Theoretische Informatik für Wirtschaftsinformatik und Lehramt

Komplexität von Problemen

Priv.-Doz. Dr. Stefan Milius
stefan.milius@fau.de

Theoretische Informatik
Friedrich-Alexander Universität Erlangen-Nürnberg

SS 2016

Gliederung

Worum geht es in diesem Abschnitt? (I)

- **Komplexität** = Aufwand algorithmischer Berechnungen.
  Man unterscheidet beim Ressourcenbedarf:
  - Anzahl Rechenschritte
  - Speicherbedarf

  Ziel: effiziente Algorithmen
- Aufwand in Abhängigkeit von der Eingabegröße
  O-Notation zur Abschätzung
  (vernachlässigt konstante Faktoren)
- Interessant hier: Unterscheidung in Algorithmen, die
  - polynomiell
  - exponentiell

  viele Schritte in Abhängigkeit der Eingabegröße benötigen.

Worum geht es in diesem Abschnitt? (II)

- Zwei Klassen von Entscheidungsproblemen:

  Klasse $\mathcal{P}$: Probleme, die von DTM in polynomieller Zeit gelöst werden können

  Klasse $\mathcal{NP}$: Probleme die von NTM in polynomieller Zeit gelöst werden können

- Speziell:

  $\mathcal{NP}$-**vollständige** Probleme: „schwerste“ Probleme in $\mathcal{NP}$

  - keine effizienten (= polynomiellen) Lösungsverfahren bekannt
  - löst man ein einziges $\mathcal{NP}$-vollständides Problem effizient, dann hat man alle $\mathcal{NP}$-Probleme effizient gelöst

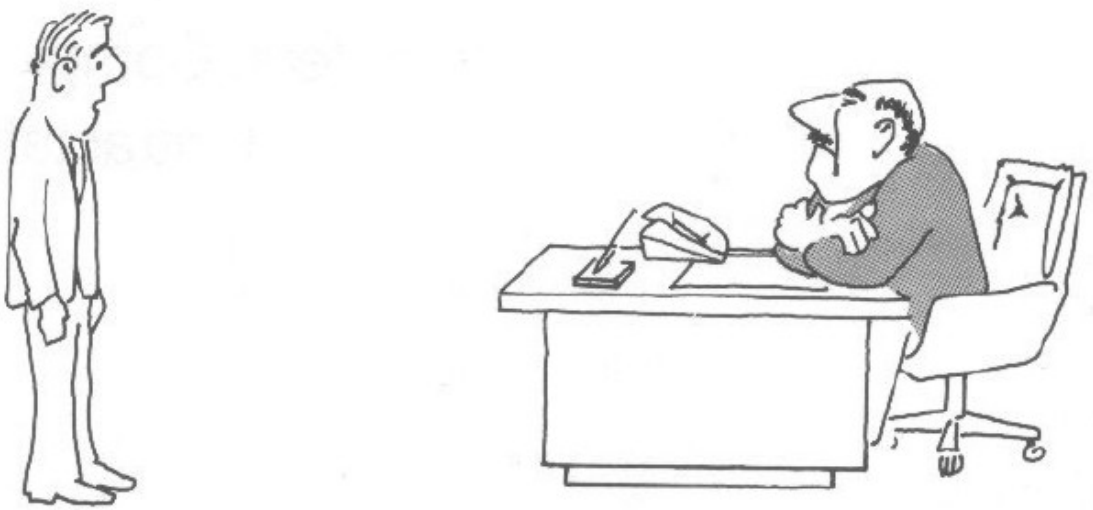
Lernziele

- die O-Notation kennen und sie bei der Abschätzung des Ressourcenbedarfs eines Algorithmus sachgerecht anwenden können
- die Klassen $\mathcal{P}$ und $\mathcal{NP}$ definieren können
- die Konzepte $\mathcal{NP}$-vollständig und $\mathcal{NP}$-hart erklären können
- bekannte $\mathcal{NP}$-vollständige Probleme kennen und erläutern können
- für ein gegebenes Problem nachweisen können, dass es in $\mathcal{P}$ oder in $\mathcal{NP}$ liegt bzw. $\mathcal{NP}$-vollständig ist

Motivation (I)

Ein Comic aus Garey/Johnson: Computers and Intractability:

Nicht so gut ...



„I can't find an efficient algorithm. I guess I'm just too dumb."

Motivation (II)

Besser . . .

