Algorithmen und Datenstrukturen

§1 Komplexitätsanalyse von Algorithmen

Prof. Dr. Georg Umlauf

H T W G
Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Inhalt der Vorlesung

1. Grundbegriffe
2. Motivation
3. Komplexitätstheorie
4. Abschätzung der Laufzeit von Algorithmen
5. NP-Probleme

- Mathematisches Glossar

§1.1 Grundbegriffe

Algorithmus

- Was ist ein Algorithmus?
- Wofür benötigt man einen Algorithmus?

Datenstruktur

- Was ist eine Datenstruktur?
- Wofür benötigt man eine Datenstruktur?

Definition:

Ein **Algorithmus** ist ein Verfahren zur Lösung eines Problems in endlich vielen Schritten.

Statue Al-Chwarizmis, TU Teheran. Quelle: wikipedia

- Beispiele:
  - Kochrezept, Bastelanleitung, Ikea-Bauanleitungen,
  - Sortieren von Datensätzen, Finden kürzester Wege, etc.
- Ein Algorithmus in der Informatik manipuliert Daten.

- Diese Algorithmen-Definition ist unabhängig von seiner
  - Implementierung in einer bestimmten Programmiersprache
  - auf einem bestimmten Rechner.

Definition:

Eine **Datenstruktur** ist eine Methode zur Strukturierung der von den Algorithmen manipulierten Daten.

- Beispiele:
  - Organisation der Zutaten für ein Kochrezept,
  - Arrays/Listen in C/C++/C#, etc.

- Diese Datenstruktur-Definition ist unabhängig von seiner
  - Implementierung in einer bestimmten Programmiersprache
  - auf einem bestimmten Rechner.

§1.2 Motivation

Effizienz ist eine wesentliche Eigenschaft von Algorithmen

- Für ein und dasselbe Problem gibt es mehrere Algorithmen.
- Die Effizienz dieser Algorithmen kann sehr unterschiedlich sein.

- Beurteilung der Effizienz sollte unabhängig sein von
  - der Implementierung und
  - dem verwendeten Rechner.

Algorithmenanalyse

- Mathematische Untersuchung zur Leistung eines Algorithmus basierend auf einem standardisierten **Berechnungsmodell** (§1.3).
- Komplexitätstheorie als Zweig der Algorithmenanalyse.

§1.2 Motivation

Es gibt verschiedene Algorithmen zur Lösung desselben Problems:

Mögliche Unterschiede

- Laufzeit
  - Die Laufzeitfunktion sollte möglichst langsam wachsen.
  - Evtl. abhängig von der Größe des Resultats (output-size-sensitive).
- Speicherplatzbedarf
- Zugriffe auf Sekundärspeicher.
- Kommunikationsaufwand.
- Implementierungsaufwand
  - Es gibt schnelle Algorithmen, die nicht verwendet werden, weil sie zu kompliziert zu implementieren sind.

Eigenschaften von Algorithmen

Allgemeinheit
- Lösung für Problemklasse, nicht für Einzelaufgabe.

Korrektheit
- Algorithmus berechnet stets die spezifizierte Ausgabe, falls er terminiert.

Determiniertheit
- Für die gleiche Eingabe wird stets die gleiche Ausgabe berechnet, aber andere Zwischenzustände sind möglich.
- Bsp. nicht-determiniert: $\mathrm{abs}(x) = -x$ für $x \leq 0$, x für $x \geq 0$.

Determinismus
- Für die gleiche Eingabe ist die Ausführung stets identisch.
- Bsp. nicht-deterministisch: Quick-Sort liefert immer eine sortierte Liste zurück, durch die Wahl eines zufälligen Pivot-Elementes aber u.U. mit verschiedenen Zwischenschritten.

Terminierung
- Der Algorithmus läuft für jede Eingabe nur endlich lange.

Effizienz
- Sparsamkeit im Ressourcenverbrauch, z.B. Operationen, Zeit, Speicher, Energie, …

Komplexitätsmaß - Effizienz

- Zeitaufwand (kurz: **Laufzeit**, engl.: **runtime**)
  - Achtung: „Zeit“ nicht wörtlich nehmen!
- Speicherplatzbedarf
- (Implementierungsaufwand)

Problemgröße

- Die meisten Algorithmen haben einen kritischen Parameter (Problemgröße), der die Laufzeit am stärksten beeinflusst.
- Beispiele:
  - Suchalgorithmus: Anzahl der zu durchsuchende Elemente
  - Sortieralgorithmus: Anzahl der zu sortierende Elemente
  - Lösen eines LGS: Anzahl Zeilen und Spalten der Matrix

Berechnungsmodell

- Abzählbar unendlich viele Speicherzellen,
  - die über natürliche Zahlen adressierbar sind,
  - beliebige ganze oder reelle Zahlen speichern können und
  - deren Zugriffszeit als vernachlässigbar angesehen werden.
    - **Achtung:** Dies ignoriert Festplattenzugriffe: Alle Daten sind im Hauptspeicher.
- **Primitive Operationen** haben konstante Laufzeit und einen konstanten Speicherplatzbedarf:
  - arithmetische Operationen ($+$, $-$, $\times$, $/$, mod, div) und Zuweisungen ($=$),
  - Vergleichsoperationen ganzer/reeller Zahlen ($<$, $>$, $==$, $!=$, $<=$, $>=$, etc.),
  - indirekte Adressierung,
  - Wurzelfunktionen, trigonometrische Funktionen, Logarithmen, etc.

Bemerkung

- Das Berechnungsmodell wird bei einigen Algorithmen noch weiter vereinfacht werden, indem nur noch einige **spezielle Operationen als relevant** betrachtet werden.
- **Beispiel**
  - Bei Sortieralgorithmen werden oft nur Vergleichs- und/oder Zuweisungs-operationen gezählt.
  - Bei einigen Suchalgorithmen werden gelegentlich nur Speicherzugriffe gezählt.
  - Bei einigen Graphen-Algorithmen werden nur die verfolgten Kanten gezählt.
- **Vorteil:** Vereinfachung der Beurteilung
- **Nachteil:** Ungenauer, i.d.R. um einen konstanten Faktor zu klein.

Definition:

Die **Laufzeit** eines Algorithmus ist die Anzahl der zur Berechnung eines Resultats nötigen primitiven Operationen.

- „Eine primitive Operation entspricht einem Zeitschritt oder Befehlszyklus o.Ä.“
- Die Laufzeit hängt in der Regel ab von
  - der Größe n der Eingabe und
  - ggf. noch weiteren Parametern.
- Die Laufzeit ist i.A. eine Funktion $T(n)$ der Eingabegröße n, z.B.
  - $T(n) = a \cdot n^2 + b \cdot \sqrt{n} + c \cdot \log n$ mit Konstante a, b, c.
- **Ausführungszeit:** Tatsächlich gemessene Zeit einer konkreten Implementierung auf einer konkreten Hardware gemessen in Sekunden.

