Technische Hochschule Rosenheim

Theoretische Informatik

Komplexitätstheorie

Technische Hochschule Rosenheim

SS 2019

Prof. Dr. J. Schmidt

Inhalt

- Einführung Zeit- und Speicherkomplexität
- Ordnung der Komplexität, O-Notation
- Optimierung am Beispiel Teile und Herrsche
- Komplexitätsklassen P, NP
- NP-Vollständigkeit
- NP-schwere Probleme
- weitere Problemklassen

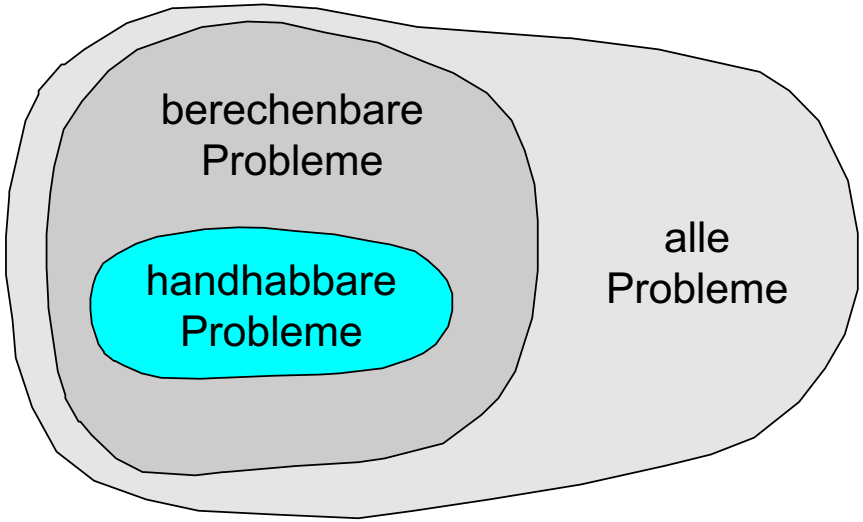
Einführung

- bisher betrachtet: Berechenbarkeit
  - ist ein Problem prinzipiell mit Computern lösbar – existiert ein Algorithmus?
- jetzt: mit welchem **Aufwand** ist ein berechenbares Problem lösbar, insbesondere
  - Zeitkomplexität
  - Speicherkomplexität
- es geht also nicht mehr nur um Effektivität, sondern um Effizienz

- im Folgenden: Zeitkomplexität
  - Speicherkomplexität wird mit ähnlichen Methoden betrachtet
  - ist aber für die Praxis oft weniger wichtig

Einführung

- nur ein Teil der berechenbaren Probleme ist handhabbar
  - die anderen rechnen für praktische Fragestellungen zu lange oder benötigen zu viel Speicher



(Lauf)Zeitkomplexität

- eines Algorithmus
  - Anzahl der Rechenschritte, die er zur Lösung des Problems benötigt.
- eines Problems
  - Laufzeitkomplexität, die ein optimaler Algorithmus zur Lösung benötigt.

Zeitkomplexität – Varianten

Technische Hochschule **Rosenheim**

- Worst-case Laufzeit
  - wie lange braucht der Algorithmus maximal (bei „schlechtesten“ Eingabedaten)
  - meist: Laufzeit = worst-case Laufzeit
- Average-case Laufzeit
  - erwartete Laufzeit bei einer gegebenen üblichen Verteilung der Daten („durchschnittliche Laufzeit“)
- Best-case Laufzeit
  - wie lange braucht der Algorithmus mindestens (bei optimalen Eingabedaten)

- Beispiel:
  verkettete Liste mit 20 Namen, suche darin einen Namen
  - Worst-Case: letzter Name → 20 Schritte
  - Average-Case: Name in der Mitte der Liste → 10 Schritte
  - Best-Case: erster Name → 1 Schritt

ORDNUNG DER KOMPLEXITÄT

Zeitkomplexität – Ziele

Technische Hochschule **Rosenheim**

- Abhängigkeit von der Größe der Eingangsdaten → Parameter n
  - wie verhält sich der Algorithmus, wenn die Anzahl der Eingabedaten sich erhöht
- „Weglassen“ von „unwichtigen“ Konstanten
  - konstante Faktoren wie verwendeter Rechner, eingesetzte Programmiersprache und deren Implementierung oder Taktfrequenz der CPU
  - Komplexitätsangabe soll nur vom Algorithmus abhängen, nicht von tatsächlich verwendeter Hardware
- Untersuchung einer oberen Schranke („asymptotische Laufzeitkomplexität“)
  - Ergebnis soll, multipliziert mit einem rechnerabhängigen konstanten Faktor, stets ÜBER der tatsächlichen Laufzeitfunktion liegen, wenn die Anzahl der Eingabewerte einmal einen bestimmten Wert überschritten haben

O-Notation
Definition

Technische Hochschule **Rosenheim**

- $O\big(f(n)\big) = \{g: \mathbb{N} \to \mathbb{N} | \exists m > 0, c > 0$ mit
  $\forall n \geq m: |g(n)| \leq c \cdot |f(n)|\}$
- d.h. $O\big(f(n)\big)$ ist die Menge aller Funktionen $g(n)$,
  - für die es die beiden Konstanten m, c gibt,
  - so dass für alle $n \geq m$ gilt, dass $|g(n)| \leq c \cdot |f(n)|$
- oder anders: $g(n)$ wächst höchstens so schnell wie $f(n)$
- dies gilt asymptotisch, also für n $\longrightarrow \infty$

- übliche Schreibweise: $g(n) = O(f(n))$
  - also z.B. $g(n) = O(n^2)$
  - eigentlich nicht korrekt, es sollte „$\in$“ verwendet werden
  - = ist hier nicht symmetrisch:
    es gilt zwar $O(n) = O(n^2)$ aber nicht $O(n^2) = O(n)$

O-Notation – Beispiele

- $f(n) = 50n + 3$
  - $f(n) = O(n)$
  - $c = 51, m = 3$

- $f(n) = 2n^2 – 50n + 3$
  - $f(n) = O(n^2)$
  - $| 2n^2 – 50n + 3 | \leq 2n^2 + |50n| + 3 \leq 2n^2 + 50n^2 + 3n^2 = 55n^2 = |55n^2|$
  - also $| 2n^2 – 50n + 3 | \leq 55 |n^2|$
  - und damit $c = 55, m = 1$

- Fazit:
  - es ist nur der am schnellsten wachsende Term relevant
  - alle langsamer wachsenden Terme und konstante Faktoren werden weggelassen

O-Notation – Beispiele

- $f(n) = \ln n - 3n + 2n^3$
  - $f(n) = O(n^3)$
- $f(n) = 3 \ln n$
  - $f(n) = O(\ln n)$
- $f(n) = \ln n^c$
  - $\ln n^c = c \ln n$ → konstanter Faktor
  - $f(n) = O(\ln n)$
- $f(n) = 3 \log_2 n$
  - $\log_2 n = \ln n / \ln 2$ → konstanter Faktor
  - $f(n) = O(\ln n)$
- Fazit:
  - Basis eines Logarithmus ist irrelevant
  - konstante Exponenten unter dem Logarithmus sind irrelevant

O-Notation – Beispiele

- $f(n) = \log n - 3n + 2n^3 + 2^n$
  - $f(n) = O(2^n)$
- $f(n) = \log n - 3n + 2n^3 + 10^n$
  - $f(n) = O(10^n)$
- $f(n) = \log n - 3n + 2n^3 + 2^n + 10^n$
  - $f(n) = O(10^n)$
- Fazit:
  - Änderung der Basis einer Exponentialfunktion ist relevant

O-Notation – Beispiele

- $f(n) = 50n + 3$
  - $f(n) = O(2^n)$
- $f(n) = 2n^2 – 50n + 3$
  - $f(n) = O(2^n)$
- $f(n) = \ln n – 3n + 2n^3$
  - $f(n) = O(2^n)$
- $f(n) = 3 \ln n$
  - $f(n) = O(2^n)$
- Fazit:
  - obige Aussagen sind richtig, aber nicht sehr hilfreich
  - gesucht ist i.a. eine **enge** obere Schranke

Landau-Symbole

- eingeführt von Paul Bachmann 1894
- benannt nach Edmund Landau (1877 – 1938)
- weitere Symbole zusätzlich zu O-Notation
- insbesondere noch interessant: Ω, Θ

$g = O(f)$	g wächst höchstens so stark wie f (obere Schranke)	$\lvert g(n)\rvert \leq c \cdot \lvert f(n)\rvert$
$g = \Omega(f)$	g wächst mindestens so stark wie f (untere Schranke)	$\lvert g(n)\rvert \geq c \cdot \lvert f(n)\rvert$
$g = \Theta(f)$	g wächst genauso stark wie f	$c_0 \cdot \lvert g(n)\rvert \leq \lvert f(n)\rvert \leq c_1 \cdot \lvert g(n)\rvert$

Typische Komplexitätsordnungen

Technische Hochschule **Rosenheim**

Bezeichnung	Komplexität	Wertung	Beispiele
Konstante Komplexität	O(1)	optimal, selten	Hashing
Logarithmische Komplexität	O(log n)	sehr günstig	Binäre Suche in sortierter Liste
Lineare Komplexität	O(n)	günstig	Lineare Suche in unsortierter Liste
Leicht überlineare Komplexität	O(n log n)	noch gut	gute Sortierverfahren, z.B. Mergesort, Quicksort (im Durchschnitt); FFT
Quadratische Komplexität	O(n^2)	ungünstig	schlechte Sortierverfahren, z.B. Bubblesort Quicksort (worst case)
Kubische Komplexität	O(n^3)	ungünstig	Matrix-Multiplikation
Exponentielle Komplexität	O(a^n)	katastrophal	Travelling-Salesman (geschickt implementiert)
Faktorielle Komplexität	O($n!$)	noch schlimmer…	Travelling-Salesman (brute-force)

