightarrow \forall c \in \mathbb{R}_+ . \forall \varepsilon \in \mathbb{N} . \exists x > \varepsilon : |f_4(x)| > c \cdot |g(x)|$$

Wähle c und ε aus $\mathbb{R}_+$ ohne Einschränkung.

Dann ist zu zeigen:
Es existiert ein $x > \varepsilon$ sodass: $f(x) > c \cdot g(x)$, d.h. $x^2 > c \cdot x$.

Gesucht ist also ein $x \in \mathbb{R}_+$ sodass $x > c$ und $x > \varepsilon$.

So ein x existiert, z.B. $x = max(c, \varepsilon) + 1$.

□

Rechenbeispiele, Rechenregeln

Es gilt also zum Beispiel:

Sei $f(n) = m \cdot n + b$ für Konstanten m, b, dann: $f \in \mathcal{O}(n)$ und $f \notin \mathcal{O}(n^2)$

$\mathcal{O}(g) = \mathcal{O}(c \cdot g)$ für beliebige Konstante c, und beliebige Funktion g

$\mathcal{O}(\log_2 n) = \mathcal{O}(\log_c n)$ für beliebige Konstanten c

$f \in \mathcal{O}(n^k)$, falls f ein Polynom vom Grad k ist.

Rechenregeln:

$\mathcal{O}(f) + \mathcal{O}(g) = max(\mathcal{O}(f), \mathcal{O}(g))$

$\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$

Gängige Komplexitätsklassen

Klasse	Umgangssprachlich	Beispielprobleme/Anmerkungen
$\mathcal{O}(1)$	Konstantzeit	erstes Element einer Liste ausgeben
$\mathcal{O}(\log(n))$	logarithmische Zeit	in einer sortierten Liste suchen
$\mathcal{O}(n)$	Linearzeit	in unsortierter Liste suchen
$\mathcal{O}(n \cdot log(n))$	„n-log-n“	Liste sortieren
$\mathcal{O}(n^2)$	quadratische Zeit	Zwei Summen ausmultiplizieren
$\mathcal{O}(n^3)$	kubische Zeit	Matrizenmultiplikation
$\mathcal{O}(n^k)$	Polynomialzeit	(jedes k ergibt eine eigene Ordnung)
$\mathcal{O}(k^n)$	Exponentialzeit	Potenzmenge berechnen
	exotischere Klassen	
$\mathcal{O}(n!)$	Alle möglichen Reihenfolgen n Städte zu besuchen.	
$\mathcal{O}(n^n)$		
...		

Beispiel: Element in unsortierter Liste suchen

Gegeben: Liste l von `int`s
Gefragt: Ist `int` i in l enthalten?

```
boolean containsElement(int[] l, int i) {
  for (int element : l) {
    if (element == i)
      return true;
  }
  return false;
}
```

Was können wir über die Laufzeitfunktion $f(n)$ in Abhängigkeit von `n=l.length` aussagen?
Best Case: $f(n) = 1 \in \mathcal{O}(1)$
Average Case: $f(n) = n/2 \in \mathcal{O}(n)$
Worst Case: $f(n) = n \in \mathcal{O}(n)$

Beispiel: In sortierter Liste suchen (binary search)

Gegeben: Sortiertes Array `l`, Zahl `i`, Array-Indizes `index1`, `index2`

Gefragt: Ist `i` in `l` zwischen den Indizes `index1` und `index2` enthalten?

```
boolean contains(int[] l, int i, int index1, int
    index2) {
  if (index1 > index2)
    return false
  int mid = (index2 + index1)/2;
  if (l[mid] == i)
    return true;
  else if (i < l[mid])
    return contains(l, i, index1, mid - 1);
  else
    return contains(l, i, mid + 1, index2);
}
```

Laufzeitanalyse binary search - Beispieldurchlauf

Input: sortiertes Array l = [1, 3, 14, 47, 94, 234, 384, 833, 902] gesuchtes Element: 234

index1 = 0, index2 = 8, mid = 4, l[mid] = 94

$234 > 94 \implies$ betrachte nur noch die rechte Hälfte

index1 = 5, index2 = 8, mid = 6, l[mid] = 384

$234 < 384 \implies$ betrachte nur noch die linke Hälfte

index1 = 5, index2 = 6, mid = 5, l[mid] = 234

$234 == 234 \implies$ liefere `true` zurück

Best Case (das gesuchte Element ist in der Mitte der Liste): $\mathcal{O}(1)$
Worst Case: $\mathcal{O}(\log n)$

