DHBW
Duale Hochschule
Baden-Württemberg
Stuttgart
Campus Horb

Algorithmen und Datenstrukturen

Teil 2:
Komplexitätstheorie – Laufzeit von Algorithmen

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 04/2018

Gliederung

- Komplexitätstheorie
- Beispiele zur Bewertung von Algorithmen
- Nicht-Determinismus, Sprachklassen P, NP

Motivation

- Bisher in VLen Programmierung und Informatik:
  - Algorithmen und Datenstrukturen kennen gelernt
  - Einige Probleme verschiedene Lösungen
  - Einige „besser“, andere „weniger gut“
  - Beispiel: Fibonacci-Zahlen iterativ vs. rekursiv
- Nun:
  - Algorithmen systematisch bewerten
  - Möglichst abstraktes Niveau
  - D.h. ohne Kenntnisse der konkreten Programmiersprache, der konkreten Hardware usw.

Methoden der Effizienzberechnung (I)

- Methode 1: Sowohl Rechner/Maschine als auch Eingabedaten festgelegt
  - Methode der Beurteilung: Zeitmessung
  - Problem: Abhängigkeit vom Rechner und den Eingabedaten, Aussage kann immer nur für konkreten Fall getroffen werden
- Methode 2: Rechner/Maschine abstrakt, Eingabedaten festgelegt
  - Methode der Beurteilung: Zählen von Operationen
  - Problem: Abhängigkeit von den Eingabedaten, Aussage kann immer nur für konkreten Fall getroffen werden
- Methode 3: Sowohl Rechner/Maschine als auch Eingabedaten abstrakt
  - Methode der Beurteilung: Berechnung (der Anzahl der Operationen)
  - Problem: Abstrakte Beschreibung ist zwar erreicht, aber Resultat ist immer noch vom Parameter der Eingabedaten abhängig

Methoden der Effizienzberechnung (II)

- Beispiel:
  - Zwei Algorithmen A_1 und A_2 berechnen das gleiche Problem
  - Sei dabei V Anzahl der Vergleiche und W Anzahl der Wertzuweisungen
  - A_1 benötigt $V_1=W_1=n\cdot\log(n)$ und A_2 benötigt $V_2=n^2$ und $W_2=n$
  - Welcher Algorithmus ist nun schneller?
  - Ohne Kenntnisse von n und Ausführungszeiten für Vergleich und Wertzuweisung können immer noch keine Aussagen getroffen werden
  - Dennoch: Wenn n genügend groß ist, dann wird auf jeden Fall A_1 schneller sein, weil sein Aufwand nur logarithmisch wächst, während sich bei A_2 dann der quadratische Anteil bemerkbar macht
- Methode 4: Rechner/Maschine abstrakt, Eingabedaten $\rightarrow \infty$
  - Methode der Beurteilung: Asymptotische Abschätzung

Komplexitätstheorie (I)

- Beschäftigt sich mit Maßen, die die Effizienz eines Algorithmus beschreiben
- Stellt Modelle zur Verfügung, um a priori Aussagen über die Laufzeit und den Speicheraufwand eines Algorithmus machen zu können
- Damit können Algorithmen verglichen werden, ohne konkret realisiert zu werden

- Komplexitätsmaß:
  - Ideal wäre als Maß eine Funktion, die jeder Eingabe die Zeit zuordnet, die zur Ausgabe benötigt wird
  - Damit wäre man aber wieder von der konkreten Umgebung (Rechner, Compiler, ...) abhängig
  - Abstraktion gelingt an dieser Stelle, indem keine konkrete Zeit, sondern die Anzahl der notwendigen Rechenschritte betrachtet wird
  - Rechenschritt: Arithmetische Grundoperation, Speicherzugriff, Vergleiche, Wertzuweisungen, ...