„I can't find an effcient algorithm, . . .
. . . but neither can all these famous people . . . "

Motivation (III)

„. . . because no such algorithm is possible.“

Komplexität (I)

- Komplexität (in der Informatik):
  „Kompliziertheit“ von Problemen, Algorithmen oder Daten
- Komplexitätstheorie:
  Analyse des Ressourcenverbrauchs von Algorithmen
- Komplexität eines Algorithmus:
  (maximaler) Ressourcenbedarf dieses Algorithmus
- Ressourcen:
  - **Zeitkomplexität:** Anzahl der benötigten Rechenschritte
  - **Raumkomplexität:** Speicherbedarf
- Von Interesse:
  asymptotisches Wachstum des Ressourcenbedarfs, wenn mehr Daten zu verarbeiten sind
  (z.B. doppelte Datenmenge $\rightsquigarrow$ doppelter oder quadrierter Aufwand)
- weniger von Interesse: Aufwand eines konkreten Programms auf einem bestimmten Computer

Komplexität (II)

- Komplexität eines **Problems:**
  Ressourcenverbrauch eines optimalen Algorithmus
- **Schwierigkeit:** alle Algorithmen für ein Problem müssen betrachtet werden
- Angabe des Ressourcenbedarfs:
  - Angabe in Abhängigkeit von der Länge n der (Kodierung der) Eingabe
  - asymptotische Abschätzung für große n unter Verwendung der „O-Notation“

O-Notation (I)

Definition (O-Notation)

Seien $f, g : \mathbb{N} \to \mathbb{N}$ Funktionen. Dann ist $f(n) \in O(g(n))$,

(„f wächst höchstens so schnell wie g“)

falls gilt: es gibt eine Konstante $c \in \mathbb{N}$ und $n_0 \in \mathbb{N}$, so dass für jedes $n \geq n_0$ gilt:

$$f(n) \leq c \cdot g(n)$$

Anschaulich: ab einem bestimmten Wert n_0 ist f kleiner als g multipliziert mit einem konstanten Faktor c:

- nur das sogenannte „asymptotische Wachstum“ zählt (Verhalten für große Werte von n)
- Konstanten werden vernachlässigt.

O-Notation (II)

Schreibweisen:

- Die Funktion f und g durch ihren definierenden Ausdruck:

$$\log n, n, n \cdot \log n, n^2, n^3, 2^n, n^n, \ldots.$$

  Im allgemeinen n als Funktionsargument.

- $O(g)$ bezeichnet die Menge aller Funktionen, die höchstens so schnell wachsen wie g.

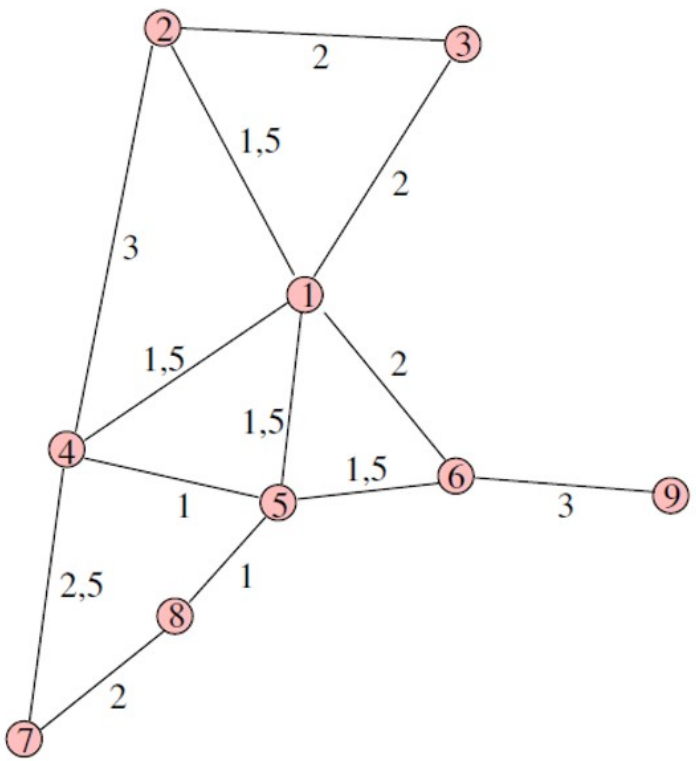
  Daher: $f \in O(g)$ (oder sogar „$f = O(g)$“).