Anmerkung a^n wächst schneller als **jedes** Polynom n^k für jedes $a > 1$

Beispiele für die O-Notation

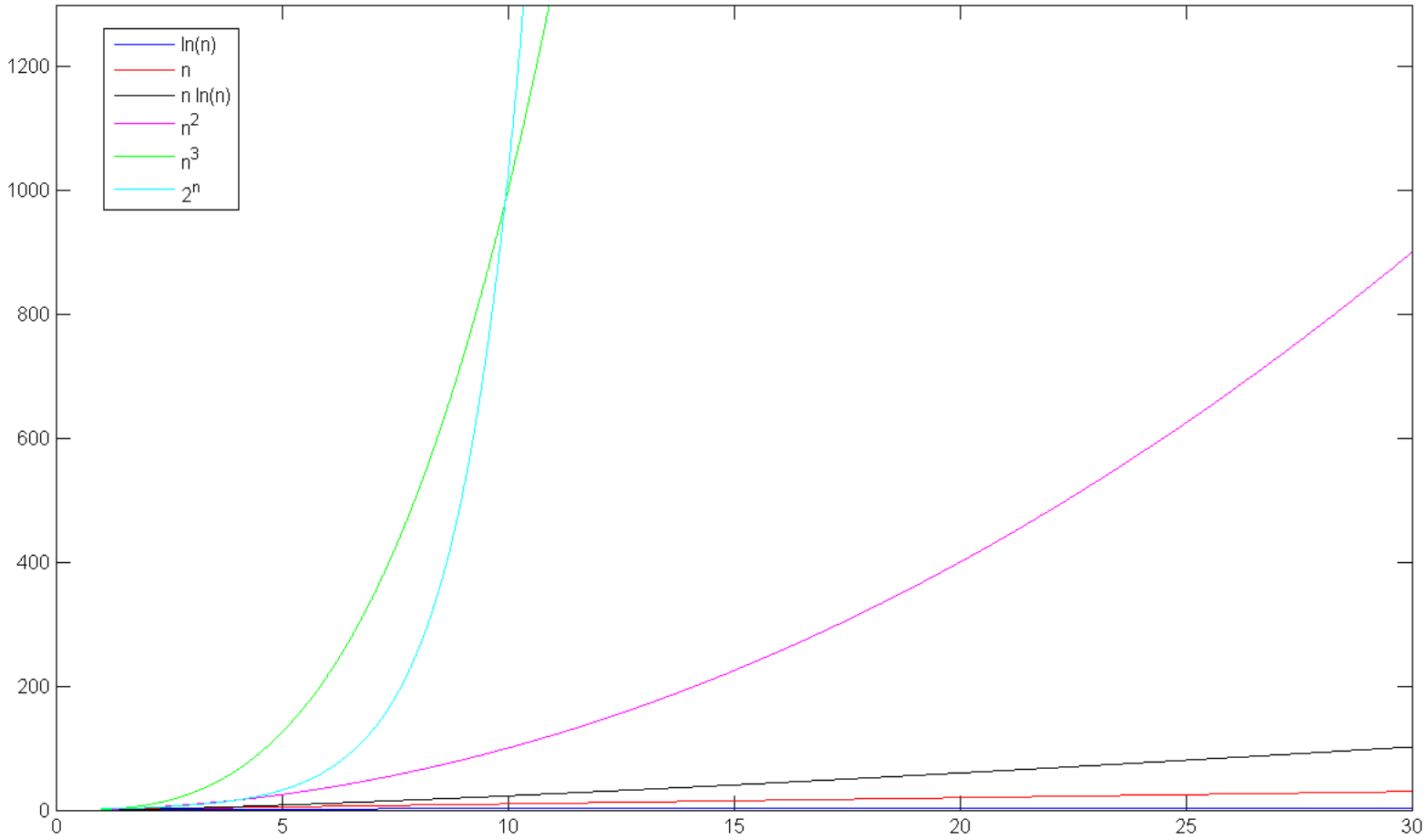
Technische Hochschule Rosenheim

n	$O(n)$	$O(n^2)$	$O(2^n)$
1	1 µsec	1 µsec	2 µsec
10	10 µsec	100 µsec	~ 1 msec
100	100 µsec	10 msec	~ $4 * 10^{16}$ Jahre
1000	1 msec	1 sec	~ $8 * 10^{288}$ Jahre

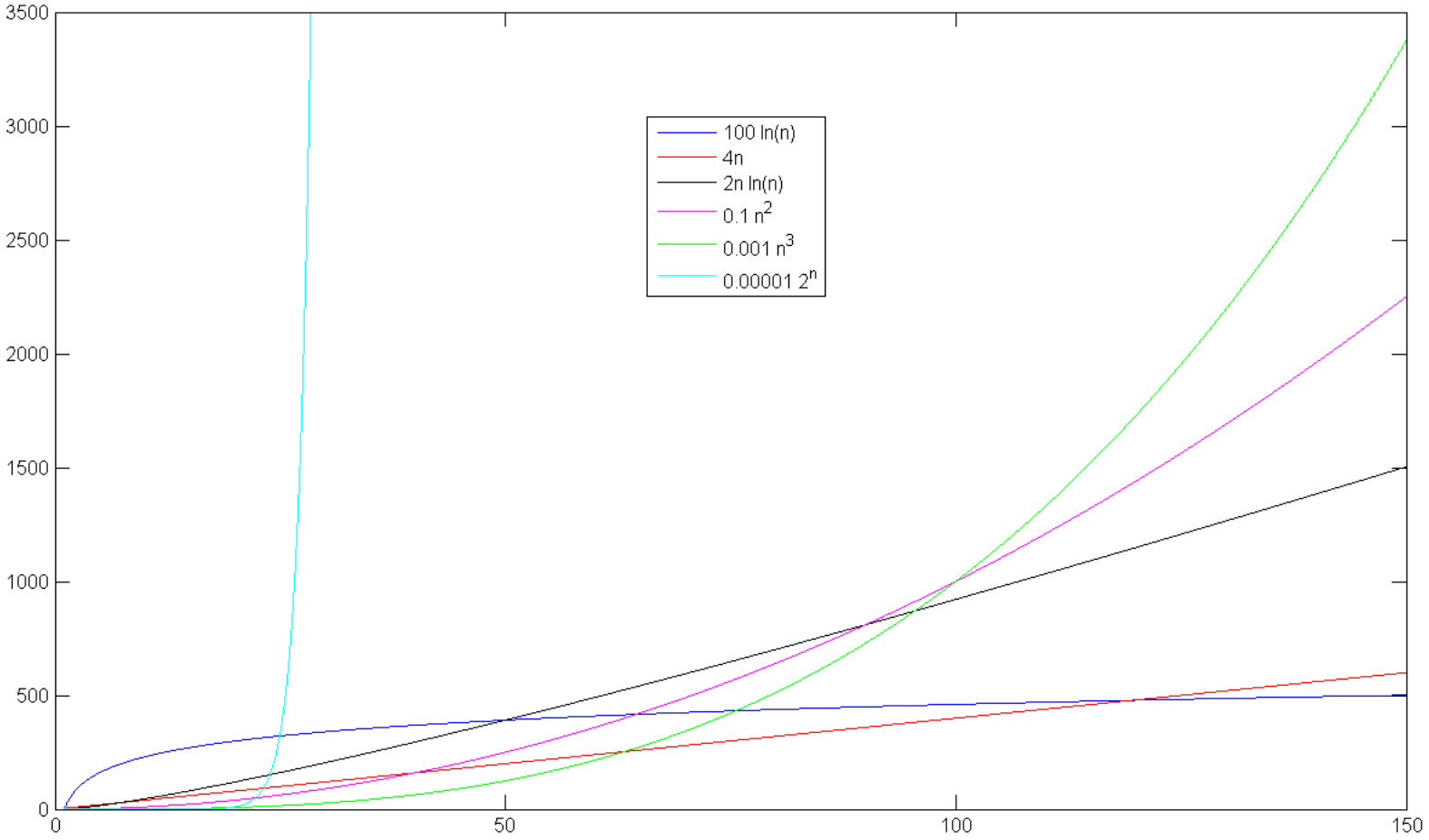
Achtung: die O-Notation gilt nur asymptotisch für $n \to \infty$

n	$O(100\,n) = O(n)$	$O(0.1\,n^2) = O(n^2)$	$O(0.0001\,2^n) = O(2^n)$
1	100 µsec	0.1 µsec	0.0002 µsec
10	1 msec	10 µsec	~ 0.1 µsec
100	10 msec	1 msec	~ $4 * 10^{12}$ Jahre
1000	100 msec	100 msec	~ $8 * 10^{284}$ Jahre

Funktionswachstum – Beispiele



Funktionswachstum – Beispiele



Komplexitätsordnung – typische Art der Problemlösung

Technische Hochschule **Rosenheim**

Bezeichnung	Komplexität	Typischer Aufbau des Algorithmus
Konstante Komplexität	O(1)	die meisten Anweisungen werden nur einmal oder ein paar Mal ausgeführt
Logarithmische Komplexität	O(log n)	Lösen eines Problems durch Umwandlung in ein kleineres, dabei Verringerung der Laufzeit um einen konstanten Anteil
Lineare Komplexität	O(n)	optimaler Fall für einen Algorithmus, der n Eingabedaten verarbeiten muss – jedes Element muss genau einmal (oder konstant oft) angefasst werden
Leicht überlineare Komplexität	O(n log n)	Lösen eines Problems durch Aufteilen in kleinere Probleme, die unabhängig voneinander gelöst und dann kombiniert werden
Quadratische Komplexität	O(n^2)	typisch für Probleme, bei denen alle n Elemente paarweise verarbeitet werden müssen (2 verschachtelte for-Schleifen) . Nur für relativ kleine Probleme verwendbar
Kubische Komplexität	O(n^3)	3 verschachtelte for-Schleifen. Nur für kleine Probleme verwendbar
Exponentielle Komplexität	O(a^n)	typisch für brute-force Lösungen, z.B. durchprobieren aller möglichen Varianten. Nur wenige Algorithmen dieser Komplexität sind praktisch einsetzbar

Bestimmung der Laufzeitkomplexität

- Einfach Anweisungen:

```
a = 15;
```

O(1)

```
x = x * a;
```

O(1)

Bestimmung der Laufzeitkomplexität

- For-Schleifen (wobei B ein Block mit konstanter Laufzeit sei)

```
for(int i = 0; i < n; i++)
{
  B;
}
```

$O(n)$

```
for(int i = 0; i < n; i++)
{
  for(int j = 0; j < n; j++)
  {
    B;
  }
}
```

$O(n^2)$

```
for(int i = 0; i < n; i++)
{
  for(int j = 0; j < i; j++)
  {
    B;
  }
}
```

$O(n^2)$

Bestimmung der Laufzeitkomplexität (2)

Technische Hochschule **Rosenheim**

- Binäre Suche:

```
while(i<n)
{
  n /= 2;
  i += 1;
}
```

O(log n)

- Rekursion

```
int fac(int n)
{
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

O(n)