Komplexitätstheorie (II)

- Charakterisierung der Eingabe erfolgt durch Angabe der Länge (Zahl der zu verarbeitenden Zeichen)
- Beispiele:
  - Länge einer Zeichenkette, die verarbeitet werden muss
  - Anzahl der Knoten als Größe eines Graphen
  - Anzahl der zu sortierenden Elemente bei Sortieralgorithmen
- Wie schon bei der Zeitangabe erfolgt auch hier keine quantitative Aussage über die Länge eines Zeichens, diese muss lediglich konstant sein
- Sowohl bei Zeitangabe als auch bei der Eingabe bezieht man sich auf relative Größen
- Damit sind Aussagen der Form „Wenn die Länge der Eingabe proportional zu n ist, ist die Laufzeit des Algorithmus proportional zu n^2“ möglich

Komplexitätstheorie (III)

- Unterschieden werden kann die Komplexität für folgende Fälle:
  - Bester Fall (best case complexity): Minimale Laufzeit bzw. minimaler Speicherplatz für eine beliebige Eingabe der Länge n
  - Durchschnittlicher Fall (average case complexity): Durchschnittliche Laufzeit bzw. durchschnittlicher Speicherplatz für eine beliebige Eingabe der Länge n
  - Schlechtester Fall (worst case complexity): Maximale Laufzeit bzw. maximaler Speicherplatz für eine beliebige Eingabe der Länge n
- Meistens wird nur der letzte Fall betrachtet, weil
  - Nicht für alle Eingaben der Länge n muss ein Algorithmus gleichviel Zeit benötigen (siehe nächste Folie)
  - Berechnung der average case complexity ist für viele Algorithmen in der Praxis mathematisch sehr anspruchsvoll
    - Stichwort: Annahmen über Wahrscheinlichkeitsverteilungen
    - So ist es bis heute für viele Algorithmen nicht gelungen die average case complexity zu bestimmen
  - Für eine Reihe von Algorithmen sind average case und worst case complexity gleich

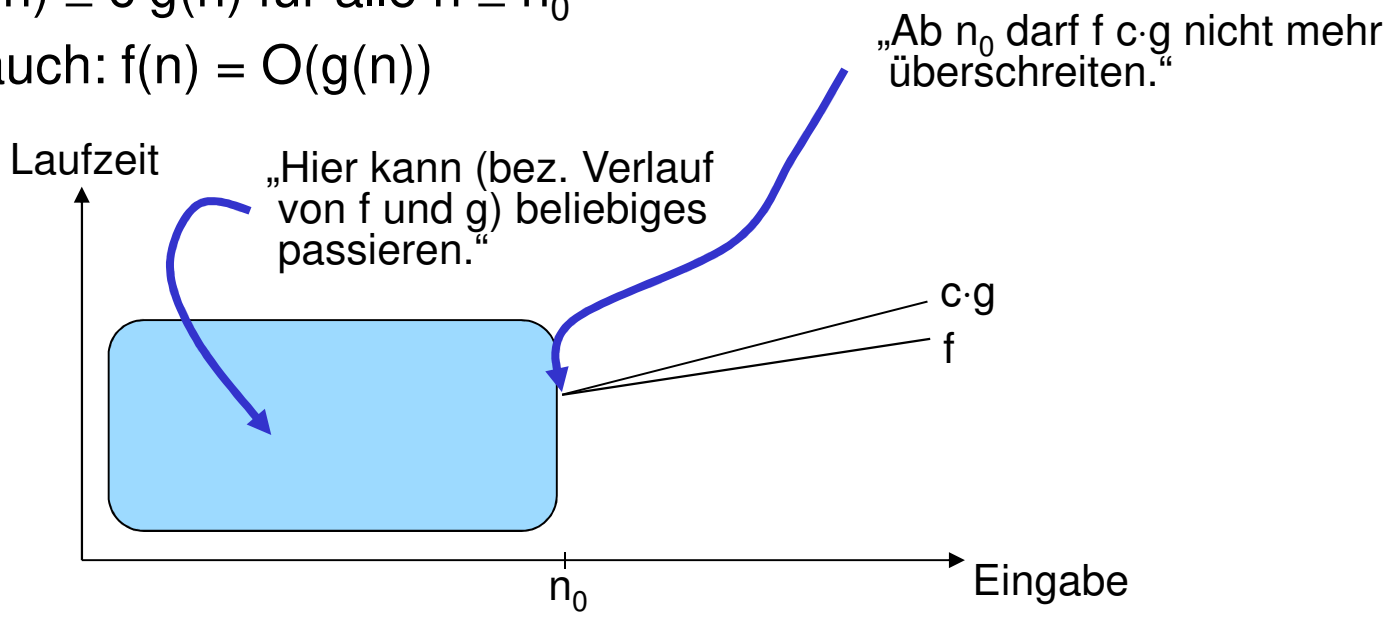
Einfluss der Eingabelänge auf Zeitkomplexität

- Warum muss ein Algorithmus nicht für alle Werte von n gleiche Zeitkomplexität haben?
- Sei für einen Algorithmus A der Aufwand $T(n) = 100 \cdot n + n^2 + 10^{n/1000-2}$
  - Für Werte kleiner als 100 dominiert der lineare Summand
  - Darüber dominiert zunächst der quadratische Summand
  - Erst ab Werten größer als 10000 kommt die exponentielle Komponente ins Spiel
- Ohne Kenntnis von T kann man den Aufwand nicht bestimmen
- Problem: Wie weit muss man messen?