Beispiel: Methodenaufrufe

```
static int mult(int m, int n) {
  int result = 0;
  for (int i = 0; i < n; i++)
    result = plus(result, m);
  return result;
}
static int plus(int m, int n) {
  int result = m;
  for (int i = 0; i < n; i++)
    result = inc(result);
  return result;
}
static int inc(int n) {
  return ++n;
}
```

Beispiel: Fibonacci

```
long fiboRek(int n) {
  if (n == 0 || n == 1)
    return n;
  return fiboRek(n - 1) + fiboRek(n - 2);
}
long fiboIt(int n) {
  if (n == 0 || n == 1)
    return n;
  long curNMinus1 = 1, curNMinus2 = 1, curN = 0;
  for (int i = 2; i < n ; i++) {
    curN = curNMinusOne + curNMinusTwo;
    curNMinusTwo = curNMinusOne;
    curNMinusOne = curN;
  }
  return curN;
}
```

Zusammenfassung, Caveats, $\mathcal{O}$-Notation

Die $\mathcal{O}$-Notation

- ist der übliche Weg, um in der Informatik die Komplexität eines Algorithmus' oder eines Problems anzugeben.
- ignoriert multiplikative Konstanten und betrachtet die Laufzeit asymptotisch, d.h. für Eingabegrößen, die gegen ∞ gehen.
- ist nützlich, um eine grobe, plattformunabhängige Laufzeitabschätzung für ein Programm zu geben.
- ist eine Grundlage der Komplexitätstheorie
- ist aber *nicht* immer aussagekräftig im Hinblick auf das tatsächliche Laufzeitverhalten von Programmen – Konstanten können groß werden.

Problemklassen

Im Folgenden betrachten Klassen von Problemen. Gängige Probleme sind zum Beispiel:

1. Enthält eine Sortierte Liste ein bestimmtes Element?
2. Sortiere diese Liste.
3. Gib die kürzeste Route an, die diese Städte besucht.
4. Gibt es eine Route, die diese Städte besucht, und die kürzer als x km ist?
5. Entspricht dieses Program dieser Spezifikation?
6. Läuft dieses Programm unendlich lange?

Probleme 1, 4, 5 und 6 sind *Entscheidungsprobleme*, d.h. Probleme, die ja oder nein zur Antwort haben.

Entscheidungsprobleme

Entscheidunsprobleme sind solche, die „ja“ oder „nein“ zur Antwort haben.

Man kann ein Entscheidungsproblem somit als Menge aller seiner Inputs schreiben, für die die Antwort „ja“ ist.

Zum Beispiel sei das Problem „Gegeben eine Ganzzahl zwischen 0 und 10, ist diese Zahl gerade?“, wir nennen es im Folgenden X.

Dann ist seine Eingabemenge $\mathit{Input}(X) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.
Und die Menge, die durch das Problem gegeben ist, ist:
$\{x \in \mathit{Input}(X) \mid x \text{ ist gerade}\} = \{0, 2, 4, 6, 8, 10\}$

Die Klassen P und NP

Falls die Laufzeit eines Programms in $\mathcal{O}(n^k)$ liegt, für ein $k \in \mathbb{N}$, sagen wir, es läuft in *Polynomzeit*.

P: Alle Entscheidungsprobleme, die in Polynomzeit lösbar sind.
Beispiele:

NP: Alle Entscheidungsprobleme, bei denen eine -Lösung- in Polynomzeit verifiziert werden kann.
Also Entscheidungsprobleme von der Form „Hat ... eine Lösung“, bei denen man, wenn man einen Lösungsvorschlag hat, in Polynomzeit prüfen kann, ob dieser eine tatsächliche Lösung ist.
Beispiele:

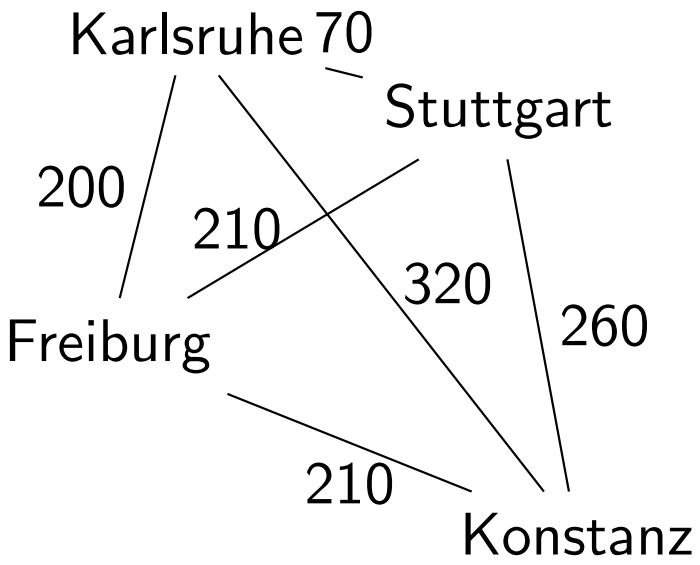
Das Handlungsreisendenproblem (TSP) (als Entscheidungsproblem)