```
int doSomething(int a, int b)
{  // Vorauss.: a < b
  if (a == b)
    return 0;
  else
    return (doSomething(a+1, b) - doSomething(a, b-1));
}
```

O(2^n)

Rekursion

Technische Hochschule Rosenheim

- betrachtet werden im Folgenden typische Varianten der Rekursion
- es werden unabhängig von einem konkreten Algorithmus Formeln zur Berechnung der Komplexität K_n angegeben
- es gilt jeweils $K_0 = 0$

- Variante 1
  - Schleife über Eingabedaten
  - in jedem Schritt wird ein Element entfernt

$$K_n = K_{n-1} + n$$

$$K_n = O(n^2 / 2)$$

Rekursion

- Variante 2
  - Eingabedaten werden in jedem Schritt halbiert
  - Aufwand innerhalb eines Schritts ist konstant

$$K_n = K_{n/2} + 1$$

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

- Variante 3
  - Eingabedaten werden in jedem Schritt halbiert
  - innerhalb eines Schritts muss jedes Element betrachtet werden

$$K_n = K_{n/2} + n$$

$$K_n = O(2n)$$

Rekursion

- Variante 4
  - Eingabedaten werden in zwei Hälften geteilt
  - Aufwand innerhalb eines Schritts ist konstant

    $K_n = 2K_{n/2} + 1$

    $K_n = O(2n)$

- Variante 5
  - Eingabedaten werden in zwei Hälften geteilt
  - alle Daten müssen vor/zwischen/nach der Halbierung betrachtet werden
  - typisch für viele „Teile und Herrsche“ („Divide-and-Conquer“) Algorithmen

    $K_n = 2K_{n/2} + n$

    $K_n = O(n \log n)$

Rechenregeln zur O-Notation

Technische Hochschule **Rosenheim**

Seien c und a_i Konstanten.

- $c = O(1)$
- $c \cdot f(n) = O(f(n))$
- $O(f(n)) + O(f(n)) = O(f(n))$
- $O(O(f(n))) = O(f(n))$
- $g(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_0 = O(n^k)$
- $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
- $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$

Beweise ergeben sich direkt aus der Definition.

Anwendung zur Analyse

- Berechnung der Gesamtkomplexität eines Algorithmus
- einfache Anweisungen sind O(1)
- Berechnung **Sequenzen** alg1; alg2; alg3;
  O(alg1) + O(alg2) + O(alg3) = O(max{alg1, alg2, alg3})
- n-malige **Iteration** in Schleife mit Rumpf O(alg)
  O(n * alg)
- **IF-THEN** alg 1 ELSE alg2
  O(max{alg1, alg2})

OPTIMIERUNG AM BEISPIEL TEILE UND HERRSCHE

Ziel der Optimierung

- finden eines besseren Algorithmus
  - d.h., mit besserer Zeitkomplexität
- sehr abhängig vom Algorithmus

- früher oft: Ersetzen von Multiplikationen durch Additionen
  - da Multiplikation um ein Vielfaches langsamer war
  - heute mit Vorsicht zu genießen, CPU-Architekturabhängig

Beispiel: Polynomauswertung

- Verfahren zur Auswertung von Polynomen f(x) an Stelle b
- $f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$

- Komplexität „normales“ Verfahren zur Berechnung von f(b):
  - Berechnung der Potenzen $x^2, \ldots, x^n$:
    - $2 + 3 + 4 + \ldots + n = n\,(n + 1) / 2 - 1$ Multiplikationen: $O(n^2/2)$
  - n Multiplikationen mit Koeffizienten a_i
  - n Additionen
  - ergibt: $n\,(n + 1) / 2 - 1 + 2n = O(n^2/2 + 2n)$

- mit Wiederverwendung bereits berechneter Potenzen
  - in jedem Schritt nur eine zusätzliche Multiplikation, gesamt: $n - 1$
  - ergibt $2n - 1$ Multiplikationen und n Additionen
  - $3n - 1 = O(3n)$

Beispiel: Polynomauswertung

- Horner-Schema
  - geschickte Klammerung des Polynoms
  - $f(x) = a_0 + x\,(a_1 + x\,(a_2 + x\,(a_3 + \ldots + x\,(a_{n-1} + a_n x)\ldots)$
- Komplexität
  - n Multiplikationen
  - n Additionen
  - O(2n)

Teile und Herrsche

- *Teile und Herrsche* (*Divide and Conquer*):
  - Zerlegung eines Problems in sich nicht überlappende *Teilprobleme*
  - Zusammensetzen der Einzellösungen zur Gesamtlösung

- Oft:
  - Teilen von Wertebereichen in zwei Intervalle
  - getrennte Verarbeitung
- Rekursion durch mehrmaliges Hintereinanderausführen

- Beispiele
  - Quicksort, Mergesort
  - Karatsuba Verfahren zur Multiplikation langer Zahlen
  - schnelle Fourier-Transformation (FFT)

Teile und Herrsche

- Aufwand zur Zerlegung eines Problems der Größe n in a Teilprobleme der Größe n/b:

$$T(n) = a\,T(n/b) + \Theta(n^k) \qquad \text{für } a \geq 1,\ b, n > 1$$
$$T(1) = 1$$

- $\Theta(n^k)$: Aufwand zum Zerlegen und Zusammensetzen

- $T(n)$ kann wie folgt abgeschätzt werden:

$$T(n) = \begin{cases} \Theta(n^k) & \text{für } a < b^k \\ \Theta(n^k \log n) & \text{für } a = b^k \\ \Theta(n^{\log_b a}) & \text{für } a > b^k \end{cases}$$

Teile und Herrsche – Beispiele

$T(n) = a\ T(n/b) + \Theta(n^k)$

$$T(n) = \begin{cases} \Theta(n^k) & \text{für } a < b^k \\ \Theta(n^k \log n) & \text{für } a = b^k \\ \Theta(n^{\log_b a}) & \text{für } a > b^k \end{cases}$$

- $T(n) = 2\ T(n/2) + \mathrm{O}(n) \quad \rightarrow \mathrm{O}(n \log n)$
- $T(n) = 2\ T(n/2) + \mathrm{O}(n^2) \quad \rightarrow \mathrm{O}(n^2)$

- $T(n) = 8\ T(n/3) + \mathrm{O}(n^2) \quad \rightarrow \mathrm{O}(n^2)$
- $T(n) = 9\ T(n/3) + \mathrm{O}(n^2) \quad \rightarrow \mathrm{O}(n^2 \log n)$
- $T(n) = 10\ T(n/3) + \mathrm{O}(n^2) \quad \rightarrow \mathrm{O}(n^{\log_3 10}) = \mathrm{O}(n^{2,09})$

Multiplikation

- Multiplikation von zwei Zahlen nach Schulmethode
- Beispiel:

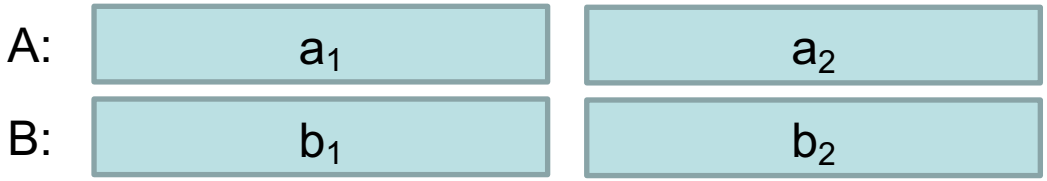
```
 1  2  3  4  5   *   6  7  8  9  0
      7  4  0  7  0
         8  6  4  1  5
            9  8  7  6  0
            1  1  1  1  0  5
+                  0  0  0  0  0
      8  3  8  1  0  2  0  5  0
```

- Komplexität
  - $O(n^2)$ – entspricht Größe der Tabelle
  - n: Anzahl der Dezimalstellen einer Zahl

Karatsuba Verfahren

Technische Hochschule Rosenheim

- nach Karatsuba und Ofman (1962)
- Idee: Zerlegung der Zahlen A und B in zwei Teile:



- Teilung erfolgt in der Mitte bei n/2 Stellen
- $A = a_1 10^{n/2} + a_2$ und $B = b_1 10^{n/2} + b_2$

- Produkt:
  $AB = (a_1 10^{n/2} + a_2)(b_1 10^{n/2} + b_2)$
  $= a_1 b_1 10^n + (a_1 b_2 + a_2 b_1) 10^{n/2} + a_2 b_2$
  - 4 n/2-stellige Multiplikationen
  - Kombination der Ergebnisse:
    - Shift um n/2 bzw. n Stellen
    - Addition

- Komplexität: $T(n) = 4\ T(n/2) + O(n)$

Karatsuba Verfahren

- Komplexität:
  $T(n) = 4\ T(n/2) + \mathrm{O}(n)$

- Ergebnis: 4 > 2 → Fall 3
  $\log_2 4 = 2$
  $T(n) = \mathrm{O}(n^2)$

$$T(n) = \begin{cases} \Theta(n^k) & \text{für } a < b^k \\ \Theta(n^k \log n) & \text{für } a = b^k \\ \Theta(n^{\log_b a}) & \text{für } a > b^k \end{cases}$$

- das ist nicht besser als vorher …

Karatsuba Verfahren

Technische Hochschule **Rosenheim**

- weitere Umformung:
  $AB = (a_1 10^{n/2} + a_2)(b_1 10^{n/2} + b_2)$
  $= a_1 b_1 10^n + (a_1 b_2 + a_2 b_1)\, 10^{n/2} + a_2 b_2$
  $= a_1 b_1 10^n + ((a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2) 10^{n/2} + a_2 b_2$
  - 3 n/2-stellige Multiplikationen (an Stelle von 4)
  - Kombination der Ergebnisse:
    - Shift um n/2 bzw. n Stellen
    - Addition

- Komplexität: $T(n) = 3\ T(n/2) + \mathrm{O}(n)$
- Ergebnis: 3 > 2 → Fall 3
  $\log_2 3 = 1{,}585$
  $T(n) = \mathrm{O}(n^{1{,}585})$
- das ist besser als vorher!

$$T(n) = \begin{cases} \Theta(n^k) & \text{für } a < b^k \\ \Theta(n^k \log n) & \text{für } a = b^k \\ \Theta(n^{\log_b a}) & \text{für } a > b^k \end{cases}$$

Karatsuba Verfahren – Anmerkungen

Technische Hochschule Rosenheim

- das gilt natürlich für beliebige Zahlensysteme
  - also auch für Basis 2 statt 10
- Es geht noch schneller
  - hat in der Praxis aber keine große Auswirkung
  - Schönhage-Strassen (1971): O(n log n log log n)
  - Fürer (2007): $O(n \text{ ld } n \, 2^{O(\text{ld}^* n)})$
    - mit ld* n = das kleinste i, für das bei i-maligem Hintereinanderschalten von ld (log zur Basis 2) gilt: ld ld … ld n ≤ 1
    - Beispiele:
      - ld* 2 = 1, ld* 4 = 2, ld* 16 = 3, ld* 65536 = 4
    - Veröffentlichung: https://wwwmath.uni-muenster.de/u/cl/WS2007-8/mult.pdf
  - Covanov und Thomé (2016): $O(n \text{ ld } n \, 2^{2\text{ld}^* n})$
    - Veröffentlichung: https://arxiv.org/abs/1502.02800
  - Harvey and van der Hoeven (2018): $O(n \text{ ld } n \, 2^{2\text{ld}^* n})$ ist eine untere Schranke für die Komplexität
    - Veröffentlichung: https://arxiv.org/abs/1802.07932

KOMPLEXITÄTSKLASSEN P – NP

Einführung

- Existenz eines Algorithmus zur Lösung eines Problems ist keine Garantie dafür, dass das Problem in der Praxis gelöst werden kann

- Fragen:
  - welche Komplexitätsordnungen kann man noch akzeptieren?
  - wie spezifiziert man die Klasse der praktisch handhabbaren Probleme?

Einführung

Problemgröße, die in 1 Stunde bewältigt werden kann

Komplexität	heute	mit 100mal schnellerem Computer	mit 1000mal schnellerem Computer