Komplexitätstheorie (IV)

- Platzbedarf und Laufzeit werden als Funktion $f : N \rightarrow R^+$ angegeben
- Um die Bestimmung der Funktion zu erleichtern, verzichtet man auf die Angabe des genauen Wertes f(n) und gibt nur den qualitativen Verlauf, d.h. die Größenordnung an
- Dies geschieht mit dem Landauschen Symbol O
- Seien f und g Funktionen von N nach R^+.

  f hat die Ordnung von g, wenn es eine Konstante $c > 0$ und ein $n_0 \in N$ gibt, so dass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$
- Man schreibt auch: $f(n) = O(g(n))$
- Anschaulich:



Zeitkomplexitätsklassen

- Sei A ein Algorithmus mit der Eingabe der Länge n
- S(w) := Anzahl der Schritte/Operationen, die A bei einer Eingabe w benötigt, bis er anhält
- $s_A(n)$:= Maximale Anzahl an Schritten zur Verarbeitung einer Eingabe der Länge n
- Anmerkung: Max. Anzahl, weil worst case betrachtet werden soll
- Unterschieden werden drei zentrale Zeitkomplexitätsklassen:
  - A heißt linear-zeitbeschränkt, wenn $s_A \in O(n)$
    (d.h. $\exists c > 0 \exists n_0 \in N_0 \forall n \geq n_0 : s_A(n) \leq c \cdot n$)
  - A heißt polynomial-zeitbeschränkt, wenn $s_A \in O(n^k)$
    (d.h. $\exists c > 0 \exists n_0 \in N_0 \forall n \geq n_0 : s_A(n) \leq c \cdot n^k$)
  - A heißt exponentiell-zeitbeschränkt, wenn $s_A \in O(k^n)$
    (d.h. $\exists c > 0 \exists n_0 \in N_0 \forall n \geq n_0 : s_A(n) \leq c \cdot k^n$)
- Von besonderem Interesse sind die polynomial-zeitbeschränkten Algorithmen, weil in dieser Klasse viele praxisrelevante Probleme liegen und die Laufzeit dieser Algorithmen häufig noch vertretbar ist

Übersicht

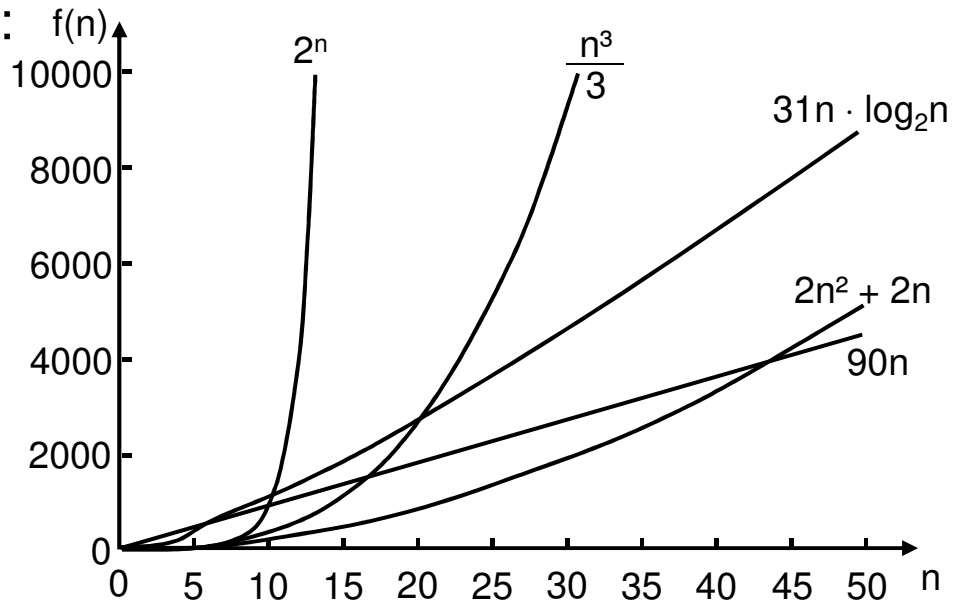
- Komplexitätstheorie
- Beispiele zur Bewertung von Algorithmen
- Nicht-Determinismus, Sprachklassen P, NP