Beispielalgorithmus

- Gegeben: Array `a[]` von n verschiedenen ganzen Zahlen.
- Gesucht: Index `j` des Elementes, das die größte Zahl im Array enthält.

```
int max_element(int[] a)
{
   int j =   0 ;        // speichert Index des Maximums
   int x = a[0];        // speichert maximalen Wert in a[0..i]
   for (int i = 1; i < a.length; i++) {
      if (a[i] > x){
         j =   i ;
         x = a[i];
      }
   }
   return j;
}
```

Beispielalgorithmus

- Gegeben: Array `a[]` von n verschiedenen Werten mit >-Operator.
- Gesucht: Index `j` des Elementes, das den größten Wert im Array enthält.

```
int max_element(KeyType[] a)
{
   int      j =   0 ;  // speichert Index des Maximums
   KeyType  x = a[0];  // speichert maximalen Wert in a[0..i]
   for (int i = 1; i < a.length; i++) {
      if (a[i] > x){
         j =   i ;
         x = a[i];
      }
   }
   return j;
}
```

Nur Zuweisungen auf Variablen vom Datentyp `KeyType` werden gezählt.

Wie oft bekommt `x` pro Durchlauf einen Wert zugewiesen?

- Best case
  - `a[0]` ist die größte Zahl, d.h. es gibt nur eine Zuweisung.
- Worst case
  - Zahlenfolge ist in aufsteigender Reihenfolge angeordnet, d.h. es werden n Zuweisungen auf `x` durchgeführt.
- Average case
  - Irgendwo zwischen 1 und n Zuweisungen.

§1.3 Komplexitätstheorie
Ein Beispielalgorithmus

- Anzahl an Zuweisungen für **x** für $n = 3$:
- Jede Anordnung ist gleich wahrscheinlich:

$$T(3) = (3 + 2 + 2 + 1 + 2 + 1)/6$$
$$= 11/6$$

- Anzahl der Zuweisungen an **x** für beliebiges n

$$T(n) = \sum_{k}^{n} \frac{k \cdot s_k}{n!}.$$

Relative Anordnung	Anzahl Zuweisungen an **x**
`a[0] < a[1] < a[2]`	3
`a[0] < a[2] < a[1]`	2
`a[1] < a[0] < a[2]`	2
`a[1] < a[2] < a[0]`	1
`a[2] < a[0] < a[1]`	2
`a[2] < a[1] < a[0]`	1

- Noch nicht explizit aufgelöst ist dabei die Anzahl der Permutationen s_k, für die das Maximum bei **`a[k]`** ist.

Da die Laufzeitfunktion oft sehr kompliziert ist, gibt es verschiedene Vereinfachungen:

- Ungünstigster Fall (**worst case**): Maximalwert von $T(n)$
- Günstigster Fall (**best case**): Minimalwert von $T(n)$
- Normalfall (**average case**): Erwartungswert von $T(n)$
- ▶ Asymptotische Ordnung (§1.4): „Führende Terme“ von $T(n)$

- *Worst case*, *best case* und *average case* haben i.d.R. unterschiedliche Laufzeiten.
- Berechnung des *average case* ist oft am schwierigsten:
  - Daher häufig Beschränkung auf *worst case*.
  - **Achtung:** *Worst case* liefert aber oft sehr/zu pessimistisch Werte.

Definition:

Die **Komplexität** eines Problems ist die Laufzeit des besten Algorithmus zur Lösung dieses Problems.

- Der beste Algorithmus hat die am langsamsten wachsende Laufzeitfunktion.

Achtung

Oft wird nicht klar unterschieden zwischen Laufzeit und Komplexität.

- Der Begriff „Komplexität“ wird oft auch für die „Laufzeit“ verwendet.

Ausführungszeit, Laufzeit und Komplexität nicht verwechseln!

Das O-Kalkül (Landau-Symbole)

- Laufzeitfunktionen können sehr kompliziert sein.
- Zur Bestimmung der Komplexität eines Problems müssen Laufzeitfunktionen bewertet und verglichen werden.
- ▶ Entscheidend ist das asymptotische Wachstum einer Funktion.
- Idee: Vergleich des Wachstums zweier Funktionen

  $g(n)$ und $f(n)$ für große n, d.h. $n \geq n_0$.

  - „Welche Funktion gewinnt das Rennen?“
- Vorteile:
  - Konstante Faktoren sind irrelevant.
  - Bei Polynomen zählt nur der größte Exponent.

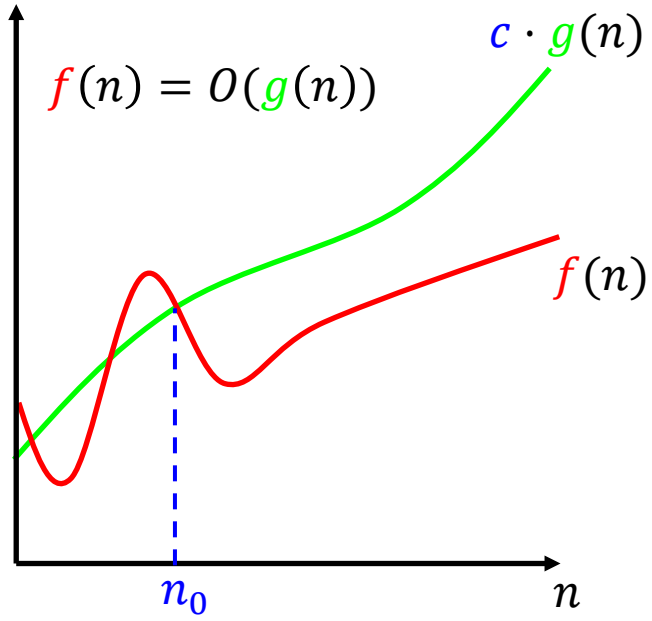
Edmund Landau, 1877-1938.
Quelle: Wikipedia

Definition: (sprich: „Groß-Oh“)

$O(g(n)) = \{f(n)$: Es gibt positive Konstanten c und n_0, so dass

$$0 \leq |f(n)| \leq c\,|g(n)| \text{ für alle } n \geq n_0\}.$$

- Schreibweise: $f(n) = O(g(n))$ statt $f(n) \in O(g(n))$.
- Beispiele:
  - $3n \quad = O(n^2)$
  - $\frac{n^2}{2} - 3n = O(n^2)$
  - $6n^3 \quad \neq O(n^2)$
- $f(n) = O(g(n))$ entspricht „$a \leq b$“.

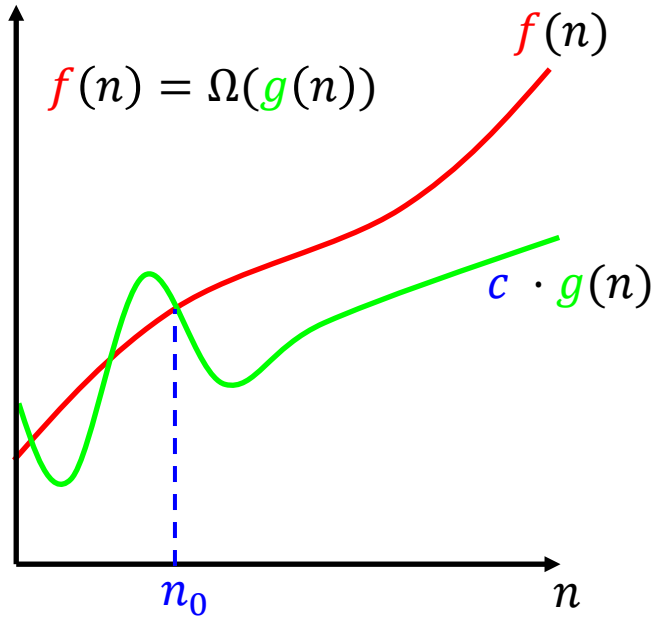


Definition: (sprich: „Groß-Omega“)

$\Omega(g(n)) = \{f(n)$: Es gibt positive Konstanten c und n_0, so dass

$0 \leq c\,|g(n)| \leq |f(n)|$ für alle $n \geq n_0\}$.