n	N_1	100 N_1	1000 N_1
n^2	N_2	10 N_2	32 N_2
n^3	N_3	4,6 N_3	10 N_3
n^5	N_4	2,5 N_4	4 N_4
2^n	N_5	N_5 + 6,6	N_5 + 10
3^n	N_6	N_6 + 4,2	N_6 + 6,3

Beobachtung: Bei exponentieller Komplexität bringt ein schnellerer Rechner praktisch nichts!

Die Klasse P

- Ein Problem heißt **effizient lösbar**, wenn es einen Algorithmus mit Zeitkomplexität O(p(n)) gibt
  - p(n) ist ein Polynom beliebigen Grades
- ein solcher Algorithmus hat **polynomielle** Laufzeit

- die **Klasse P** umfasst alle Algorithmen, die durch eine **deterministische** TM in polynomieller Zeit gelöst werden können

Die Klasse NP

- die **Klasse NP** umfasst alle Algorithmen, die durch eine **nichtdeterministische** TM in polynomieller Zeit gelöst werden können
  - NP steht für **N**ichtdeterministisch **P**olynomiell
- offensichtlich gilt: $P \subseteq NP$
  - jede deterministische TM ist auch eine nichtdeterministische TM, die keine Wahl bei Zustandsübergängen/Bewegungen hat
  - eine nichtdeterministische TM kann aber in polynomieller Zeit
    - eine exponentielle Anzahl an Lösungen „erraten“
    - und diese parallel überprüfen
- NP umfasst alle **effizient prüfbaren** Probleme
  - die nichtdeterministische TM „rät“ in polynomieller Zeit die Lösung
  - dieses kann dann in polynomieller Zeit von einer deterministischen TM auf Korrektheit geprüft werden

Beispiel

- Primfaktorisierung
  - gegeben: natürliche Zahl
  - gesucht: Zerlegung in Primfaktoren
  - Zerlegen ist aufwändig: Was sind die Primfaktoren von 8633?
  - Prüfen ist einfach
    - Faktoren: 89 * 97 = 8633
  - Anmerkung: Ob Primfaktorisierung wirklich schwierig ist, ist ein offenes Problem…

Beispiel

- Erfüllbarkeitsproblem der Aussagenlogik
  - gegeben: logischer Ausdruck
  - gesucht: für welche Variablenwerte ist der Ausdruck „wahr“
  - Suchen ist aufwändig: $(\neg x_1 \vee x_2) \wedge x_3 \wedge (x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$
  - Prüfen ist einfach: $x_1 = 0, x_2 = 1, x_3 = 1$
  - Anmerkung: dieses Problem ist nachweislich schwierig…

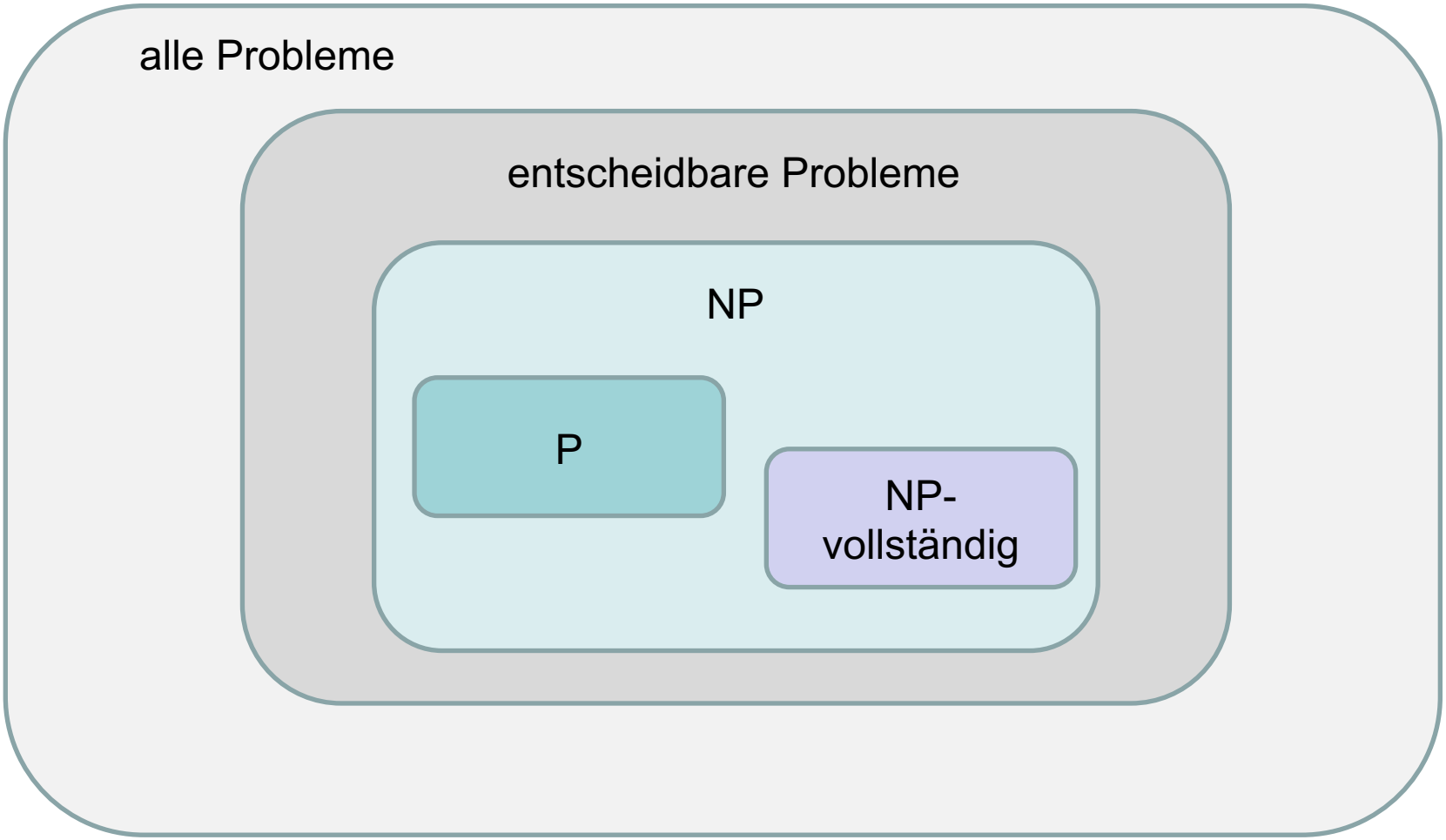
P = NP?

- die wichtigste Frage der theoretischen Informatik:
  Ist P = NP?
  - sind die zwei Problemklassen vielleicht gar nicht verschieden?
  - dieses Problem ist seit den 1970er Jahren offen und bisher ungelöst
  - es wurde 2000 in die Liste der Millenium-Probleme aufgenommen
    - enthält 7 ungelöste Probleme der Mathematik (mittlerweile noch 6)
    - auf die Lösung ist ein Preisgeld von 1 Million US-Dollar ausgesetzt
      http://www.claymath.org/millennium/P_vs_NP/

- Praktische Bedeutung
  - es gibt sehr viele Probleme
    - von denen man leicht zeigen kann, dass sie in NP liegen
    - für die aber bisher kein polynomieller Algorithmus bekannt ist
  - es könnte sein, dass man einfach noch keinen gefunden hat (P = NP)
  - oder es gibt keinen (P ≠ NP)

- Vermutung: P ≠ NP

Überblick



Annahme: P ≠ NP

NP-schwere Probleme

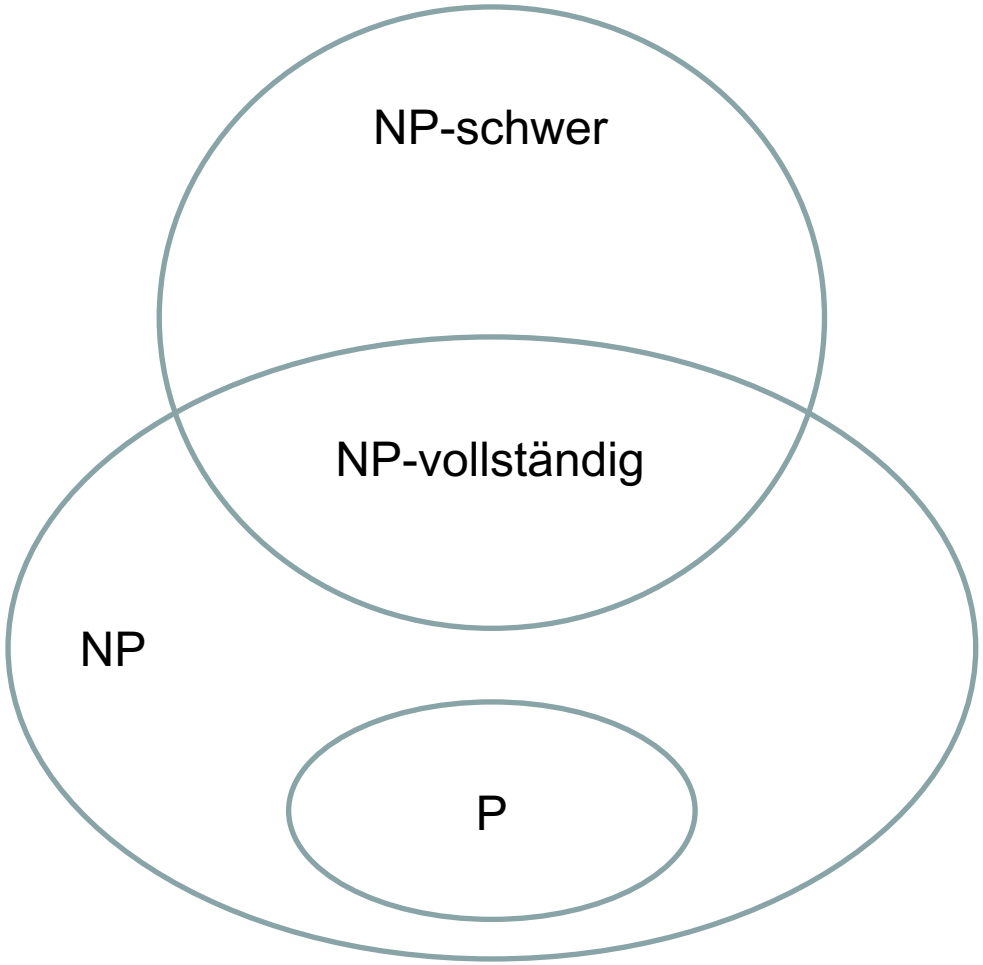
Technische Hochschule **Rosenheim**

- Polynomiale Reduktion
  - Ein Problem A heißt **polynomial reduzierbar** auf B, wenn es einen Algorithmus mit polynomieller Komplexität gibt, der A in B umformt: $x \in A \Leftrightarrow f(x) \in B$
  - schreibweise: $A \leq_p B$
  - damit ergibt sich insbesondere:
    - wenn $A \leq_p B$ und $B \in P$ (oder $B \in NP$)
    - dann ist auch $A \in P$ (bzw. $A \in NP$)

- Ein Problem X heißt **NP-schwer** (oder NP-hart), wenn es mindestens so schwierig ist wie jedes Problem in NP
  - d.h., für alle Probleme $L \in NP$ gilt: $L \leq_p X$

- Ein Problem X heißt **NP-vollständig**, wenn es NP-schwer ist und in NP liegt

NP-Vollständigkeit



Annahme: P ≠ NP

NP-Vollständigkeit

- NP-vollständig
  - NP-schwere Probleme, die vollständig in NP liegen
  - die schwierigsten Probleme der Klasse NP
- Ist auch nur ein einziges NP-vollständiges Problem in P, dann gilt P = NP
  - alle Probleme in NP können dann ja polynomial darauf reduziert werden
  - ein Nachweis der NP-Vollständigkeit für ein Problem ist damit praktisch gleichbedeutend damit, dass es (wahrscheinlich) keine effizienten Algorithmen für dieses Problem gibt