Beispiel

- Die Algorithmen A_1 bis A_5 lösen ein gegebenes Problem:

	A_1	A_2	A_3	A_4	A_5
Benötigte Zeiteinheiten	2^n	$n^3/3$	$31n \cdot \log_2 n$	$2n^2+2n$	$90n$
Komplexität	$O(2^n)$	$O(n^3)$	$O(n \cdot \log n)$	$O(n^2)$	$O(n)$

- Laufzeit der Algorithmen:



- Fazit: Obwohl A_5 die beste Komplexität aufweist, sind für kleine Werte für n andere Algorithmen effizienter

Komplexität der Fibonacci-Berechnung

- Iterative Lösung:
  - Schleife muss n-mal durchlaufen werden $\Rightarrow$ O(n)
- Rekursive Lösung:
  - Maß für Komplexität: Anzahl rekursiver Aufrufe
  - Entsprechen etwa Fibonacci-Zahl fib(n)
  - Damit: Komplexität entspricht fib(n)
  - Keine Beschränkung durch Polynom möglich
  - D.h. für jedes Polynom p existiert $n_0 \in N$, so dass fib(n) > p für $n \geq n_0$
  - Damit: Rekursive Fibonacci-Berechnung in Klasse der exponentiell-zeitbeschränkten Algorithmen
- Anmerkung:
  - Als weiterer Nachweis, dass die rekursive Lösung exponentiell-zeitbeschränkt ist, dient folgende Formel zur Berechnung der Fibonacci-Zahl:

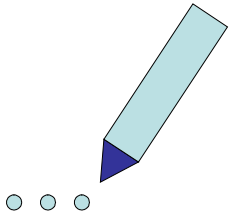
$$\frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

  - Die exponentielle Laufzeit kann aus dem Auftreten von n im Exponenten geschlossen werden

Felder

- Gegeben: Feld mit Eingabegröße n

Problem	Komplexität
Speicherplatzverbrauch	n
Lesen des ersten Elements	1
Finde größtes Element	n
Vergleich „Jeder mit Jedem“	n*n
Wie oft ist Wert x im Feld?	n
Dreifache Summe aller Elmente	
Lösung 1: • Schleife zum Aufaddieren • Anschließend Multiplikation mit 3	n
Lösung 2: • Schleife • In der Schleife Element mit 3 multiplizieren • Summe bilden	n aber langsam
Lösung 3: • Drei Schleifen hintereinander • Aufaddieren	n aber sehr langsam

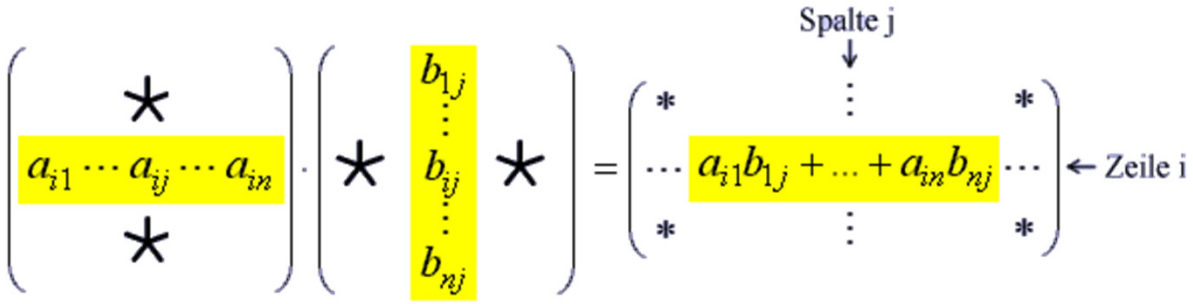


Matrizenmultiplikation (I)

- Zwei Matrizen lassen sich multiplizieren, wenn die Spaltenzahl der ersten der Zeilenzahl der zweiten entspricht
- Definition:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mp} \end{pmatrix}$$

mit $c_{ik} := \sum_{j=1}^{n} a_{ij} \cdot b_{jk}$ für $i = 1, \ldots, m$ und $k = 1, \ldots, p$