- vgl. LV „Algorithmen und Datenstrukturen“ sowie Anhang dieser LE

Zeitkomplexitätsklassen

Das Travelling Salesperson Problem (TSP) (I)

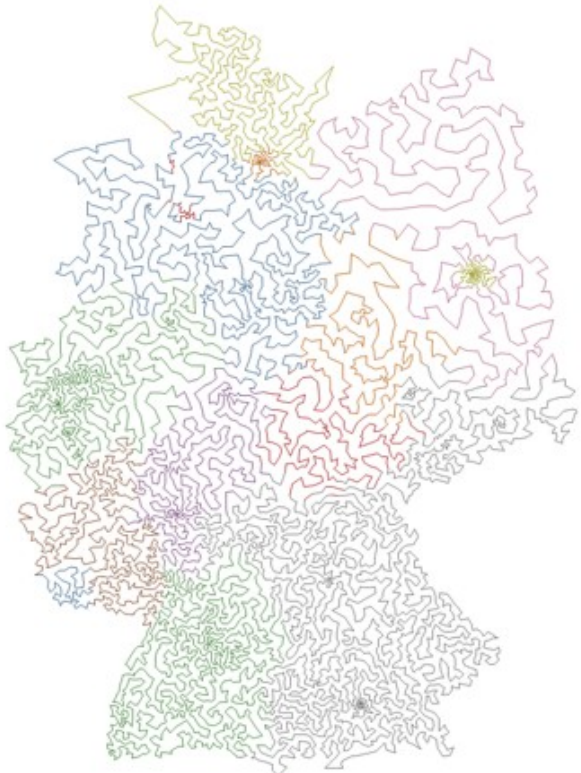
Ist der kürzeste Weg, der alle Knoten besucht und wieder zum Ausgangsort zurückkehrt, kürzer als eine gegebene Distanz d?



Ist es wirklich nötig, alle Touren auszuprobieren oder gibt es ein effizienteres Verfahren?

Das Travelling Salesperson Problem (TSP) (II)

Beispiel für eine Tour durch alle Postleitzahlengebiete Deutschlands (berechnet mittels Lin-Kernighan-Heuristik)



Länge der Tour: 51799 km

Das Tourenplanungsproblem (I)

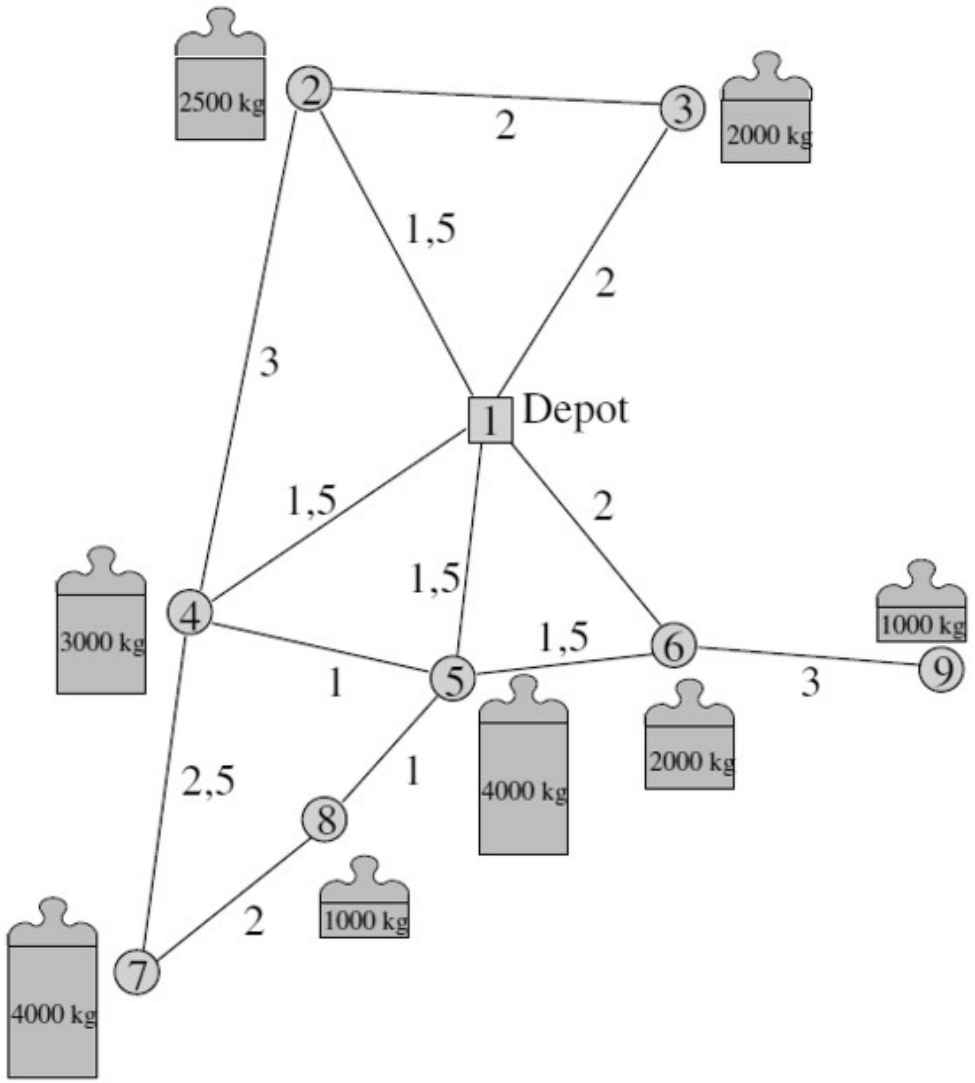
Eingabe:

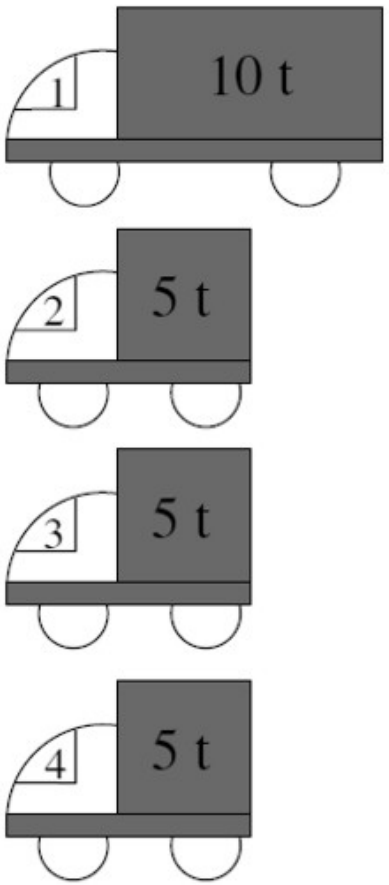
- ein Graph (= Wegenetz) wie beim TSP
- für jeden Knoten ein Gewicht (= abzuholende Ladung) (außer speziellem Depot-Knoten)
- eine Menge von Lastwagen mit Ladebeschränkung
- eine Schranke d

Aufgabe:

- Kann man den Lastwagen so Touren zuordnen, dass die Ladebeschränkungen eingehalten werden und die Gesamtstrecke kleiner als d bleibt?

Das Tourenplanungsproblem (II)





Laufzeit von Algorithmen und O-Notation

- O-Notation zur Angabe der Laufzeit von Algorithmen
- Laufzeit = Anzahl der Schritte, die der Algorithmus ausführt

Beispiele

- Sortieralgorithmen mit Laufzeit:

$$\begin{aligned} O(n \cdot \log n) &\quad \text{(z.B. Mergesort)} \\ O(n^2) &\quad \text{(z.B. Bubblesort).} \end{aligned}$$

- Jedes bekannte Verfahren für das Travelling Salesperson Problem hat exponentielle Laufzeit:

$$O(k^n) \quad \text{für Konstante } k.$$

Dabei bezeichnet der Parameter n immer die Größe der Eingabe.

Zeitkomplexitätsklassen (I)

Definition der Klasse aller Sprachen, die von einer deterministischen Turingmaschinen mit Zeitbeschränkung akzeptiert werden.

Definition (Zeitbeschränkte DTM und akzeptierte Sprachen)

Sei $f : \mathbb{N} \to \mathbb{N}$ eine (totale) Funktion. Die Klasse $\mathrm{TIME}(f(n))$ besteht aus allen Sprachen L, für die es eine deterministische Mehrband-Turingmaschine M gibt mit:

- $L = L(M)$ und
- $\mathrm{time}_M(w) \leq f(|w|)$ für alle Wörter $w \in \Sigma^*$.

Hierbei gilt: $\mathrm{time}_M(w) =$ Anzahl Rechenschritte von M bei Eingabe w.

Also:

- Anzahl der Rechenschritte der Turingmaschine ist beschränkt und
- die Beschränkung ist abhängig von der Länge der Eingabe.

Zeitkomplexitätsklassen (II)

Zur Erinnerung:

- Es seien K_i Konfigurationen und

$$K_0 \vdash K_1 \vdash K_2 \vdash \cdots \vdash K_\ell$$

  eine Rechnung einer TM M.
- Dann ist ℓ die Länge der Rechnung von M.
  (d. h. ℓ = Anzahl der Übergänge von einer Konfiguration zur nächsten)

Einband- vs. Mehrband-Turingmaschine (I)

Warum **Mehrband**-TM?

- Mehrband-Turingmaschinen $\rightsquigarrow$ realistischere Komplexitätsfunktionen f
- Einband-Turingmaschine:
  - viel Rechenzeit für das „Hin-und-herlaufen“ auf dem Band wegen sequenzieller Speicherung
  - diese Rechenzeitanteile liegen nur am eingeschränktem Berechnungsmodell
    (nicht am Problem)
- Mehrband-Turingmaschine durch Einband-Turingmaschine simulierbar: . . .