- Beispiele:
  - $3n \quad \neq \Omega(n^2)$
  - $\frac{n^2}{2} - 3n = \Omega(n^2)$
  - $6n^3 \quad = \Omega(n^2)$
- $f(n) = \Omega(g(n))$ entspricht „$a \geq b$“.

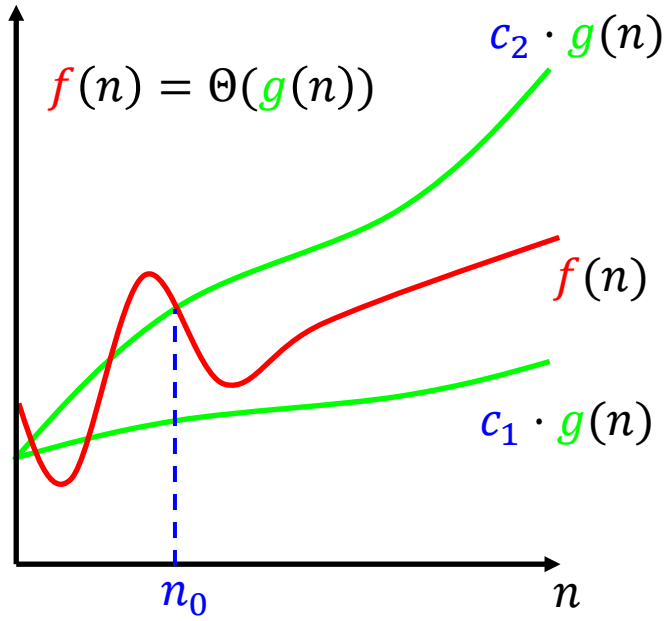


Definition: (sprich: „Groß-Theta“)

$\Theta(g(n)) = \{f(n)$: Es gibt positive Konstanten c_1, c_2 und n_0, so dass

$0 \leq c_1 \, |g(n)| \leq |f(n)| \leq c_2 \, |g(n)|$ für alle $n \geq n_0\}$.

- Beispiele:
  - $3n \quad \neq \Theta(n^2)$
  - $\frac{n^2}{2} - 3n = \Theta(n^2)$
  - $6n^3 \quad \neq \Theta(n^2)$
- $f(n) = \Theta(g(n))$ entspricht „$a = b$“.



Definition: (sprich: „Klein-Oh“)

$o(g(n)) = \{f(n)$: Für alle Konstanten $c > 0$ gibt es ein $n_0 > 0$, so dass

$$0 \leq |f(n)| < c\,|g(n)| \text{ für alle } n \geq n_0\}.$$

- Beispiele:
  - $3n \quad = o(n^2)$
  - $\frac{n^2}{2} - 3n \neq o(n^2)$
  - $6n^3 \quad \neq o(n^2)$
- $f(n) = o(g(n))$ entspricht „$a < b$“.
- Äquivalente Bedingung für $f(n) = o(g(n))$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Definition: (sprich: „Klein-Omega“)

$\omega(g(n)) = \{f(n)$: Für alle Konstanten $c > 0$ gibt es ein $n_0 > 0$, so dass

$$0 \leq c\,|g(n)| < |f(n)| \text{ für alle } n \geq n_0\}.$$

- Beispiele:
  - $3n \quad \neq \omega(n^2)$
  - $\frac{n^2}{2} - 3n \neq \omega(n^2)$
  - $6n^3 \quad = \omega(n^2)$
- $f(n) = \omega(g(n))$ entspricht „$a > b$“.
- Äquivalente Bedingung für $f(n) = \omega(g(n))$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

Zusammenfassung:

- $O(g) \cap \Omega(g) = \Theta(g)$
- $O(g) \setminus \Theta(g) = o(g)$
- $\Omega(g) \setminus \Theta(g) = \omega(g)$
- $o(g) \subset O(g), \Theta(g) \subset O(g)$
- $\omega(g) \subset \Omega(g), \Theta(g) \subset \Omega(g)$
- $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \Leftrightarrow \quad f = o(g) \quad \Leftrightarrow \quad g = \omega(f) \\ \infty & \Leftrightarrow \quad f = \omega(g) \quad \Leftrightarrow \quad g = o\,(f) \\ \text{const} \neq 0, \infty & \Leftrightarrow \quad f = \Theta(g) \quad \Leftrightarrow \quad g = \Theta(f) \end{cases}$$

- **Faustregel 1:** Konstante Faktoren spielen keine Rolle:
  - Eine quadratische Funktion wird mit keinem noch so großen konstanten Faktor zu einer exponentiellen Funktion.
  - Die Basis von Logarithmen kann weg gelassen werden:

$$\log_a(n) = c \log_b(n) \text{ mit } c = \log_a b.$$

- **Faustregel 2:** Bei Polynomen und Potenzen zählt nur der größte Exponent.

- **Aber:** Wahl einer sinnvollen Charakterisierung der Komplexität/Laufzeit:
  - $kn^2 = O(2^n)$, aber auch $kn^2 = O(n^2)$.

§1.3 Komplexitätstheorie

Das O-Kalkül

$O(1)$	Die Operationen werden konstant oft ausgeführt.
$O(\log n)$	Anzahl der Operationen wächst langsamer als n.
$O(n)$	Verdoppelung von n führt zur Verdoppelung des Aufwands.
$O(n \log n)$	Bei Algorithmen, die Problem in kleinere Teilprobleme zerlegen (teile-und-herrsche, divide-and-conquer).
$O(n^2)$	Meist Bearbeitung von Paaren von Eingabeelementen. Verdoppelung von n führt zur Vervierfachung des Aufwands.
$O(n^3)$	Meist Bearbeitung von Tripeln von Eingabeelementen. Verdoppelung von n führt zur Verachtfachung des Aufwands.
$O(2^n)$	Algorithmus kommt in der Praxis meist nicht in Frage.