[https://homepages.physik.uni-muenchen.de/~milq/spiele/iupages/matrix_m.html]

Matrizenmultiplikation (II)

- Implementierung:

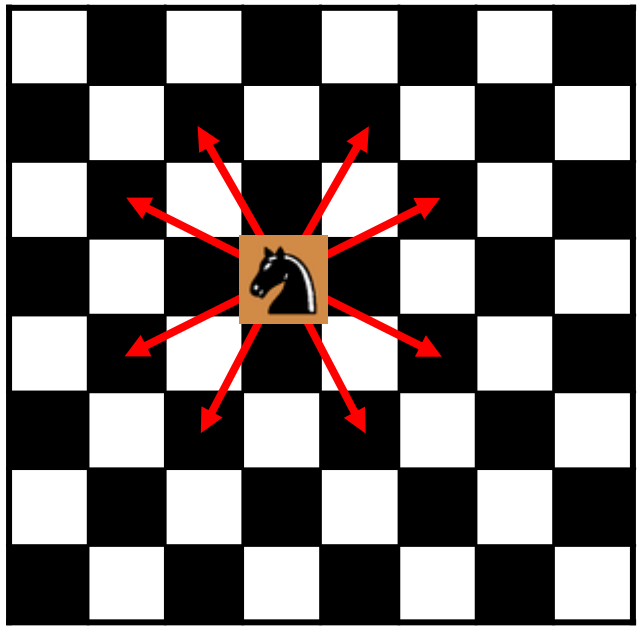
```
( 1) // Methode multiply zum Multiplizieren der Matrizen m1 und m2
( 2) // Rückgabe des Resultats in m3
( 3) public boolean multiply(Matrix m1, Matrix m2) throws
( 4)                MatrixIncompatibleForMultiplicationException{
( 5) int wert = 0;
( 6)   if (m1.spalten != m2.zeilen){
( 7)      throw new MatrixIncompatibleForMultiplicationException
( 8)                              ("Multiplikation nicht möglich");
( 9)   }
(10)   else{
(11)     for (int i=0;i<m1.zeilen;i++){
(12)       for (int j=0;j<m2.spalten;j++){
(13)         wert = 0;
(14)         for (int k=0;k<m1.spalten;k++){
(15)           wert += m1.getMatrix(i,k) * m2.getMatrix(k,j);
(16)         }
(17)         this.setMatrix(i,j,wert);
(18)       }
(19)     }
(20)     return true;
(21)   }
(22)}
```

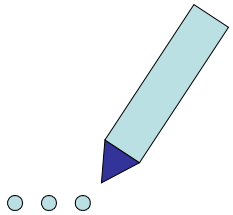
Matrizenmultiplikation (III)

- Komplexität:
  - Eingabegröße: n := max {n,m,p}
  - Laufzeit: $O(n^3)$

Springerproblem

- Springer auf Schachbrett (Größe nxn)
- Springer startet auf beliebigem Feld
- Jedes Feld auf Brett soll genau einmal besucht werden





- Lösungsidee?
- Komplexität?

Übersicht

- Komplexitätstheorie
- Beispiele zur Bewertung von Algorithmen
- Nicht-Determinismus, Sprachklassen P, NP

Nichtdeterminismus

- Algorithmus A heißt nichtdeterministisch, wenn A folgendes Sprachelement enthält: `Anweisung_1 OR Anweisung_2`
- Es kann entweder mit `Anweisung_1` oder `Anweisung_2` fortgesetzt werden, Auswahl erfolgt dabei willkürlich, insbesondere unabhängig vom bisherigen Programmverlauf
- Ein nichtdeterministischer Algorithmus rät dabei stets die richtige Lösung
- Dieses Vorgehen ist natürlich fiktiv
- Mit diesem Mittel kann man beispielsweise den Weg aus einem Labyrinth in kürzester Zeit finden