Einband- vs. Mehrband-Turingmaschine (II)

- **Es gilt:**
  Eine $f(n)$ rechenzeitbeschränkte Mehrband-Turingmaschine kann durch eine $O(f^2(n))$ rechenzeitbeschränkte Einband-Turingmaschine simuliert werden.
- Ist f ein Polynom, dann liefert die Operation des Quadrierens wieder ein Polynom.
- **Also:** Unterscheidung Einband- oder Mehrband-TM ist unerheblich.

Kodierung von Eingabedaten

Wichtig: vernünftige Kodierung aller Eingabedaten!

Vernünftig heißt:

- Zahlen in Binär- oder Dezimalcode
- Graphen/Bäume als Adjazenzmatrizen oder Adjazenzlisten
- etc.

Unäre Kodierung von n als $|^n$ ist „unvernünftig“, denn:

- $| |^n | = n$
- $|\text{Binärcode von } n| \in O(\log n)$

Dann ist:

- Wert von n (= Länge der unären Kodierung) ist exponentiell in der Länge der Binärkodierung
- Algorithmen mit unärer Kodierung brauchen exponentiell mehr Zeit (nur zum Lesen der Eingabe!)

Die Klasse $\mathcal{P}$ (I)

Zunächst Wiederholung Polynom (in einer Variablen).

Definition (Polynom)

Ein Polynom vom Grad $k \in \mathbb{N}$ ist eine Funktion $p : \mathbb{N} \to \mathbb{N}$ mit $k, a_k, ..., a_0 \in \mathbb{N}$ und der Abbildungsvorschrift

$$\begin{aligned} p_k(n) &= a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \\ &= \sum_{i=0}^{k} a_i n^i \end{aligned}$$

Satz 10.1

Sei $p(n)$ ein Polynom vom Grad k. Es gilt: $p(n) \in O(n^k)$.

Die Klasse $\mathcal{P}$ (II)

Beweis: Wir schätzen $p(n)$ ab:

$$\begin{aligned} p(n) &= a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \\ &\leq an^k + an^{k-1} + \cdots + an + a \\ &\quad \text{mit } a = max\{a_i \mid 0 \leq i \leq k\} \\ &\leq \underbrace{an^k + an^k + \cdots + an^k + an^k}_{k+1 \text{ -mal}} \qquad \text{für alle } n \geq 1 \\ &= a(k+1)n^k \qquad \text{für alle } n \geq 1 \end{aligned}$$

Setze $c = a(k+1)$. Dann gilt für alle $n \geq 1$:

$$p_k(n) \leq c \cdot n^k.$$

Also: $p(n) \in O(n^k)$. $\square$

Die Klasse $\mathcal{P}$ (III)

Klasse aller Sprachen, die von deterministischen Turingmaschinen mit polynomieller Laufzeitbeschränkung akzeptiert werden.

Definition (Komplexitätsklasse $\mathcal{P}$)

$$\begin{aligned}\mathcal{P} &= \{L \mid \text{es gibt eine DTM } M \text{ und ein Polynom } p \text{ mit } L(M) = L \quad \text{und} \quad \text{time}_M(x) \leq p(|x|)\} \\ &= \bigcup_{p \text{ Polynom}} \text{TIME}(p(n))\end{aligned}$$

Informell:

P enthält alle Probleme, für die effiziente Algorithmen existieren (effizient = polynomielle Laufzeit).

Klasse $\mathcal{P}$ mit O-Notation

Formulierung mit Hilfe der O-Notation:

Eine Problem liegt in $\mathcal{P}$ genau dann, wenn es von einer deterministischen TM entschieden wird,
die polynomiell viele Schritte macht:

d.h. es existiert ein $k \in \mathbb{N}$ so dass die TM für jedes Eingabewort der Länge n nach

$$O(n^k) \quad \text{Schritten hält.}$$

Laufzeit von Algorithmen (I)

Beispiel: DTM M_1 zur Spiegelungsfunktion

$$z_0 w \vdash^* z w^R \qquad (z w^R \text{ final})$$

	a	b	\$	$\square$
z_0	(z_0, a, R)	(z_0, b, R)	–	$(z_1, \square, \mathsf{L})$
z_1	(z_2, a, L)	(z_2, b, L)	–	–
z_2	$(z_a, \$, \mathsf{R})$	$(z_b, \$, \mathsf{R})$	–	$(z_5, \square, \mathsf{R})$
z_a	(z_a, a, R)	(z_a, b, R)	$(z_a, \$, \mathsf{R})$	(z_3, a, N)
z_b	(z_b, a, R)	(z_b, b, R)	$(z_b, \$, \mathsf{R})$	(z_3, b, N)
z_3	(z_3, a, L)	(z_3, b, L)	$(z_4, \$, \mathsf{L})$	–
z_4	(z_2, a, N)	(z_2, b, N)	$(z_4, \$, \mathsf{L})$	$(z_5, \square, \mathsf{R})$
z_5	–	–	$(z_5, \square, \mathsf{R})$	–

Es gilt: $\text{time}_{M_1}(n) \in O(n^2)$.