Gegeben: Ein vollständiger Graph mit gewichteten Kanten $G = (V, E, w), E \subseteq V \times V, w : E \to \mathbb{N}$, der Städte und ihre Abstände untereinander (in km) darstellt, sowie eine Kilometerzahl k, die angibt, wie weit man maximal zu fahren bereit ist.

Frage: Gibt es eine Reihenfolge, in der man alle Städte nacheinander abfahren kann, und am Ausgangspunkt ankommen, ohne die gegebene Kilometerzahl zu überschreiten?

Das Handlungsreisendenproblem als Menge:
$\{(G, k) \mid G$ ist ein gewicheteter Graph, k eine Zahl, und G hat eine Rundreise deren Kantensumme kleiner gleich k ist$\}$

Das Handlungsreisendenproblem (TSP) – Beispiel



$k = 750$

TSP $\in$ NP

Behauptung: Das Handlungsreisendenproblem ist in NP.

Nachweis:
Raten: Rate eine Reihenfolge, in der die Städte abgefahren werden sollen.
Verifizieren: Summiere die entsprechenden Kantengewichte und vergleiche die Summe mit k.

Tatsächlich ist TSP sogar NP-vollständig.

Definition

Ein Problem ist *NP-vollständig*, wenn

1. es in NP liegt
2. es *NP-schwierig* ist, d.h. sich jedes Problem in NP darauf in Polynomzeit reduzieren lässt (s. nächste Folie)

Polynomzeitreduktion

UNI FREIBURG

Definition

Seien X, Y zwei Probleme (formuliert als Mengen). Wir sagen X *lässt sich auf Y in Polynomzeit reduzieren*, wenn es eine in Polynomzeit berechenbare Funktion $f : \mathit{Input}(X) \to \mathit{Input}(Y)$ gibt, so dass für jedes $x \in \mathit{Input}(X)$ gilt:

$x \in X$ gdw. $f(x) \in Y$

Wir schreiben dann $X \leq_P Y$.

Wesentliche Folgerung:

Falls gilt $X \leq_P Y$ und Y in P, dann ist auch X in P.

NP-vollständige Probleme

SAT $\{\phi \mid \phi$ist eine erfüllbare aussagenlogische Formel$\}$

HAMPATH $\{G \mid G$ ist ein Graph, in dem es einen Pfad gibt, der jeden Knoten genau einmal besucht$\}$

Graph Färbung $\{(G, k) \mid G = (V, E)$ ist ein Graph, der mit k Farben, so gefärbt werden kann (die Knoten), dass benachbarte Knoten nie die selbe Farbe haben$\}$

.. und viele mehr

P vs. NP

Stand der Forschung:

Man weiß nicht sicher, ob P und NP zusammenfallen, oder nicht.

Bisher sind alle Beweisversuche für die Ungleichheit gescheitert.

Es hat aber auch niemand einen Polynomzeitalgorithmus für ein NP-vollständiges Problem gefunden.

Die Frage ist eines der Milleniumsprobleme.

Falls P=NP: möglicherweise technische Revolution, bisher starke Kryptografie wird schwach, ...

Falls P!=NP: Es würde sich nicht viel ändern, die meisten rechnen damit jetzt schon.

Ausblick: Berechenbarkeitstheorie

Die NP-vollständigen Probleme sind keineswegs die schwierigsten Probleme der Informatik.
Es gibt Probleme, die sicher exponentielle Zeit brauchen.
Es gibt Probleme, die überhaupt nicht lösbar sind.
Die Berechenbarkeitstheorie fragt nicht nach der Komplexität eines Problems, sondern ob es überhaupt in endlicher Zeit von einem Computer lösbar ist.

Zusammenfassung, P vs. NP

- NP ist die Klasse von Problemen, die in Polynomzeit (d.h. meist effizient) verifizierbar sind.
- NP-vollständige Probleme sind solche, die in NP liegen, und die NP-schwierig sind.
- Kann man ein NP-schwieriges Problem in Polynomzeit lösen, so kann man alle Probleme in NP in Polynomzeit lösen, somit P=NP.