Entscheidungs-/Optimierungsprobleme

- Entscheidungsproblem:
  - Gesucht ist Lösung ja/nein
  - Beispiele:
    - Ist gegebene Folge sortiert?
    - Ist gegebener Graph zusammenhängend?
    - Terminiert gegebenes Java-Programm?
- Optimierungsproblem:
  - Gesucht ist optimale Lösung (z.B. Maximum oder Minimum einer Funktion)
  - Definierte Randbedingungen legen Menge zulässiger Lösungen fest
  - Lösung des Optimierungsproblems ist zulässige Lösung, die gefordertes Optimierungskriterium bestmöglich erfüllt
  - Beispiele:
    - Färbe Knoten eines Graphen, so dass adjazente Knoten keine gemeinsame Farbe haben und Anzahl der Farben minimal ist
    - Finde in kantenmarkiertem Graphen Weg von Knoten k_1 zu Knoten k_2 mit minimaler Kantensumme

Problemklasse P

- Definition:

  Problemklasse P (P wie polynomial) umfasst alle Probleme, für die Algorithmus mit deterministisch polynomial-zeitbeschränkter Laufzeit existiert.
- Beispiele:
  - Kürzester Weg in Graphen
  - Sortieren einer Folge von Werten
- Beispiele für Probleme außerhalb von P:
  - Knotenfärbung eines Graphen
  - Rundreiseproblem
- Bedeutung der Problemklasse P:
  - In Praxis wichtig, da diese Algorithmen mit vertretbarem Aufwand zu Ergebnis kommen
  - Werden daher auch „Klasse der effizient lösbaren Probleme“ genannt

Problemklasse NP

- Definition:

  Problemklasse NP (NP wie nichtdeterministisch polynomial) umfasst alle Probleme, für die Algorithmus mit nicht-deterministisch polynomial-zeitbeschränkter Laufzeit existiert.

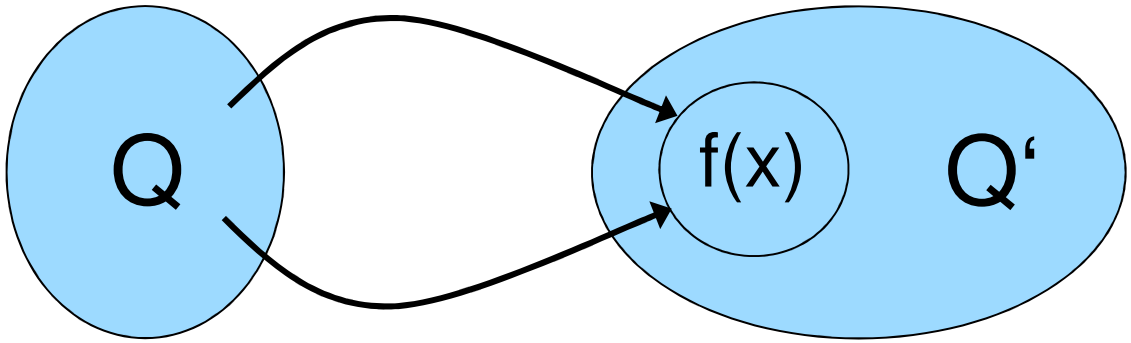
- Beziehung P zu NP:
  - P Teilmenge gleich NP
  - Frage, ob P = NP oder P ≠ NP offen
  - Bekannt als P=NP-Problem
  - DAS herausragende offene Problem der Theoretischen Informatik
  - Vermutung: P ≠ NP

Transformation von Problemen

- Definition:

  Problem Q heißt auf Problem Q‘ transformierbar, wenn eine Funktion f existiert, die Q auf Q‘ abbildet.

  In Zeichen: $Q \leq_f Q'$



- Ist f Polynom, spricht man von polynomialer Transformierbarkeit:

  $Q \leq_p Q'$ mit p Polynom

Problemklasse NP-schwer

- Definition:

  Problemklasse NP-schwer umfasst alle Probleme, auf die sich beliebiges Problem P‘ aus NP in Polynomialzeit transformieren lässt.

  Formal: P heißt NP-schwer g.d.w. $\forall P' \in NP: P' \leq_p P$.