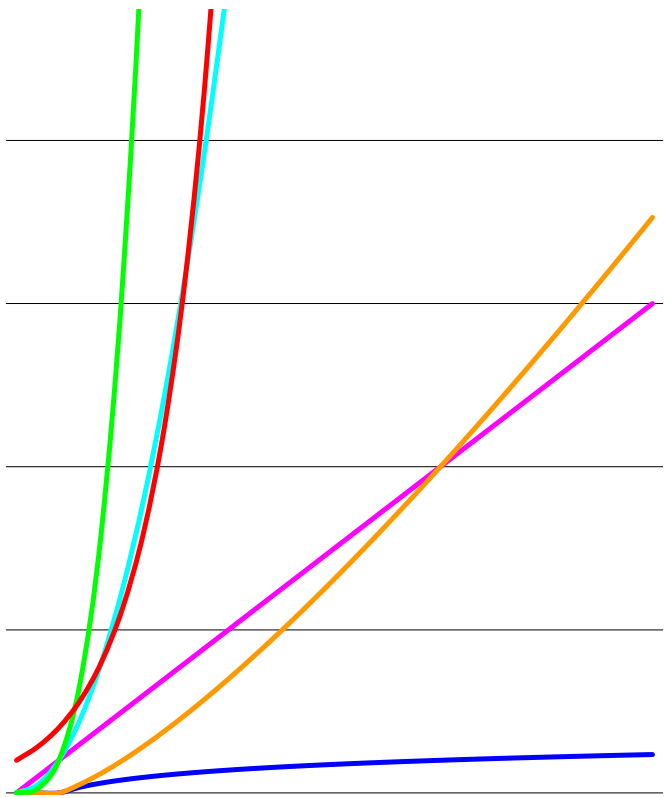
§1.3 Komplexitätstheorie

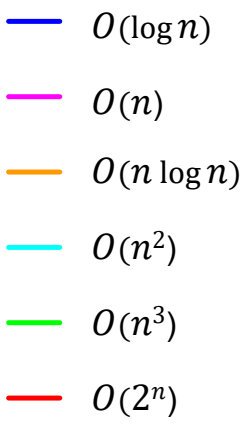
Vergleich von Größenordnungen

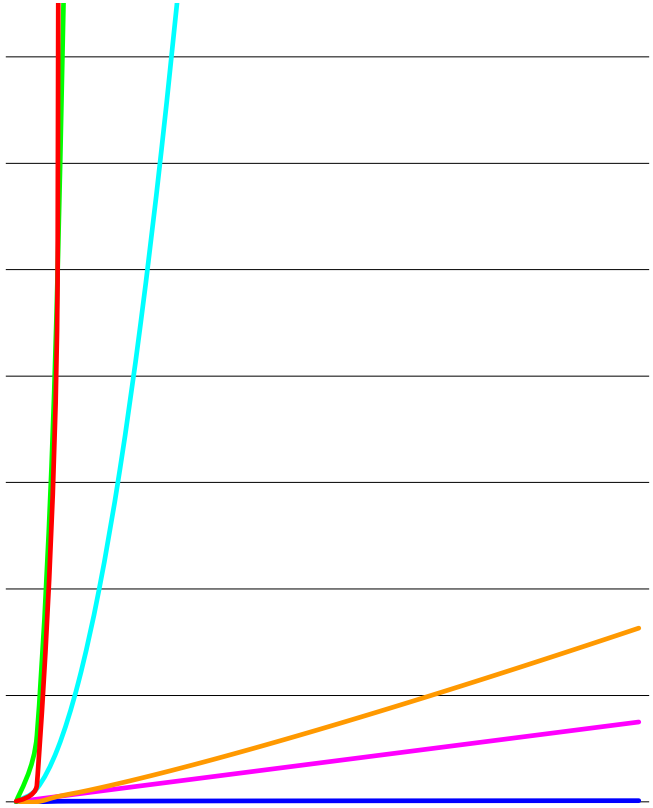
$\log_{10} n$	$\log_2 n$	n	$n \log_2 n$	n^2
1	$3 \approx 10^0$	$10 = 10^1$	$30 \approx 10^1$	$100 = 10^2$
2	$6 \approx 10^1$	$100 = 10^2$	$600 \approx 10^2$	$10.000 = 10^4$
3	$9 \approx 10^1$	$1.000 = 10^3$	$9.000 \approx 10^4$	$1.000.000 = 10^6$
4	$13 \approx 10^1$	$10.000 = 10^4$	$130.000 \approx 10^5$	$100.000.000 = 10^8$
5	$16 \approx 10^1$	$100.000 = 10^5$	$1.600.000 \approx 10^6$	$10.000.000.000 = 10^{10}$
6	$19 \approx 10^1$	$1.000.000 = 10^6$	$19.000.000 \approx 10^7$	$1.000.000.000.000 = 10^{12}$

§1.3 Komplexitätstheorie

Vergleich von Größenordnungen







$n = 0, \ldots, 14$

$n = 0, \ldots, 150$

§1.3 Komplexitätstheorie

Vergleich von Größenordnungen

Operationen je Sekunde	Problemgröße: 10^6			Problemgröße: 10^9			Problemgröße: 10^{12}		
	n	$n \log n$	n^2	n	$n \log n$	n^2	n	$n \log n$	n^2
10^6	Sekunden	Sekunden	Wochen	Stunden	Stunden	niemals	Wochen	Monate	niemals
10^9	sofort	sofort	Stunden	Sekunden	Sekunden	Jahrzehnte	Minuten	Stunden	niemals
10^{12}	sofort	sofort	Sekunden	sofort	sofort	Wochen	Sekunden	Minuten	niemals
10^{15}	sofort	sofort	sofort	sofort	sofort	Minuten	sofort	sofort	Jahrzehnte

Mit einem schnellen Algorithmus kann ein Problem auf einem langsamen Computer gelöst werden.

Ein schneller Computer hilft nicht, wenn man einen langsamen Algorithmus verwendet.

§1.4 Abschätzung der Laufzeit von Algorithmen
Allgemeine Regeln

Regel 1: **Schleifen***

- Die Laufzeit von Schleifen ist die Laufzeit des Schleifenkörpers multipliziert mit Anzahl der Iterationen.

```
for (int i=0; i<n; i++)
   k++;
```

$O(n)$

* Mit unabhängigen Schleifenkopf und Schleifenkörper.

Regel 2: **Verschachtelte Schleifen***

- Die Laufzeit verschachtelter Schleifen ist die Laufzeit der inneren Schleife multipliziert mit Anzahl der Iterationen der äußeren Schleife.

```
for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      k++;
```

$O(n^2)$

Regel 3: **Sequenz von Schleifen**

- Die Laufzeit ist Summe der Laufzeiten der einzelnen Schleifen.
- Das bedeutet, dass die größere Laufzeit zählt.

```
for (int i=0; i<n; i++)
   a[i]=0;
for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      k++;
```

$O(n^2)$

Regel 4: **if-then-else**

- Die Laufzeit ist niemals mehr als Laufzeit des Tests plus die größere der Laufzeiten von **`S1`** und **`S2`**.

```
if (condition) S1;
else           S2;
```

Regel 5: **Rekursionen**

- Es gibt keine Regel für allgemeine Rekursionen.
- Für spezielle Rekursionen (teile-und-herrsche, divide-and-conquer) gibt es eine Regel... siehe Seite 36.

§1.4 Abschätzung der Laufzeit von Algorithmen

Beispiel: Sequenzielle Suche

```
int search(KeyType a[], KeyType key)
{  // Voraussetzung: keine
   for (int i = 0; i < a.length; i++)
      if (a[i] == key) return i;
   return -1;
}
```

- **Problem:** Suche den Index des Elements mit Schlüssel **`key`** in einem Feld **`a`** der Länge **`n`**.
- **Laufzeit der sequenziellen Suche:** Zähle nur Schlüsselvergleiche
  - Best case: $O(1)$
  - Worst case: $O(n)$
  - Average case: $O(n)$
    - **Annahme:** Alle Zahlen sind gleich wahrscheinlich.
    - $(1 + 2 + \cdots + n)/n = (n + 1)/2$

§1.4 Abschätzung der Laufzeit von Algorithmen

Beispiel: Binäre Suche

```
int search(KeyType a[], KeyType key)
{  // Voraussetzung: a ist aufsteigend sortiert
   int li = 0;
   int re = a.length-1;
   while (re >= li)
   {
      int m = (li+re)/2;
      if (key == a[m]) return m;
      if (key <  a[m]) re = m-1;
      else             li = m+1;
   }
   return -1;
}
```

- **Laufzeit der binären Suche:** Zähle nur Schlüsselvergleiche
  - Annahme: Das Feld ist aufsteigend sortiert.
  - Best case: $O(1)$
  - Worst case: $O(\log n)$