Laufzeit von Algorithmen (II)

Beispiel: DTM M_1 zur Spiegelungsfunktion (Fortsetzung)

Begründung:

- Vorbereitung: letztes Zeichen suchen ($n+1$ (hier: 5) Schritte)
  $\square z_0 abab\square \vdash^* \square abaz_1 b\square$
- letztes Zeichen ist erstes Zeichen des Ergebnisses;
  vorletztes Zeichen suchen: (1 Schritt)
  $\square abaz_1 b\square \vdash \square abz_2 ab\square$
- vorletztes Zeichen verarbeiten;
  Zeichen durch \$-Zeichen ersetzen: (1 Schritt)
  $\square abz_2 ab\square \vdash \square ab\$z_a b\square$
  Zeichen transportieren und abliefern: (2 Schritte)
  $\square ab\$z_a b\square \vdash^* \square ab\$bz_3 a\square$

Laufzeit von Algorithmen (III)

Beispiel: DTM M_1 zur Spiegelungsfunktion (Fortsetzung)

- nächstes zu transportierendes Zeichen suchen:
  zum Zeichen vor dem letzten \$-Zeichen
  $\square ab\$bz_3a\square \vdash_* \square az_4b\$ba\square$ (3 Schritte)
  von dort zum nächsten zu transportierenden Zeichen
  $\square az_4b\$ba\square \vdash \square az_2b\$ba\square$ (1 Schritt)
- nächstes Zeichen verarbeiten:
  Zeichen durch \$-Zeichen ersetzen
  $\square az_2b\$ba\square \vdash \square a\$z_b\$ba\square$ (1 Schritt)
  Zeichen transportieren und abliefern
  $\square a\$z_b\$ba\square \vdash_* \square a\$\$baz_3b\square$ (4 Schritte)
- nächstes zu transportierendes Zeichen suchen:
  zum Zeichen vor dem letzten \$-Zeichen
  $\square a\$\$baz_3b\square \vdash_* \square az_4\$\$bab\square$ (4 Schritte)

Laufzeit von Algorithmen (IV)

Beispiel: DTM M_1 zur Spiegelungsfunktion (Fortsetzung)

- nächstes zu transportierendes Zeichen suchen (Fortsetzung):
  von dort zum nächsten zu transportierenden Zeichen
  $\Box az_4\$\$bab\Box \vdash_* \Box z_2 a\$\$bab\Box$ (2 Schritte)
- nächstes Zeichen verarbeiten:
  Zeichen durch \$-Zeichen ersetzen
  $\Box z_2 a\$\$bab\Box \vdash \Box\$z_a\$\$bab\Box$ (1 Schritt)
  Zeichen transportieren und abliefern
  $\Box\$z_a\$\$bab\Box \vdash_* \Box\$\$\$babz_3 a\Box$ (6 Schritte)
- nächstes zu transportierendes Zeichen suchen:
  zum Zeichen vor dem letzten \$-Zeichen
  $\Box\$\$\$babz_3 a\Box \vdash_* \Box\$z_4\$\$baba\Box$ (5 Schritte)
  von dort zum nächsten zu transportierenden Zeichen
  $\Box\$z_4\$\$baba\Box \vdash_* \Box z_5\$\$\$baba\Box$ (3 Schritte)

Laufzeit von Algorithmen (V)

Beispiel: DTM M_1 zur Spiegelungsfunktion (Fortsetzung)

- Nachbereitung: nichts mehr zu transportieren, \$-Zeichen löschen
  $\square z_5 \$\$\$baba\square \vdash_* \square z_5 baba\square$ (4 Schritte)

Laufzeit von Algorithmen (VI)

Beispiel: DTM M_1 zur Spiegelungsfunktion (Fortsetzung)

Bilanz:

- Vorbereitung: $n+1$ Schritte, also $O(n)$
- letztes Zeichen verarbeiten: 1 Schritt, also $O(1)$
- i-tes Zeichen ($i \geq 2$, von hinten gesehen) verarbeiten
  - 1 Schritt, um Zeichen durch \$-Zeichen zu ersetzen
  - $2(i-1)$ Schritte, um Zeichen zu transportieren
  - $(i+1)+(i-1)$ Schritte, um nächstes zu transportierendes Zeichen zu suchen

Summe: $1+(2i-2)+(i+1)+(i-1) = 4i-1$ Schritte;

Summiere über $i = 2, \ldots, n$:

$$\sum_{i=2}^{n}(4i-1) \leq 4 \cdot \sum_{i=1}^{n} i = 4 \cdot \frac{n \cdot (n+1)}{2} \in O(n^2).$$

Insgesamt: $O(n+1+n^2) = O(n^2)$.