- wenn man ein erstes NP-vollständiges Problem A hat, kann man die NP-Vollständigkeit anderer Probleme durch polynomiale Reduktion auf A zeigen

SAT

- gibt es überhaupt NP-vollständige Probleme?

- Ja: Das **Erfüllbarkeitstheorem der Aussagenlogik** SAT
  - das erste Problem, von dem NP-Vollständigkeit nachgewiesen wurde
  - Beweis 1971 von S. Cook
    - „The Complexity of Theorem Proving Procedures"
    - 1982 erhielt er dafür den Turing-Award
  - gegeben: Aussagenlogische Formel F
  - gesucht: ist F erfüllbar? Also: Gibt es eine Variablenbelegung aus {0, 1}, so dass F den Wert 1 annimmt?

- Beweis besteht aus zwei Teilen
  - SAT $\in$ NP (nicht so schwierig)
    - Prinzip: NTM „errät" Lösung und prüft die Korrektheit (in polynomieller Zeit)
  - SAT ist NP-schwer (schon schwieriger…)
  - Details siehe Literatur

SAT

- **jedes** Problem in NP ist auf SAT reduzierbar
- deterministische Algorithmen zur Berechnung von SAT haben exponentielle Komplexität $2^{O(n)}$
  - typische Lösung: alle Variablenbelegungen durchprobieren
- damit ergibt sich eine obere Abschätzung der Komplexität für alle Probleme in NP durch $2^{p(n)}$
  - p(n) ist ein Polynom

- Anmerkung:
  - betrachtet werden **Entscheidungsprobleme**
    - also Fragen nach „Gibt es …?“
  - das Finden der tatsächlichen Lösung kann noch schwieriger sein

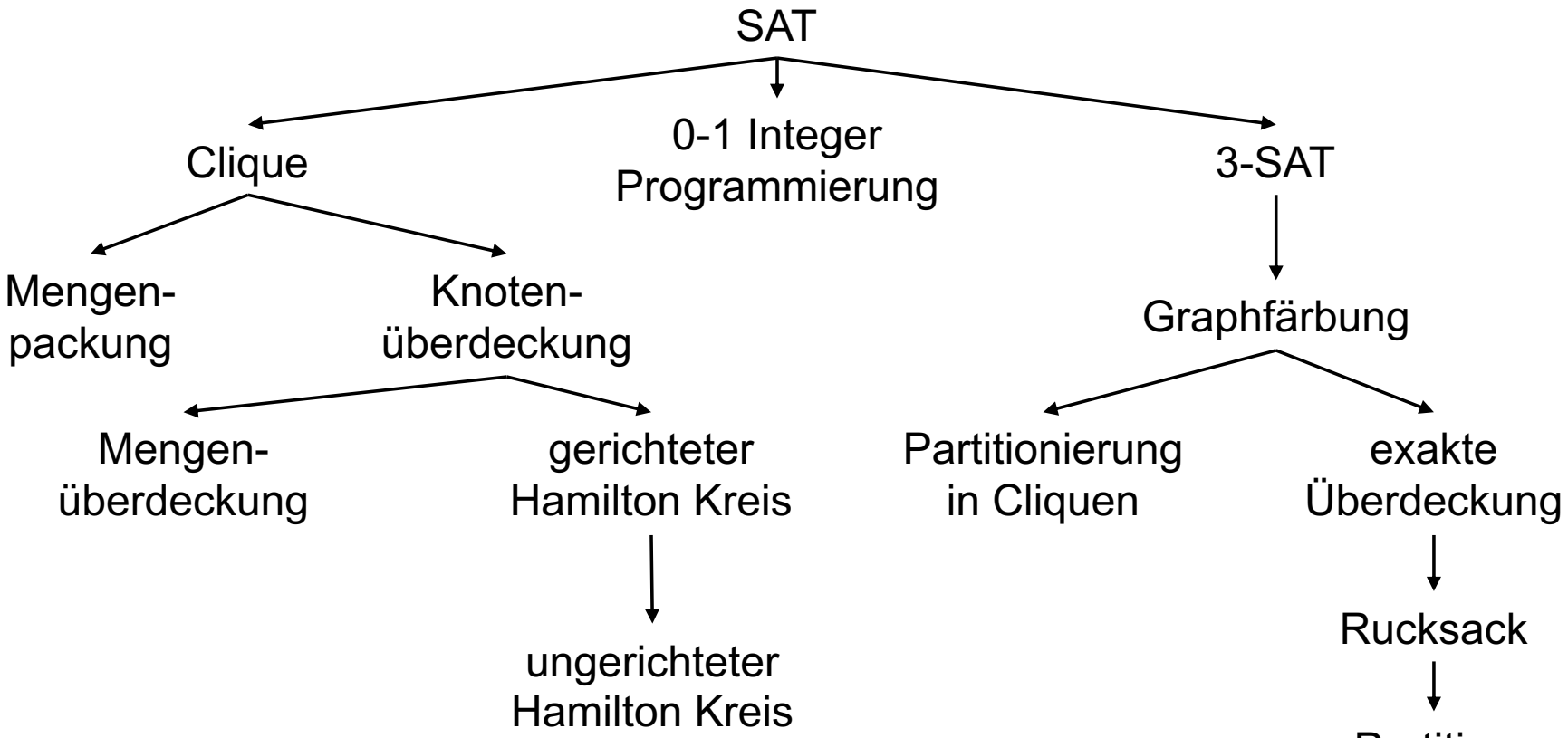
Konsequenzen aus NP-Vollständigkeit von SAT

Technische Hochschule **Rosenheim**

- es sind **mehrere tausend** NP-vollständige Probleme bekannt
  - diese sind oft auf den ersten Blick sehr verschieden
  - eine Auswahl findet man z.B. hier: http://en.wikipedia.org/wiki/List_of_NP-complete_problems
- findet man für **irgendeines** davon einen Algorithmus mit polynomieller Laufzeit, dann
  - hat man automatisch für alle Probleme in NP einen solchen
  - gezeigt, dass P = NP gilt
- und:
  - kann man von irgendeinem davon zeigen, dass es nicht in P liegt, so liegt keines in P, und es gilt P ≠ NP

Karps NP-vollständige Probleme

Technische Hochschule **Rosenheim**



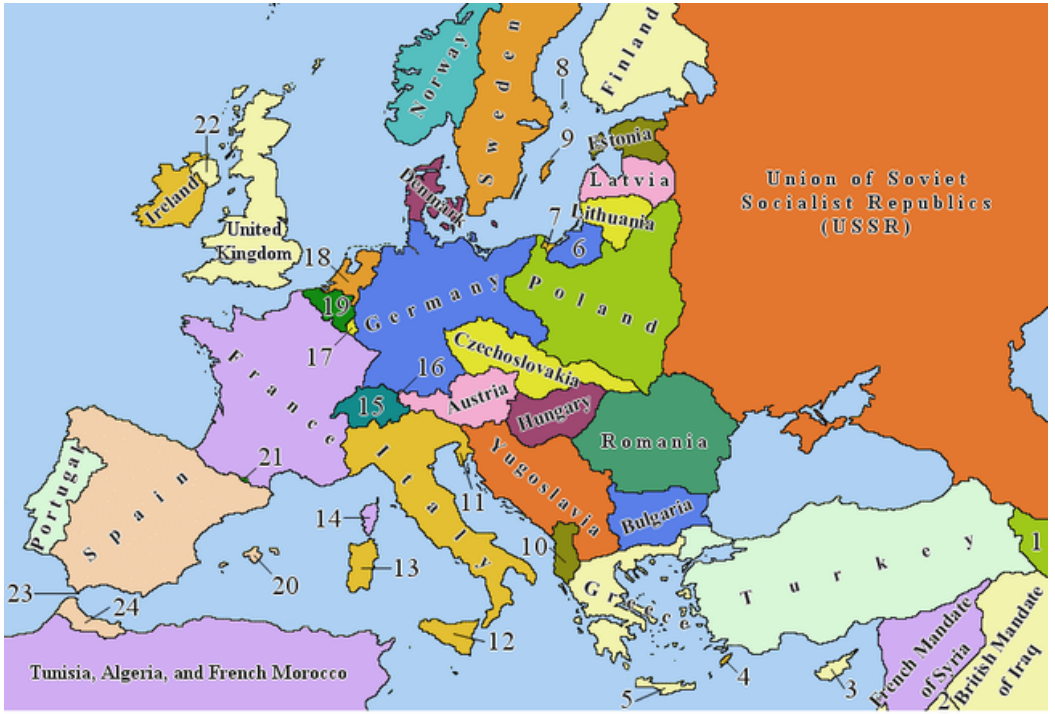
Auswahl der 21 NP-vollständigen Probleme und ihrer Reduktionen von R. Karp, 1972: „Reducibility Among Combinatorial Problems“

3-SAT

- starke Einschränkung von SAT
- gegeben: Aussagenlogische Formel F in konjunktiver Normalform (KNF) mit höchstens 3 Variablen pro Term
- gesucht: ist F erfüllbar? Also: Gibt es eine Variablenbelegung aus {0, 1}, so dass F den Wert 1 annimmt?
- es kann gezeigt werden: SAT $\leq_p$ 3-SAT
  - 3-SAT ist NP-vollständig
- Anmerkungen:
  - jede Formel kann in KNF umgeformt werden
  - diese erfordert allerdings exponentiellen Aufwand
  - gefordert ist durch die polynomielle Reduktion aber keine exakte Äquivalenz
  - sondern lediglich: wenn F erfüllbar, dann ist auch die umgeformte Formel F‘ erfüllbar (und umgekehrt)
- alle k-SAT Probleme mit k ≥ 3 sind NP-vollständig
- 2-SAT dagegen liegt in P

Färben von Landkarten

Kann man eine Landkarte mit k Farben so einfärben, dass benachbarte Länder immer verschiedenfarbig sind?



Europe 1929-1939

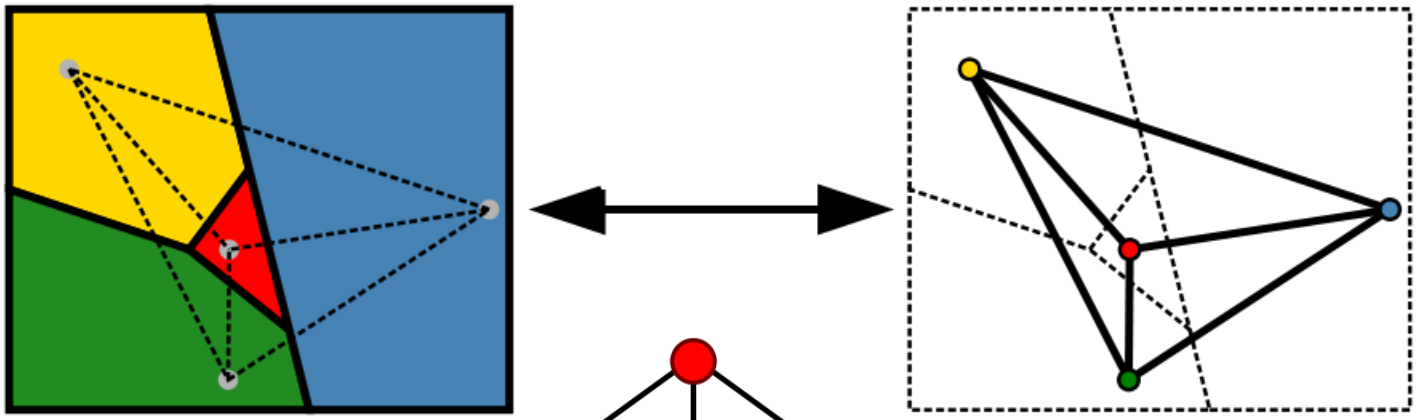
Legend

1) Persia (Iran)
2) British Mandate of Palestine
3) Cyprus (British Crown Colony)
4) Rhodes and Dodecanese (Italy)
5) Crete (Greece)
6) East Prussia
7) Free City of Danzig
8) Aland Islands (Finland)
9) Gotland (Sweden)
10) Albania
11) Istria (Italy)
12) Sicily (Italy)
13) Sardinia (Italy)
14) Corsica (France)
15) Switzerland
16) Liechtenstein
17) Luxembourg
18) Netherlands
19) Belgium
20) Balearic Islands (Spain)
21) Andorra
22) Northern Ireland (United Kingdom)
23) Gibraltar (British Crown Colony)
24) Spanish Morocco (Spain)

[Hum]

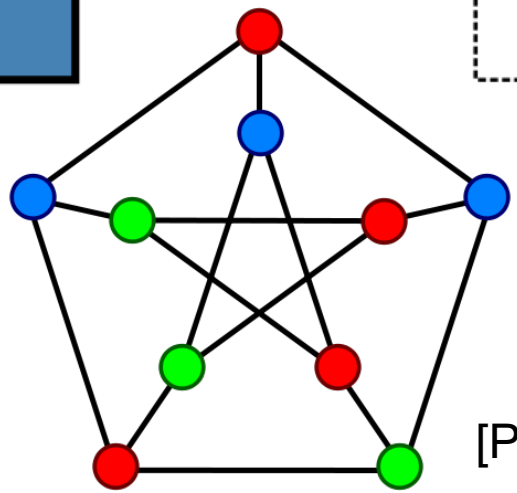
Graphfärbung

- entspricht dem Problem der Graphfärbung
  - wobei die Knoten zu färben sind
  - und die Kanten die Nachbarschaft definieren



[Gra]

nicht planarer Graph



[Pet]

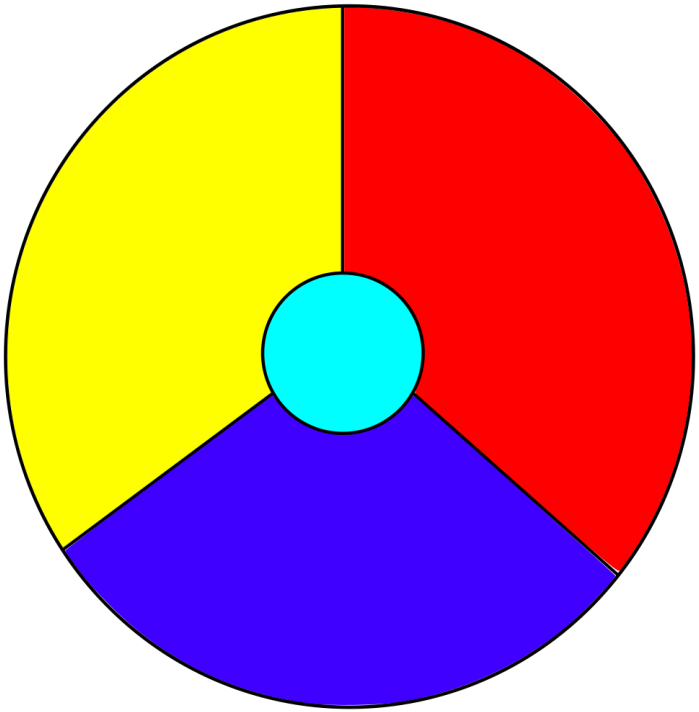
Graphfärbung

- allgemeine Graphen
  - k-Färbbarkeit für k ≥ 3 ist NP-vollständig
  - 2-Färbbarkeit dagegen liegt in P
- planare Graphen
  - 2-Färbbarkeit ist in P
  - 3-Färbbarkeit ist NP-vollständig
  - 4-Färbbarkeit hat **konstante Laufzeit**!

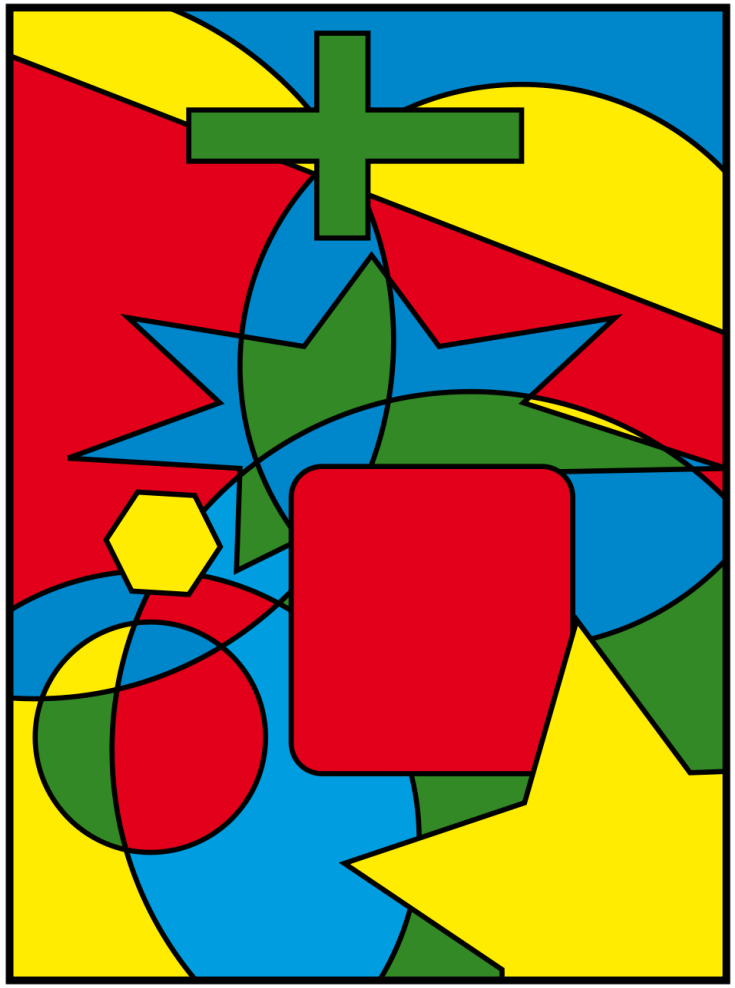
4-Farben Problem