§1.4 Abschätzung der Laufzeit von Algorithmen

Beispiel: Rekursive Suche

```
int search(KeyType a[], int li, int re, KeyType key)
{  // Voraussetzung: a ist aufsteigend sortiert

   if (re-li <   0)  return -1;
   int m = (li+re)/2;
   if (key == a[m])  return m;
   if (key <  a[m])  return search(a,li,m-1,key);
   else              return search(a,m+1,re,key);
}
```

- **Laufzeit der rekursiven Suche:** Zähle nur Schlüsselvergleiche
  - Annahme: Das Feld ist aufsteigend sortiert.
  - Worst case: $T(n) = 1 \cdot T(\lfloor n/2 \rfloor - 1) + 2 = O(?)$

Teile-und-herrsche-Verfahren (engl.: *divide and conquer*)

```
void lösProblem(Problem x, Größe n) {
   if Problem ist einfach zu lösen (z.B. Größe n = 1) {
      // Basisfall
      Löse Problem x direkt;
   } else    {
      // Teile-Schritt:
      Teile Problem x in Teilprobleme x_1,...,x_k, wobei die
      Teilprobleme x_i derselben Art sind wie das
      Ausgangsproblem und die Größe n/b, b > 1, haben;
      // Löse Teilprobleme rekursiv:
      Löse a ≤ k (alle notwendigen) der Teilprobleme rekursiv;
      // Herrsche-Schritt:
      Setze Teillösungen zur Gesamtlösung zusammen;
   }
}
```

§1.4 Abschätzung der Laufzeit von Algorithmen
Allgemeine Regeln

- Ein Teile-und-Herrsche-Algorithmus mit Laufzeitfunktion $T(n)$
  1. Löst den Basisfall mit Laufzeit $f(n)$,
  2. zerlegt ein Problem der Größe n mit Laufzeit $f(n)$ in $k \geq 1$ Teilprobleme der Größe n/b, $b > 1$,
  3. löst $a \leq k$ Teilproblem mit Laufzeit $T(n/b)$ und
  4. fügt die Teillösungen mit Laufzeit $f(n)$ zusammen.
  - Für den Basisfall (Schritt 1.), das Zerlegen (Schritt 2.) und Zusammenfügen (Schritt 4.) wird insgesamt Laufzeit $f(n)$ aufgewendet.

  } f heißt auch Split-Funktion

- Die gesamte Laufzeit eines Teil-Und-Herrsche-Algorithmus kann dann mit dem

  Master-Theorem

  berechnet werden.

Theorem (Master Theorem für Rekursionen)

Sei $T(n)$ eine Laufzeitfunktion der Form $T(n) = a \cdot T(n/b) + f(n)$ mit

- Konstanten $a \geq 1$ und $b > 1$ und $\alpha := \log_b a$ (aka watershed constant),
- einer Funktion $f(n) > 0$ für hinreichend großes n, sowie
- n/b interpretiert als $\lceil n/b \rceil$ oder $\lfloor n/b \rfloor$.

Dann gibt es für $T(n)$ die folgenden drei Abschätzungen:…

Theorem (Master Theorem für Rekursionen)

Sei $T(n)$ eine Laufzeitfunktion der Form $T(n) = a \cdot T(n/b) + f(n)$ mit …

Dann gibt es für $T(n)$ die folgenden Abschätzungen:

1. Falls $f(n) = O(n^{\alpha-\varepsilon})$ für $\varepsilon > 0$, dann ist $T(n) = \Theta(n^{\alpha})$.
2. Falls $f(n) = \Theta\left(n^{\alpha} \log^{\delta} n\right)$
   a) mit $\delta = 0$ (Standardfall), dann ist $T(n) = \Theta(n^{\alpha} \log n)$,
   b) mit $\delta > -1$ (insb. Standardfall $\delta = 0$), dann ist $T(n) = \Theta\left(n^{\alpha} \log^{\delta+1} n\right)$,
   c) mit $\delta = -1$, dann ist $T(n) = \Theta(n^{\alpha} \log \log n)$,
   d) mit $\delta < -1$, dann ist $T(n) = \Theta(n^{\alpha})$.
3. Falls $f(n) = \Omega(n^{\alpha+\varepsilon})$ für $\varepsilon > 0$, dann ist $T(n) = \Theta(f(n))$,
   falls zusätzlich $a \cdot f(n/b) \leq c \cdot f(n)$ für $c < 1$ und hinreichend großes n.

Bemerkungen zum Master-Theorem

Fall 1: Der Aufwand für Teilen/Herrschen/Basisfälle **wird dominiert** durch den Aufwand zur Lösung der Teilprobleme.

- Das Hauptgewicht des Rekursionsbaums liegt in den Blättern (aka leaf-heavy).

Fall 2: Der Aufwand für Teilen/Herrschen/Basisfälle **ist asymptotisch ausbalanciert** mit dem Aufwand zur Lösung der Teilprobleme.

Fall 3: Der Aufwand für Teilen/Herrschen/Basisfälle **dominiert** den Aufwand zur Lösung der Teilprobleme.

- Das Hauptgewicht des Rekursionsbaums liegt in der Wurzel (aka root-heavy).

Nicht abgedeckte Fälle sind z.B. $f(n) = n^{\alpha} \cdot \log n \cdot \log\log n$ oder $f(n) = n^{\alpha} \cdot (\log\log n)^{r}$.

- Allgemeiner Versionen des Theorems werden benötigt.

§1.4 Abschätzung der Laufzeit von Algorithmen

Beispiel: Rekursive Suche

```
int search(KeyType a[], int li, int re, KeyType key)
{  // Voraussetzung: a ist aufsteigend sortiert

   if (re-li <   0)  return -1;
   int m = (li+re)/2;
   if (key == a[m])  return m;
   if (key <  a[m])  return search(a,li,m-1,key);
   else              return search(a,m+1,re,key);
}
```

- **Laufzeit der rekursiven Suche:** Zähle nur Schlüsselvergleiche
  - Annahme: Das Feld ist aufsteigend sortiert.
  - Worst case:
    - Master-Theorem: Laufzeitfunktion $T(n) = 1 \cdot T(n/2) + 2$ mit Konstanten $a = 1, b = 2, \delta = 0, \alpha = 0$, und einer Funktion $f(n) = 2$.
    - Fall 2a: $O(\log n)$

Die Komplexität für ein Problem ist oft noch unbekannt.

Beispiele:

- Sortieren
  - Die Komplexität $O(n \log n)$ kann nicht weiter verringert werden (§3).
  - Äquivalentes Problem: Konvexe Hülle im 2d.
- FFT
  - Laufzeit: $O(n \log n)$
  - Äquivalentes Problem: Polynommultiplikation

- Erfüllbarkeitsproblem (SAT)
  - Entscheidung, ob es für einen aussagenlogischen (Boolschen) Ausdruck (in KNF) mit n Variablen eine erfüllbare Wahrheitswertebelegungen gibt.
  - Beispiele:
    - $(\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge \neg x_3$ ist erfüllbar für $x_1 = F, x_2 = T, x_3 = F$.
    - $x_1 \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge x_3$ ist nicht erfüllbar.
  - Algorithmus
    1. Setze für den Booleschen Ausdruck systematisch alle möglichen Kombinationen von Wahrheitswerten ein.
    2. Prüfe, ob der Ausdruck erfüllt ist.
  - Bei n verschiedenen Aussagevariablen gibt es 2^n verschiedene Wahrheitswertebelegungen.