Laufzeit von Algorithmen (VII)

Beispiel: NTM M_2 zur Teilwortüberprüfung

Die nichtdeterministische TM M_2 mit akzeptierendem Zustand z_+ und der Zustandstafel

	a	b	$\square$
z_0	(z_0, a, R) (z_1, a, R)	(z_0, b, R)	–
z_1	–	(z_2, b, R)	–
z_2	(z_+, a, N)	–	–
z_+	–	–	–

entscheidet die Sprache $\{w \in \{a, b\}^* \mid aba \text{ ist Teilwort von } w\}$.

Es gilt: $\text{time}_{M_2}(n) \in O(n)$.

Die Klasse $\mathcal{NP}$ (I)

Zeitbeschränkte nichtdeterministische Turingmaschinen und zugehörige Sprachklassen.

Definition (Zeitbeschränkte NTM und akzeptierte Sprachen)

Sei $f : \mathbb{N} \to \mathbb{N}$ eine (totale) Funktion. Die Klasse $\mathrm{NTIME}(f(n))$ besteht aus allen Sprachen L, für die es eine nichtdeterministische Mehrband-Turingmaschine M gibt mit:

- $L = L(M)$ und
- $\mathrm{ntime}_M(w) \leq f(|w|)$ für alle Wörter $w \in \Sigma^*$.

Hierbei gilt:

$$\mathrm{ntime}_M(w) = \begin{cases} \min\{\text{Länge akzeptierender Rechnungen von } M \text{ bei Eingabe } w\} & w \in L(M) \\ 0 & w \notin L(M) \end{cases}$$

Die Klasse $\mathcal{NP}$ (II)

- **Anschauung:** $\mathcal{NP}$ = Klasse von Problemen, die von nicht-deterministischen Turingmaschinen mit polynomieller Laufzeitbeschränkung akzeptiert werden können

Bemerkung

Es sei M NTM mit $\quad \text{ntime}_M(w) \leq f(|w|) \quad$ für alle $w \in \Sigma^*$.

Dann ist $L(M)$ entscheidbar.

Grund: Man kann einen Schrittzähler $1, 2, \ldots, f(|w|)$ mitlaufen lassen und nach $f(|w|)$ Schritten anhalten.
(da nach höchstens $f(|w|)$ Schritten akzeptiert wird)

Die Klasse $\mathcal{NP}$ (III)

Definition (Komplexitätsklasse *NP*)

$$\mathcal{NP} = \bigcup_{p \text{ Polynom}} \text{NTIME}(p(n))$$

Es gilt offensichtlich:

$$\mathcal{P} \subseteq \mathcal{NP}$$

Offenes Problem: ist $\mathcal{P}$ echt in $\mathcal{NP}$ enthalten?

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

Die Klasse $\mathcal{NP}$ (IV)

- Das **P-NP-Problem** $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ informell:
  - Lässt sich jedes Problem, das sich nichtdeterministisch in polynomieller Zeit lösen lässt, auch deterministisch in polynomieller Zeit lösen?
  - **Allgemeine Erwartung:** dies ist nicht möglich.
- Andere Beschreibung von $\mathcal{NP}$:
  Alle Probleme für die eine vorgelegte (= „nichtdeterm. geratene“) Lösung in polynomieller Zeit auf Korrektheit überprüft werden kann.
- **P-NP-Problem** $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ ist dann:
  „Gibt es Probleme, für die es schwieriger ist, eine Lösung zu bestimmen, als zu überprüfen, ob eine gegebene Lösung korrekt ist?“
- **P-NP-Problem** ist eines der 10 Millenium-Probleme:
  1.000.000 US$ Preisgeld für die Lösung
  siehe: `http://www.claymath.org/millennium/P_vs_NP/`

Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$

Zusammenfassung:

Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$

Die Komplexitätsklassen $\mathcal{P}$ und NP sind wie folgt definiert:

$$\begin{aligned}\mathcal{P} &= \{L \mid \text{es gibt eine DTM } M, \text{ die } L \text{ entscheidet,} \\ &\qquad \text{und } k \in \mathbb{N} \text{ mit } \text{time}_M(n) \in O(n^k)\} \\ \mathcal{NP} &= \{L \mid \text{es gibt eine NTM } M, \text{ die } L \text{ entscheidet,} \\ &\qquad \text{und } k \in \mathbb{N} \text{ mit } \text{ntime}_M(n) \in O(n^k)\}\end{aligned}$$