- 4-Farben genügen **immer**, um einen planaren Graphen (Landkarte) einzufärben
- Vermutung bestand seit 1852
- eines der ersten Probleme, das mit Hilfe eine Computersystems bewiesen wurde (1976)
- ein formaler Beweis mit Hilfe eines Theorembeweisers folgte 2004

4-Farben Problem

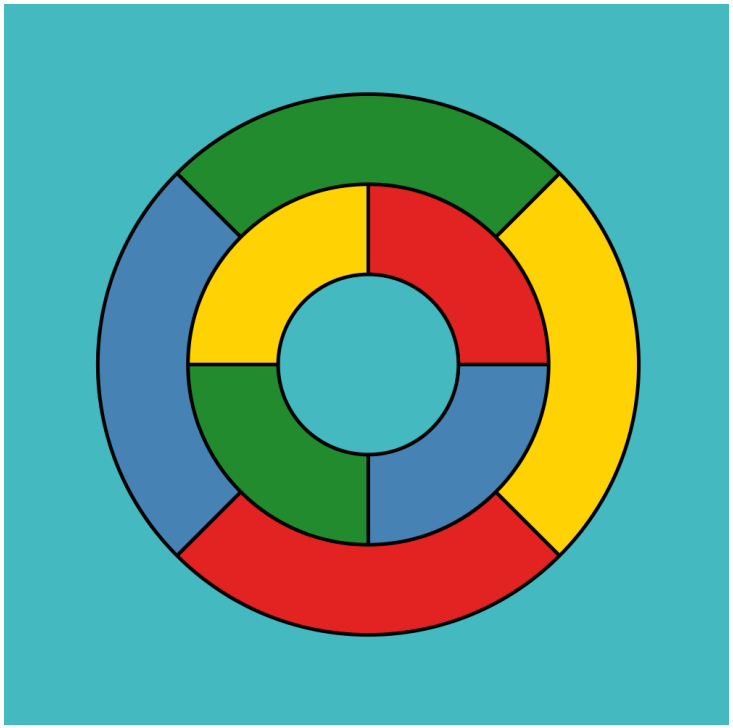


[4FP1]

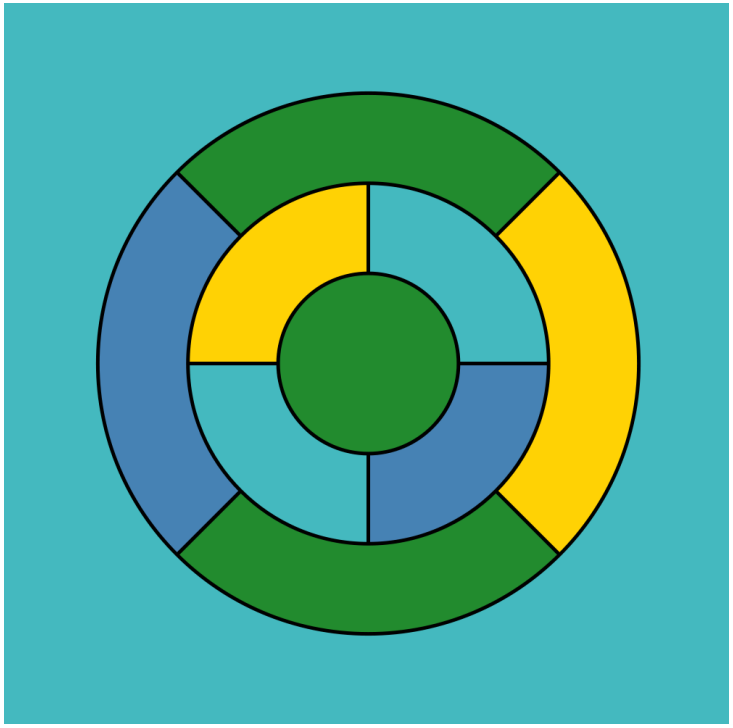


[4FP2]

4-Farben Problem



Färbung mit 5 Farben



es genügen aber 4

[4FP3]

Graphfärbung

- außer dem Färben von Landkarten noch viele weitere Anwendungen
- Ablaufplanung
  - Prozessplanung in Betriebssystemen
  - Zuweisung von Flugzeugen zu Flügen
  - Zuweisung von Bandbreite an Radio-/Fernsehsender, Mobilfunkbetreiber, …
- Compilerbau
  - welche Variablenwerte werden in Registern gehalten?
- Erstellen von Stundenplänen
- Sudoku
  - spezieller Graph, 81 Knoten, 9 Farben

Problem des Handlungsreisenden

Technische Hochschule **Rosenheim**

- TSP: Travelling Salesman Problem
  - gegeben: n Städte, sowie die Entfernungen (km, Zeit, Kosten …) dazwischen
  - Frage: Welche Städtefolge ist die kürzeste Rundreise?
    - alle Städte sollen genau einmal enthalten sein
    - bzw. als Entscheidungsproblem: Gibt es eine Rundreise mit Länge kleiner einer gegebenen Konstanten k?

- entspricht Hamilton Kreisen in Graphen
  - jede Stadt ist ein Knoten
  - jede Verbindung zwischen Städten ist eine Kante
  - die Entfernung entspricht einem Kantengewicht
  - Hamilton Kreis hat genau so viele Kanten wie Knoten

TSP

- TSP (Entscheidungsproblem) ist NP-vollständig
  - die Zeitkomplexität der naiven Lösung beträgt sogar O(n!)
  - gute Algorithmen verringern dies auf $O(2^n)$
- TSP (tatsächliche Lösung) ist NP-schwer

- wie aufwändig ist O(n!)?
  - angenommen, man braucht für 10 Städte 1 Sekunde
  - dann braucht man für 20 Städte 670 442 572 800 Sekunden
    - das sind 21259 **Jahre**

TSP – Beispiel

- Rundreise durch die 15 größten Städte Deutschlands
- es gibt 14! / 2 verschiedene Wege
  - 14! / 2 = 43 589 145 600
- die unten gezeigte ist die kürzeste Rundreise

[TSP]

TSP – wieviel geht?

- Rundreise durch 15112 deutsche Städte (2001)
  - Verwendung von 110 Prozessoren
  - äquivalente Rechenzeit (500MHz Alpha CPU): **22,6 Jahre**
- Rundreise durch 24978 schwedische Städte (2004)
  - Länge: 72500 km
  - Linux-Cluster mit 96 Intel Xeon 2,8GHz CPUs (dual core)
  - äquivalente Rechenzeit (2,8GHz dual core Xeon): **84,8 Jahre**
- Layout von elektronischen Schaltungen
  - 85900 Knoten (2005/06) – der bisherige Rekord für TSP
  - äquivalente Rechenzeit (2,4GHz AMD Opteron): **136 Jahre**

Fazit

alle NP-vollständigen Probleme sind

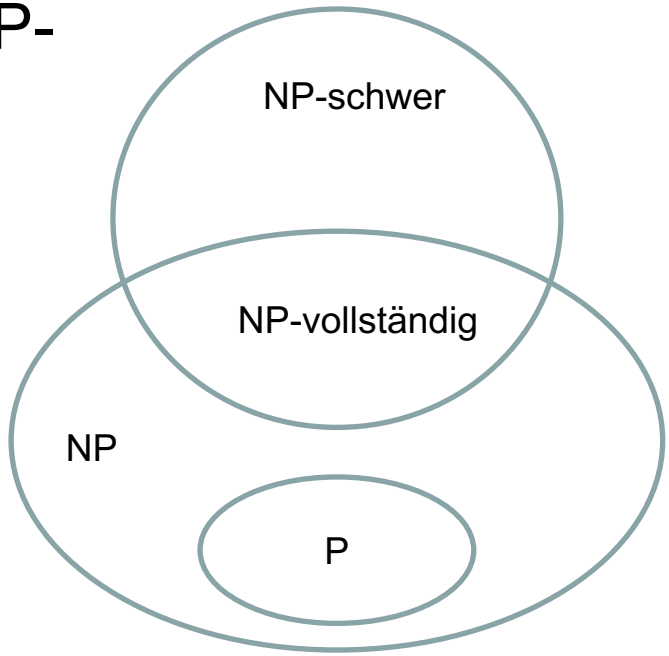
- in polynomialer Zeit aufeinander reduzierbar
- also tatsächlich nur verschiedene Varianten ein und desselben Problems – so verschieden sie auch aussehen mögen

WEITERE PROBLEMKLASSEN

NP-schwere Probleme außerhalb von NP

Technische Hochschule **Rosenheim**

- Nachweis, dass Probleme in NP liegen gelingt hier nicht
- diese sind also noch schwieriger als NP-vollständige Probleme

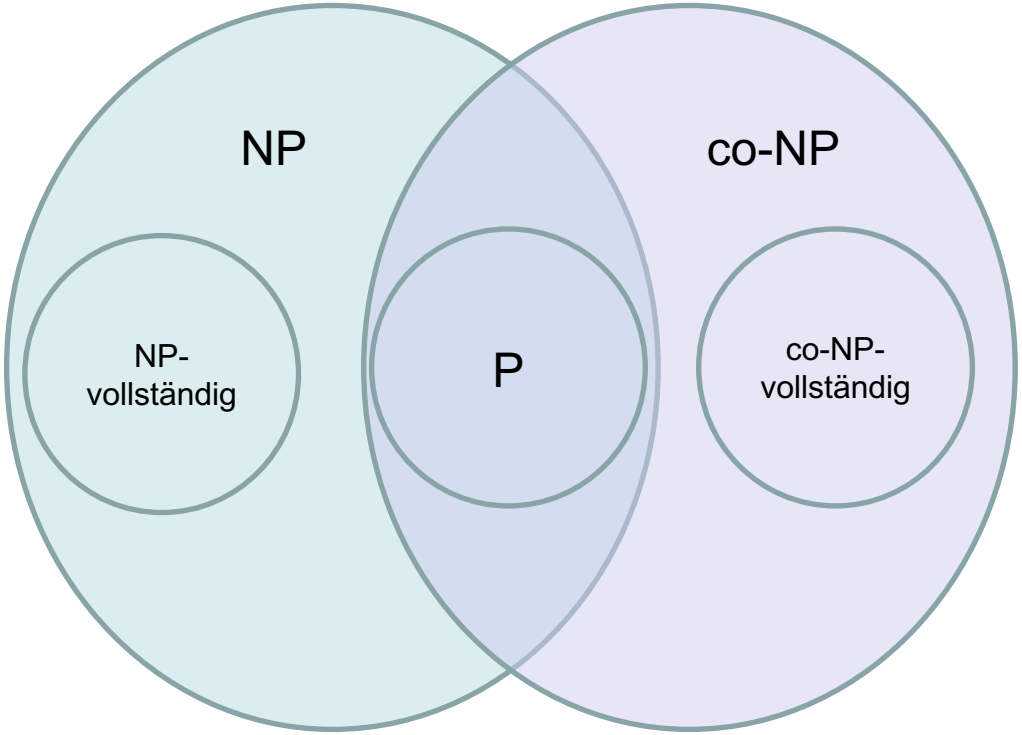


- Beispiele:
  - Wortproblem für Typ-1 Sprachen
  - Inäquivalenz für reguläre Ausdrücke
    - und damit: reguläre Grammatiken bzw. nichtdeterministische endliche Automaten
    - Äquivalenz von deterministischen endlichen Automaten ist in P
      - Umformung nicht-det. → det. erfordert Konstruktion der Potenzmenge
      - und hat damit exponentielle Komplexität

co-NP

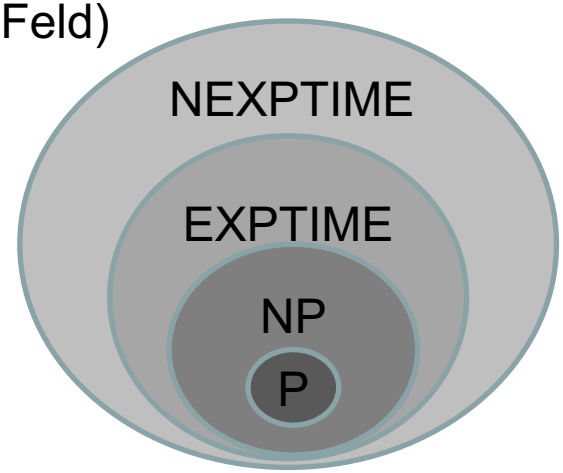
- co-NP: Menge der Entscheidungsprobleme, deren **Komplemente** in NP enthalten ist
- Beispiel:
  - „Ist eine Zahl prim?“ ist in NP
  - „Ist eine Zahl nicht prim (= zusammengesetzt)?“ ist in co-NP
- Vermutung: NP ≠ co-NP
  - sollte man für ein NP-vollständiges Problem nachweisen können, dass es sowohl in NP als auch in co-NP liegt gilt: NP = co-NP
  - bisher hat man keines gefunden, daher die Vermutung
- Im Fall P = NP gilt NP = co-NP
  - da P bzgl. Komplementbildung abgeschlossen ist: P = co-P
- Primzahltest ist übrigens in NP **und** co-NP
  - ein starkes Indiz dafür, dass ein Problem nicht NP-vollständig ist
  - tatsächlich ist Primzahltest in P

co-NP – Überblick



EXPTIME und NEXPTIME

- EXPTIME