§1.5 NP-Probleme

Das Erfüllbarkeitsproblem (SAT)

- Laufzeit zur Berechnung der Entscheidung, ob SAT lösbar: $O(2^n)$
  - Grund: Für eine gegebene Wahrheitswertbelegung kann in polynomialer Laufzeit getestet werden, ob er den Ausdruck erfüllt oder nicht.
- Es ist unbekannt, ob es einen effizienteren Algorithmus gibt.
- ▶ Die Komplexität ist unbekannt.

- **Aber:** Eine Lösung von SAT kann leicht verifiziert werden:
  - Ist eine Wertebelegung eine Lösung des aussagenlogischen Ausdrucks?
- Laufzeitenvergleich:
  - **Berechnung** einer Lösung von SAT: $O(2^n)$
  - **Verifikation** einer Lösung von SAT: $O(n^k) \subset o(2^n)$, für ein $k \in \mathbb{N}$.

§1.5 NP-Probleme
Die Menge NP

- Es gibt noch viele weitere Problem, die vergleichbar „schwierig“ sind.
  - Die Lösung zu berechnen ist deutlich aufwändiger als eine potenzielle Lösung zu verifizieren.

- SAT gehört zur Klasse der Entscheidungsprobleme:
  - Hier: Ist ein aussagenlogischer Ausdruck erfüllbar?

Definition:

Ein Problem ist ein **Entscheidungsproblem**, wenn von einem gegebenen Objekt eine Eigenschaft nachgeprüft werden soll.

Alle Entscheidungsprobleme mit einem polynomialen Algorithmus zu **Berechnung** einer Lösung bilden die **Menge P**.

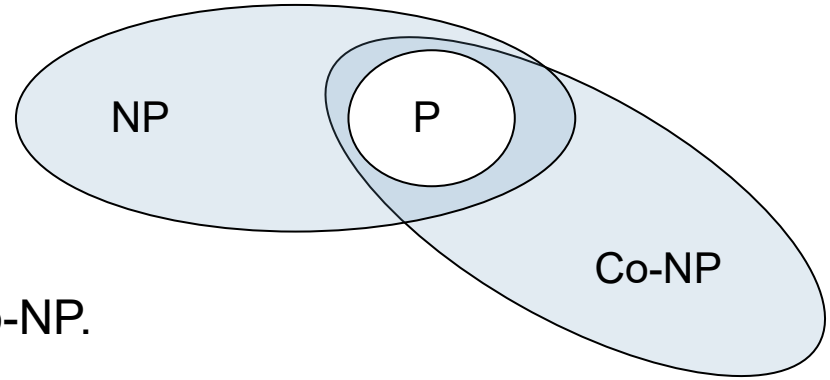
Alle Entscheidungsprobleme mit einem polynomialen Algorithmus zur **Verifikation** einer Lösung bilden die **Menge NP**.

Alle Entscheidungsprobleme mit einem polynomialen Algorithmus zur **Verifikation** eines Gegenbeispiels bilden die **Menge Co-NP**.

§1.5 NP-Probleme
Die Menge NP

- Es gilt sicher: P ⊆ NP und P ⊆ Co-NP !
  - Jeder polynomiale Lösungsalgorithmus kann auch zur Verifikation einer Lösung bzw. eines Gegenbeispiels verwendet werden.

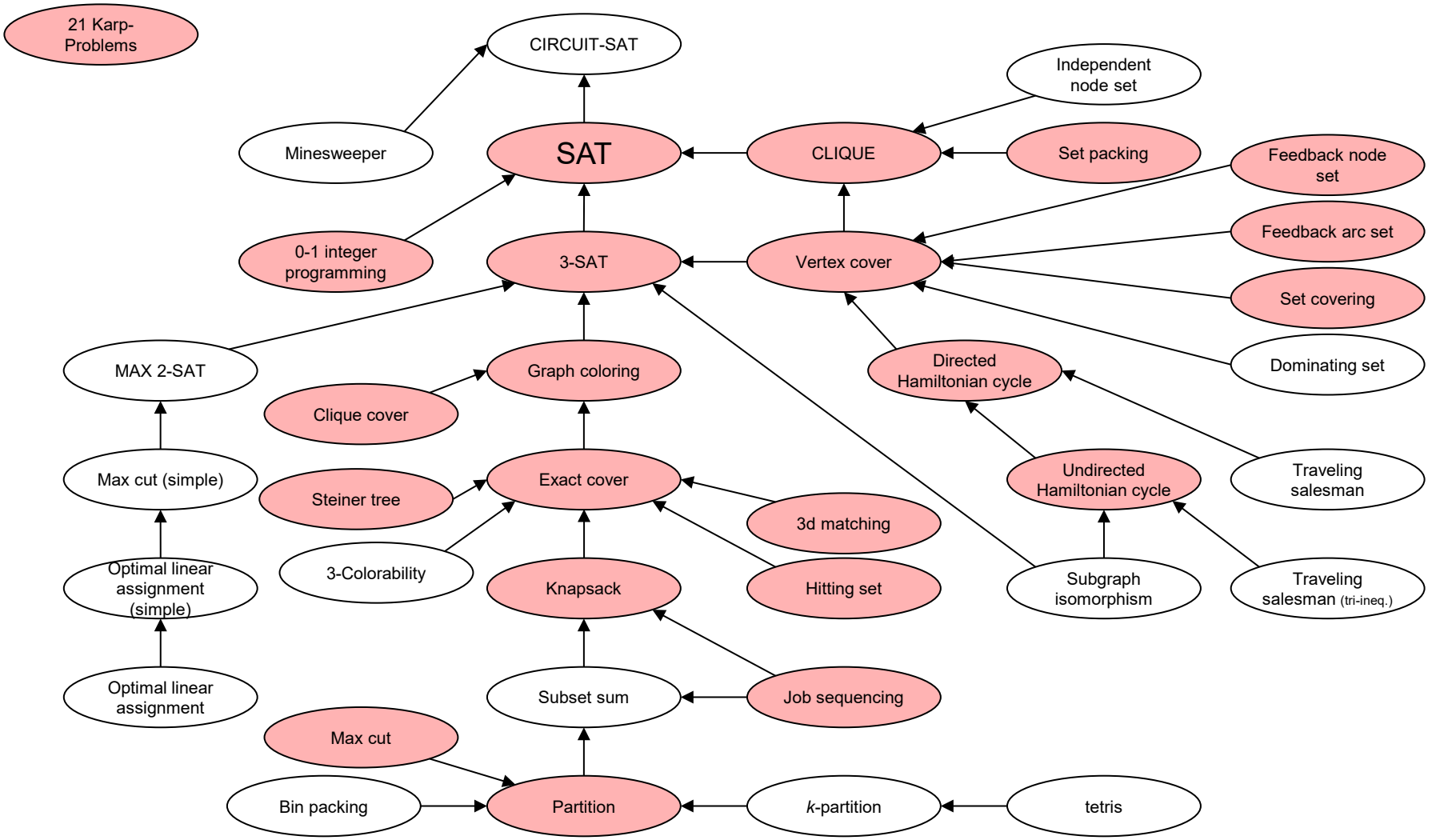
- **Aber** ob P = NP gilt, ist die größte ungeklärte Frage der Informatik!



  - **Vermutlich** gilt P ≠ NP und NP ≠ Co-NP.
  - Es sprechen einige Indizien dafür:
    1. In 60 Jahren Forschung ist
       - der Gleichheitsbeweis nicht gefunden worden ;-)
       - kein polynomialer Lösungsalgorithmus für die schwierige Probleme gefunden worden ;-)
    2. Es gibt in der Menge NP Probleme, die schwieriger sind als andere...