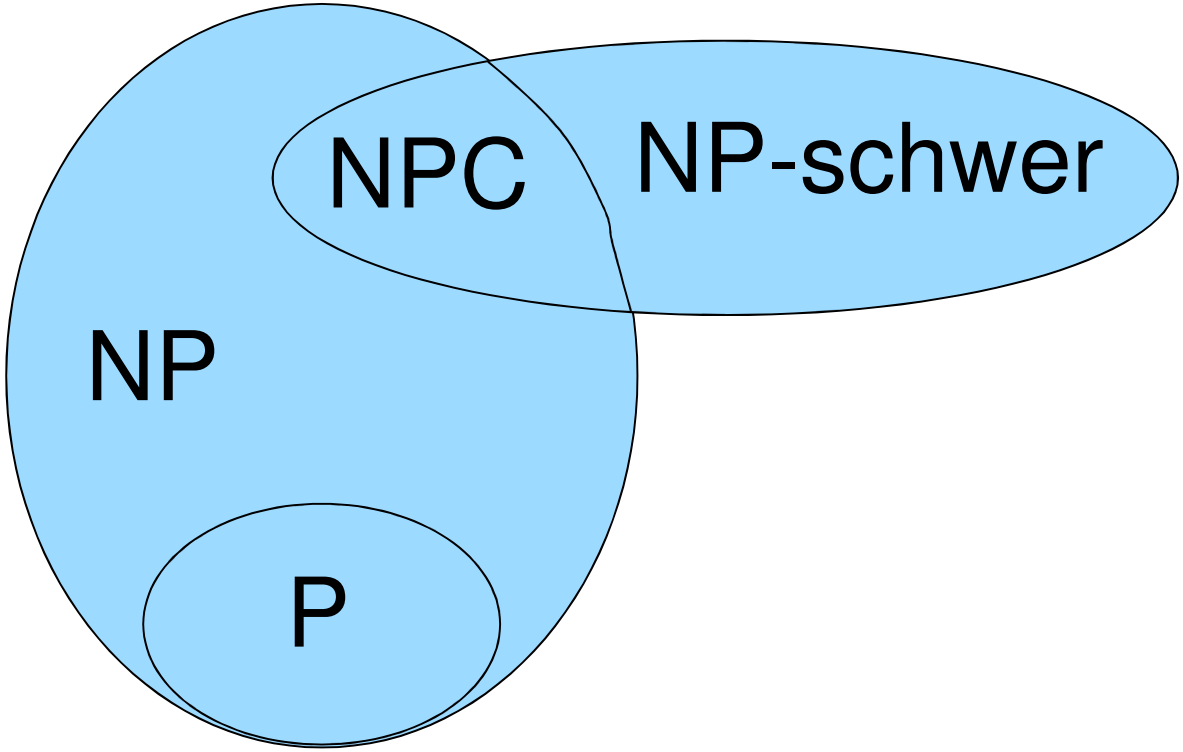
- Anmerkung:
  - In Literatur manchmal Bezeichnung NP-hart

Problemklasse NP-vollständig

- Definition:
  Problemklasse NPC (NP-vollständig, NP complete) umfasst alle Probleme, die selber Element von NP und NP-schwer sind.
  Formal: P heißt NP-vollständig g.d.w.
  (1) $P \in NP$
  und
  (2) $\forall P' \in NP: P' \leq_p P$ mit p Polynom.
- Anschaulich: NPC ist Klasse der schwierigsten Probleme innerhalb von NP
- Anmerkung:
  - NPC- Definition ist „rekursiv“
  - D.h. mindestens ein Problem notwendig, dessen NPC-Zugehörigkeit anders bewiesen wird
  - Erfüllbarkeitsproblem: Für Boolesche Funktion $f(x_1, \ldots, x_n)$ wird gefragt, ob Belegung von $x_1, \ldots, x_n$ existiert, so dass f wahr wird
  - Erfüllbarkeitsproblem ist in NPC (Satz von Cooke 1971)

Zusammenhang P, NP, NP-schwer, NPC

- Unter Annahme P ≠ NP gilt:



NP-vollständige Probleme

- Diverse NP-vollständige, praxisrelevante Probleme bekannt (z.B. viele aus Bereichen Transport und Logistik)
- Beispiele:
  - Rucksackproblem (Knapsack Problem)
  - Anschaulich: „Packe einen Rucksack so voll wie möglich“
  - Formaler: Gegeben sind n Werte a_i, und das Rucksackvolumen v, gesucht ist eine Indexmenge $I \in \{1,...,n\}$ mit $\sum_{i \in I} a_i = c \leq v$ und $v - c$ ist minimal

  - Problem des Handlungsreisenden (Traveling Salesman Problem)
  - Anschaulich: „Finde kürzeste Rundreise“
  - Formaler: Ein Handlungsreisender muss n Orte besuchen und will zum Ausgangsort zurückkehren, zwischen zwei Orten ist die Entfernung bekannt