  - Menge aller Entscheidungsprobleme, die von einer deterministischen TM in der Zeit $O(2^{p(n)})$ gelöst werden können
    - p(n) ist ein Polynom
  - es gibt EXPTIME-vollständige Probleme, z.B.
    - modifiziertes Halteproblem: Hält eine det. TM nach höchstens k Schritten?
    - Stellungsanalyse für generalisiertes Schach, Dame, Go (beliebig viele Spielfiguren auf beliebig großem Feld)
- NEXPTIME
  - entsprechend für nichtdeterministische TM
- Anmerkungen
  - wenn P = NP, dann EXPTIME = NEXPTIME
  - es gilt: $P \subsetneq EXPTIME$ und $NP \subsetneq NEXPTIME$



PSPACE und NPSPACE

Technische Hochschule Rosenheim

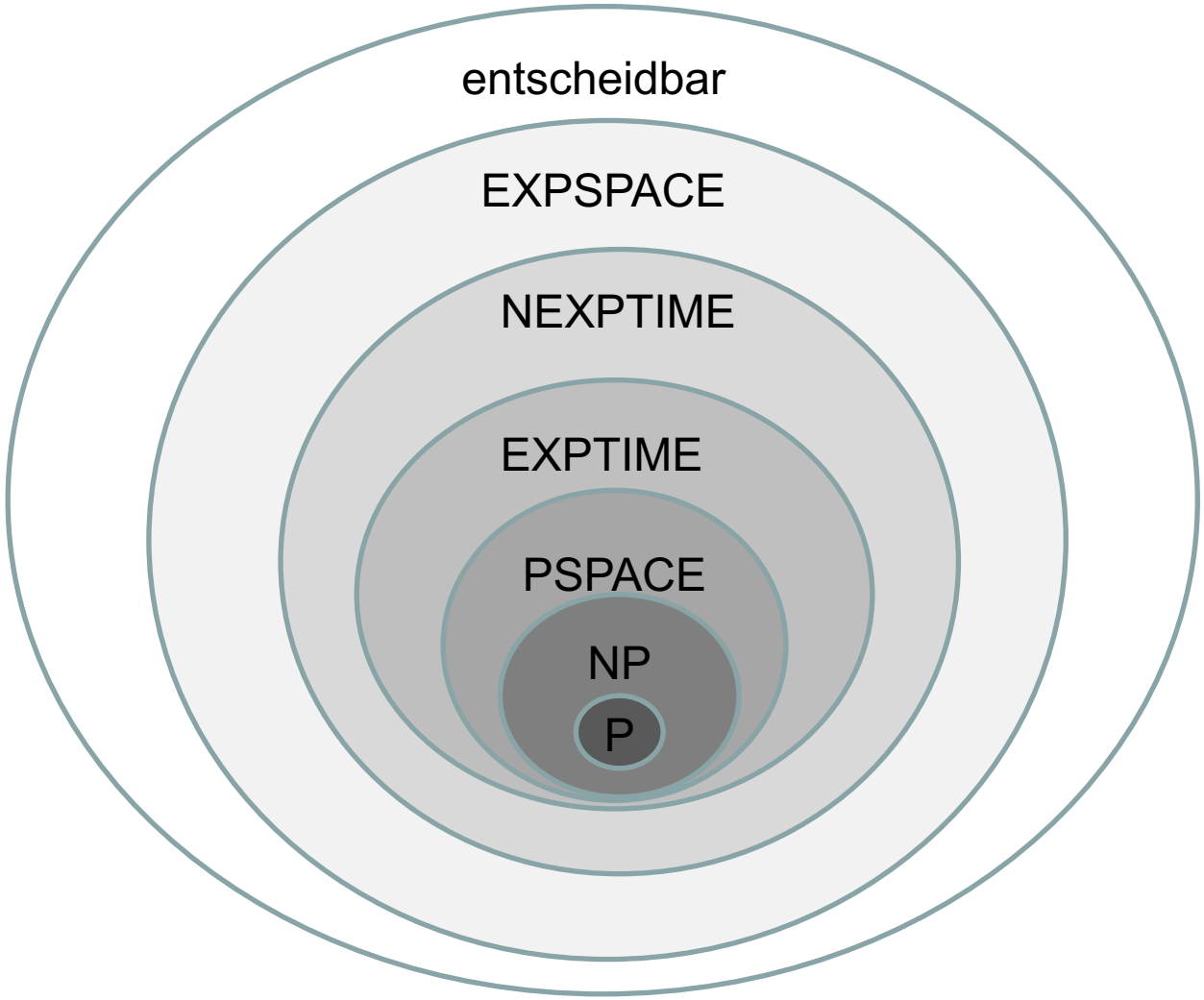
- PSPACE
  - Menge aller Entscheidungsprobleme, die von einer deterministischen TM mit polynomiellen Speicher gelöst werden können
- NPSPACE
  - entsprechend für nichtdeterministische TM
- Offensichtlich: $P \subseteq PSPACE$ und $NP \subseteq NPSPACE$
  - eine TM kann mit einer polynomiellen Anzahl an Bewegungen (Zeit) höchstens polynomiell viele Zeichen auf das Band schreiben
- Tatsächlich kann man nachweisen: PSPACE = NPSPACE
- es gibt PSPACE-vollständige Probleme, z.B.
  - Wortproblem für Typ-1 Sprachen
  - Erfüllbarkeit boolescher Formeln mit Quantoren ($\forall$, $\exists$)

EXPSPACE und NEXPSPACE

- EXPSPACE
  - Menge aller Entscheidungsprobleme, die von einer deterministischen TM mit $O(2^{p(n)})$ Speicher gelöst werden können
    - p(n) ist ein Polynom
- NEXPSPACE
  - entsprechend für nichtdeterministische TM
- Es gilt
  - EXPSPACE = NEXPSPACE
  - PSPACE $\subsetneq$ EXPSPACE
  - EXPTIME $\subseteq$ EXPSPACE (vermutlich: EXPTIME $\subsetneq$ EXPSPACE)
- es gibt EXPSPACE-vollständige Probleme, z.B.
  - definieren zwei gegebene reguläre Ausdrücke verschiedene Sprachen?

Komplexitätsklassen – Überblick

Technische Hochschule Rosenheim



Es gibt noch mehr davon…

- für probabilistische Algorithmen
- unterhalb von P
- für Quantencomputer
- zur Betrachtung der Berechnung einer Lösung an Stelle des Entscheidungsproblems

Zusammenfassung

- O-Notation
  - gilt asymptotisch
- Komplexitätsordnung
  - typische Trennung zwischen polynomieller und exponentieller Komplexität
  - in der Praxis wird es bereits ab ca. $O(n^4)$ schwierig
- Komplexitätsklassen
  - P: Entscheidungsprobleme, die durch det. TM in polynomieller Zeit gelöst werden können
  - NP: wie P für nichtdet. TM → entspricht $O(2^{p(n)})$ für det. Algorithmen
- NP-Vollständigkeit
  - Probleme, die vollständig in NP liegen
  - ob P = NP ist eines der großen ungelösten Probleme der Informatik
  - Vermutung: P ≠ NP
  - es gibt sehr viele NP-vollständige Probleme mit praktischer Relevanz

Quellen

Die Folien entstanden auf Basis folgender Literatur

- H. Ernst, J. Schmidt und G. Beneken: Grundkurs Informatik. Springer Vieweg, 6. Aufl., 2016.
- Hopcroft, J.E., Motwani, R. und Ullmann, J.D.: *Einführung in die Automatentheorie, formalen Sprachen und Komplexitätstheorie*. Pearson Studium (2002)
- Schöning, U.: *Theoretische Informatik - kurz gefasst*. Spektrum Akad. Verlag (2008)
- Sedgewick, R.: *Algorithmen in C++*, Addison-Wesley (1992)

Quellenangaben Bilder

Technische Hochschule Rosenheim

[Hum] Wikimedia.org, Autor: Jan Humpolík
http://commons.wikimedia.org/wiki/File:EUROPE_1919-1929_POLITICAL_01.png
Lizenz: [1]

[Gra] Wikimedia.org, Autor: Inductiveload
http://commons.wikimedia.org/wiki/File:Four_Colour_Planar_Graph.svg
Lizenz: [1]

[Pet] Wikimedia.org, Autor: Jan Humpolík
http://commons.wikimedia.org/wiki/File:Petersen_graph_3-coloring.svg
Lizenz: Public Domain

[4FP1] Wikimedia.org, Autor: Germo
http://commons.wikimedia.org/wiki/File:Fourcolorsmap.svg
Lizenz: [1]

[4FP2] Wikimedia.org, Autor: Inductiveload
http://commons.wikimedia.org/wiki/File:Four_Colour_Map_Example.svg
Lizenz: [1]

[4FP3] Wikimedia.org, Autor: Dmharvey
http://commons.wikimedia.org/wiki/File:4CT_Non-Counterexample_1.svg
http://commons.wikimedia.org/wiki/File:4CT_Non-Counterexample_2.svg
Lizenz: Public Domain

[TSP] Wikimedia.org, Autor: MrMonstar / CIA
http://commons.wikimedia.org/wiki/File:TSP_Deutschland_3.png
Lizenz: Public Domain

[1] Attribution-ShareAlike 3.0 Unported
http://creativecommons.org/licenses/by-sa/3.0/deed.en