§1.5 NP-Probleme

Die Menge NP

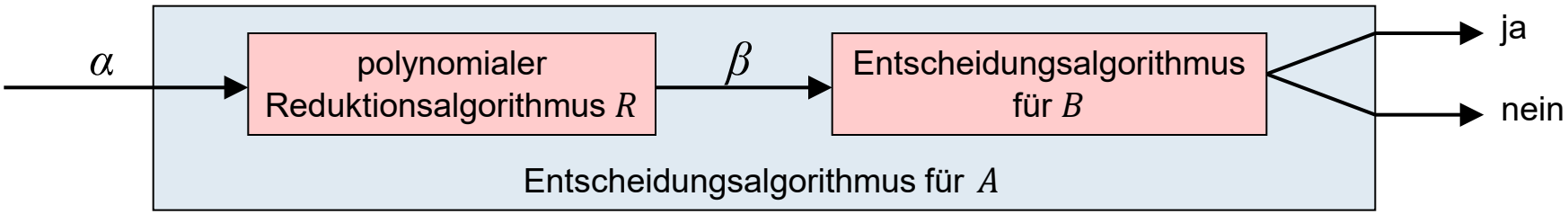


- Wie weist man nach, dass ein Problem genauso schwierig ist wie ein anderes?

Polynomiale Reduktion

Ein Problem A ist auf ein anderes Problem B polynomial reduzierbar, wenn

1. es einen **Reduktionsalgorithmus** R mit polynomialer Laufzeit gibt, der aus einer Eingabe α für A eine Eingabe β für B berechnet, so dass
2. der Algorithmus zur Verifikation bzw. Lösung für B auf β angewendet eine Lösung bzw. Verifikation liefert,
3. die dann als Antwort auf die Eingabe α interpretiert werden kann.



- Kurzschreibweise: $A \leq B$.
- Anschaulich: Der beste Algorithmus für A ist mindestens so schnell wie der beste Algorithmus für B,
  - falls der Aufwand für die Reduktion R „vernachlässigbar“ ist,

  denn die Reduktion auf B liefert einen solchen Algorithmus.
- Anschaulich: Der beste Algorithmus für B ist höchstens so schnell wie der beste Algorithmus für A,
  - falls der Aufwand für die Reduktion R „vernachlässigbar“ ist,

  denn sonst lieferte die Reduktion auf B einen schnelleren Algorithmus für A.

Bemerkung: Die polynomiale Reduzibilität ist transitiv, d.h.

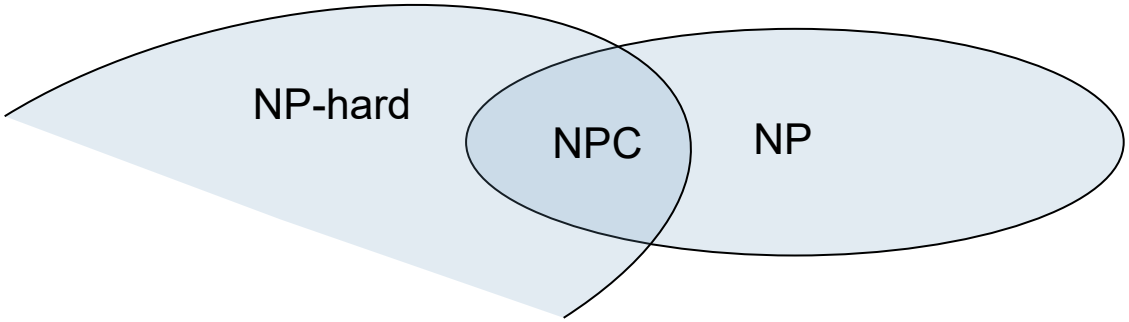
$$A \leq B \text{ und } B \leq C \;\Rightarrow\; A \leq C.$$

Definition:

Ein Entscheidungsproblem B heißt **NP-vollständig** (engl.: np-complete),

1. wenn es zu NP gehört,
   - d.h. es gibt eine polynomiale Verifikation,
2. und wenn es **NP-schwer** (engl.: np-hard) ist,
   - d.h. es gibt für jedes A aus NP eine polynomiale Reduktion auf B: $A \leq B$.
   - d.h. B ist höchstens so schnell lösbar wie jedes Problem A aus NP.

- Die NP-vollständigen Probleme bilden die **Menge NPC**.



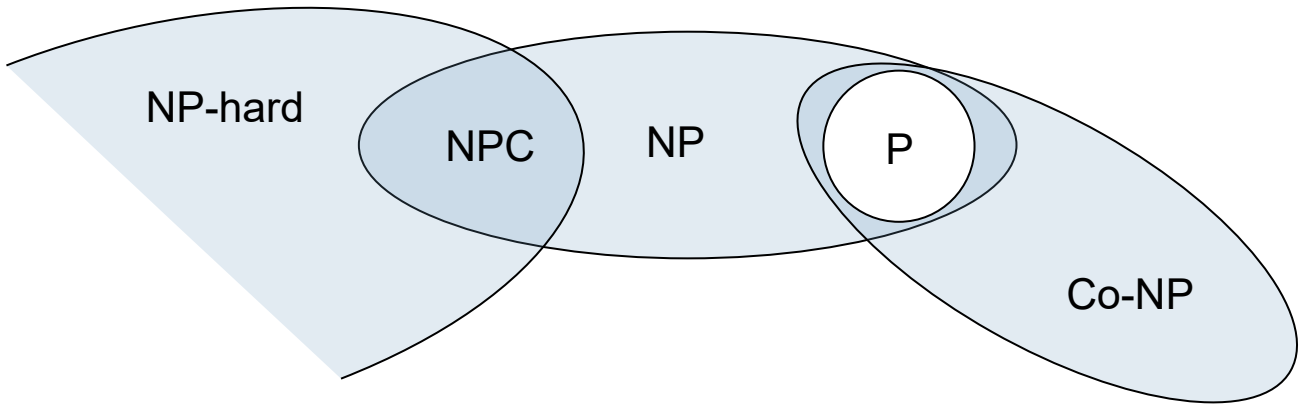
§1.5 NP-Probleme

NP-vollständige Probleme

- Alle NP-vollständigen Probleme sind gleich schwierig:
- Ein polynomialer Algorithmus für ein einziges NPC-Problem würde $P = NP$ beweisen!

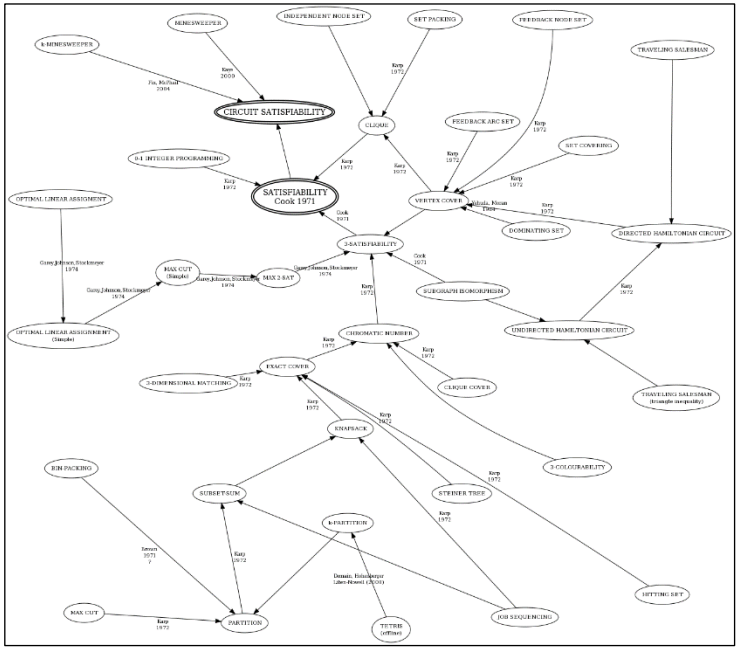
Gibt es für ein NP-vollständiges Problem einen polynomialen Algorithmus, gibt es für alle NP-vollständigen Probleme einen polynomialen Algorithmus.