    (Graph mit den Orten als Knoten, markierte Kanten geben die Entfernungen an)

    Gesucht ist ein geschlossener Weg im Graphen

Zusammenfassung

- Komplexitätstheorie:
  - Methoden der Bewertung
  - Komplexitätsklassen

- Beispiele zur Bewertung von Algorithmen:
  - Fibonacci-Zahlen
  - Felder
  - Matrizen
  - Springerproblem

- Nicht-Determinismus, Sprachklassen P, NP:
  - Nicht-Determinismus (Sprachelement ***OR***)
  - Klassen P und NP, P=NP-Problem
  - NP-Vollständigkeit

Übungen (I)

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
P Z Z W D C C Q P N N P J N T R D C N Q Q B B R C
I O K S A Q P X L T H V T S U H T Q I I T O Q E W
L Q L P M J X Z D G B L J C A O N F C D L M Z G Z
I G F Y A R D F H I V G K C P J W B H N N C M H I
P O J H N G D J K S M S L B K O V H T V I T H W J
K G O W I O H B S B A U W U U W C U D O D N B R U
F N Z S V L M B G C F T B R W C C L E D Z W I E A
L Y Q T G R E I K K R K M E H Z P L T Y T D O U E
F O M A E P Y P A L A A Z Q T F P O E A K C Y F V
W P O D B V R Q A L T I Y A D B B V R J R S P C W
H M R H M O T Q M H Z T Y C D H B N M N L B P M G
S P T Z B I O T M K T E G W T M Z H I X A O F P Z
S R P L V T C X N H A W I Q C I Z P N M O O H N Q
X O E Y U P P Z M A W S C T I V G H I K R S D G G
C M C M K I M G M Y B I D S B Z I F S M W J E N H
X N W H U O F K N K O R Z K R E D D M K M L E S T
S U Y X R V N N B A Y O J D E R S D U N I F K G Q
F Z G W C B B P V K S Q V F C S M C S K E F G D F
V Q Q X J J R P D Q I P Q B O N A E H S Z D V M W
G N O J W N I C S X L X Y R F U A C Q R B G Y B K
V U B T T I A B Z E E X Z S W X A Y T T A B Q F Q
P N Z D O L G O X V A V Q K I Y N H I S M E P Z Q
N O Z E Q O G I D N E A T S L L O V P N R L N T E
E C N Z T M C E X M J C E T I V F B G R Z O F K B
Z D N X H A S D O N P M U K N H W U Q H S X W U T
```

Übungen (II)

- **Aufgabe 2**
  a) Wie lässt sich die Komplexität eines Algorithmus prinzipiell messen?
  b) Was ist die wünschenswerte Methode?
  c) Was ist das Landausche Symbol?
  d) Was sind polynomial zeitbeschränkte Algorithmen?
  e) Was ist Nicht-Determinismus?
  f) Wie sind die Problemklassen P und NP definiert?
  g) Wann ist ein Problem/Algorithmus NP-vollständig?
  h) Gilt P=NP?

Übungen (III)

- **Aufgabe 3**

  Geben Sie zu den folgenden benötigten Zeiteinheiten jeweils den Komplexitätswert O an!

  a) $4n^2 + 7n + 5$
  b) $2^n + n^7$
  c) log(log n)
  d) 42
  e) 2n + n!
  f) $6n^{15} + n^2 - 5$

  Welche dieser Komplexitäten gelten als effizient berechenbar?

- **Aufgabe 4**

  Geben Sie für die folgenden Pseudocode-Algorithmen jeweils die Laufzeitkomplexität in Abhängigkeit von n an!

  a) Gegeben sei ein Feld A der Länge n.

```
Für alle Permutationen von a_1, ...,a_n ∈ A:
  Wenn ∀ i ∈ {1, ..., n-1} : a_i ≤ a_i+1 ⇒
        Feld ist sortiert, beende Algorithmus
```

Übungen (IV)

b)

```
for (i = 0; i<=n; i++){
   for (j = 0; j<=n; j++){
      "Do something"
   }
}
```

c) Gegeben sei ein Feld A der Länge n.

```
Gebe den Wert von a[1] zurück;
```

d) Gegeben sei ein Liste L mit n Elementen.

```
Gebe alle Elemente von L viermal aus.
```

e) Gegeben sei ein Feld A der Länge n.

```
Addiere die Werte in A und gebe das Resultat aus.
```