- Für alle NPC-Probleme sind aber nur exponentielle Algorithmen bekannt.
  - Es ist nicht bekannt, ob es für NPC-Probleme polynomiale Algorithmen gibt.



§1.5 NP-Probleme

- Weitere Beispiele:
  - Aussagenlogik: SAT, CIRCUIT-SAT, etc.
  - Graphentheorie: Cliquen-Problem, Hamilton-Zyklus-Problem, etc.
  - Anwendungen: Problem des Handlungsreisenden, Teilsummenproblem, Rucksack-Problem, etc.
  - Mehr als 3000 NPC-Probleme sind bekannt.



Quelle: wikipedia

- Bemerkung:
  - Integer-Faktorisierung ist in NP und Co-NP.
  - Es ist aber nicht bekannt, ob Integer-Faktorisierung in P liegt!

§1.5 NP-Probleme
Lösungsansätze

- Lösungsansätze für schwierige Probleme
  - **Randomisierte Algorithmen**: Der Ablauf wird zufällig gesteuert.
    - **Las-Vegas-Algorithmen**:
      - Liefern immer das richtige Ergebnis.
      - Der Erwartungswert der Laufzeit ist polynomial, im *worst case* aber exponentiell.
    - **Monte-Carlo-Algorithmen**:
      - Liefern nur zu einer vorgegebenen Wahrscheinlichkeit eine korrekte Lösung.
      - Die Laufzeit ist im *worst case* (i.d.R.) polynomial.
  - **Approximative Algorithmen**:
    - Die Lösung ist nicht optimal, aber innerhalb einer vorgegebenen Toleranz.
    - Die Laufzeit ist im *worst case* polynomial.
    - Die Laufzeit ist exponentiell in der vorgegebenen Toleranz.

Lernziele

Am Ende dieser Lerneinheit sollten Sie …

- … die Bedeutung und die wichtigen Inhalte der Komplexitätstheorie kennen,
- … den Unterschied zwischen best case, average case und worst case Laufzeiten kennen,
- … das O-Kalkül beherrschen und Laufzeitfunktionen vergleichen können,
- … die Laufzeit eines Algorithmus eigenständig bestimmen können,
- … den Unterschied zwischen Laufzeit und Komplexität kennen,
- … schwierige Probleme und deren Behandlung kennen.

Mathematisches Glossar

- Die **Summe** der Folgenglieder einer Folge $a_1, a_2, a_3, \ldots$ von a_m bis a_n, $m \leq n$, wird abgekürzt als

$$\sum_{i=m}^{n} a_i = a_m + a_{m+1} + \cdots + a_n.$$

- Spezielle Summen:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2},$$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6},$$

$$\sum_{i=0}^{n} q^i = 1 + q + q^2 + q^3 + \cdots + q^n = \frac{1 - q^{n+1}}{1 - q}, \quad q \neq 1.$$

Mathematisches Glossar

- Die n-te **Potenz** von a: $a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n}$
  - $a \in \mathbb{R}$ heißt Basis
  - $n \in \mathbb{N}$ heißt Exponent (auch $n \in \mathbb{R}$ möglich)
- Spezielle Exponenten: $a^0 = 1 \qquad (0^0 := 1)$
  $a^1 = a$
- Negative Exponenten: $a^{-n} = \frac{1}{a^n}$
- Rationale Exponenten ($m \in \mathbb{N}$): $a^{\frac{m}{n}} = \sqrt[n]{a^m} = \left(\sqrt[n]{a}\right)^m$
- Produkt von Potenzen $x, y \in \mathbb{R}$: $a^x \cdot a^y = a^{x+y}$
- Potenz von Potenzen $x, y \in \mathbb{R}$: $(a^x)^y = a^{x \cdot y}$

Mathematisches Glossar

- **Logarithmus** von a zur Basis b: $\log_b a$
  - Derjenige Exponent, mit dem man b potenzieren muss, um a zu erhalten, d.h. $b^x = a \Leftrightarrow x = \log_b a$
- Spezielle Werte: $\log_b 1 = 0,\ \log_b b = 1$
- Wichtige Logarithmen:
  - Dekadischer Logarithmus (Basis 10): $\lg a = \log_{10} a$
  - Natürlicher Logarithmus (Basis e): $\ln a = \log_e a$
  - Dualer Logarithmus (Basis 2): $\mathrm{ld}\, a = \mathrm{lb}\, a = \log_2 a$
- Umrechnung von Logarithmen: $\log_b x = \frac{\ln x}{\ln b}$
- Die gewöhnlichen Rechenregeln für Logarithmen und Exponential-funktionen (Produkte, Summen, etc.) werden als bekannt vorausgesetzt.

Mathematisches Glossar

- **Ableitungen** spezieller Funktionen
  - Potenzfunktion ($x \neq 0, a \in \mathbb{R}$): $\frac{d}{dx}(x^a) = a \cdot x^{a-1}$
  - Logarithmusfunktion: $\frac{d}{dx} \ln x = \frac{1}{x}$

    $\frac{d}{dx} \log_a x = \frac{1}{x \cdot \ln a}$
  - Exponentialfunktion: $\frac{d}{dx} e^x = e^x$

    $\frac{d}{dx} a^x = a^x \cdot \ln a$

Mathematisches Glossar

Ableitungsregeln

- Summenregel: $(f + g)' = f' + g'$
- Produktregel: $(f \cdot g)' = f' \cdot g + f \cdot g'$
- Quotientenregel ($g \neq 0$): $\left(\frac{f}{g}\right)' = \frac{f' \cdot g - f \cdot g'}{g^2}$
- Kettenregel: $\big(f(g)\big)' = f'(g) \cdot g'$
- Regel von de l'Hospital ($g' \neq 0$)

$$\lim_{x \to x_0} \frac{f'(x)}{g'(x)} = c \quad \Rightarrow \quad \lim_{x \to x_0} \frac{f(x)}{g(x)} = c$$

mit einer von x unabhängigen Konstanten c.

Bemerkung: Dabei ist $x_0 = \pm\infty$ und $c = \pm\infty$ erlaubt.

Mathematisches Glossar

- Bestimmung des **Erwartungswerts** E einer Zufallsvariable X.
  - **Werte:** Die Zufallsvariable X kann die Werte

    $$X_1, \dots, X_n$$

    annehmen.
  - **Wahrscheinlichkeitsverteilung:** Die Werte $X_1, \dots, X_n$ werden mit den Wahrscheinlichkeiten

    $$p_1, \dots, p_n$$

    angenommen.

    Dabei muss gelten: $\sum_{i=1}^{n} p_i = p_1 + \cdots + p_n = 1$.
  - Der **Erwartungswert** $E(X)$ berechnet sich als

    $$E(X) = \sum_{i=1}^{n} p_i \cdot X_i \,.$$