$\mathcal{NP}$-Vollständigkeit

Polynomielle Reduzierbarkeit (I)

- Konzept der Reduzierbarkeit
  (analog zur Berechenbarkeitstheorie)
- „A auf B polynomiell reduzierbar“ ($A \leq_p B$) impliziert:
  Wenn B effizient lösbar ist, dann ist auch A effizient lösbar und man benötigt für B nur einen polynomiell größeren Zeitaufwand als für A.
- Ermöglicht $\mathcal{NP}$-vollständige Probleme:
  Die „schwersten“ Probleme der Komplexitätsklasse $\mathcal{NP}$.
- Das sind Probleme in $\mathcal{NP}$, auf die alle anderen $\mathcal{NP}$-Probleme polynomiell reduzierbar sind.

Polynomielle Reduzierbarkeit (II)

Definition (Polynomielle Reduzierbarkeit)

Seien Σ ein Alphabet und $L_1, L_2 \subseteq \Sigma^*$ zwei Sprachen.

L_1 heißt **polynomiell reduzierbar auf** L_2 (geschrieben: $L_1 \leq_p L_2$), wenn es eine totale berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt, so dass gilt:

a) $w \in L_1 \iff f(w) \in L_2$ für alle $w \in \Sigma^*$.

b) Es gibt eine DTM M, die f berechnet, und $k \in \mathbb{N}$ mit $\text{time}_M(n) \in O(n^k)$.

Polynomielle Reduzierbarkeit (III)

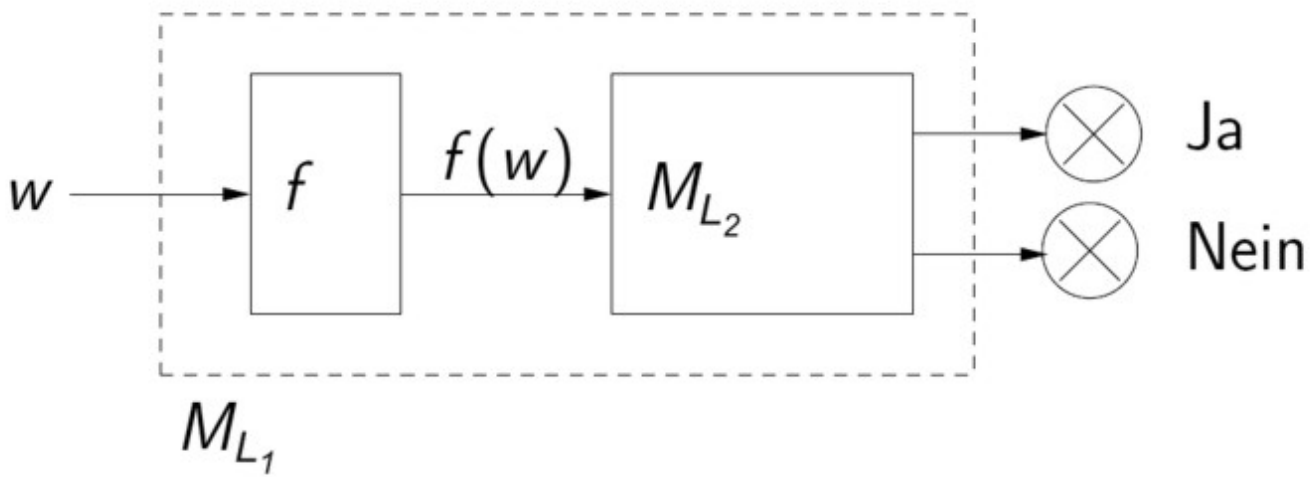
Satz 10.2

Seien L_1, L_2 Sprachen mit $L_1 \leq_p L_2$. Dann gilt:

a) Ist $L_2 \in \mathcal{P}$, so ist $L_1 \in \mathcal{P}$.

b) Ist $L_2 \in \mathcal{NP}$, so ist $L_1 \in \mathcal{NP}$.

Beweisskizze: $L_1 \leq_p L_2$ via $f : \Sigma^* \to \Sigma^*$. Laufzeitanalyse von:



M_{L_2} hat polynomielle Laufzeit $\implies$ M_{L_1} hat polynomielle Laufzeit

$\mathcal{NP}$-hart und $\mathcal{NP}$-vollständig (I)

Definition ($\mathcal{NP}$-hart)

Eine Sprache L heißt $\mathcal{NP}$-**hart**, wenn $L' \leq_p L$ für alle $L' \in \mathcal{NP}$ gilt.

Definition ($\mathcal{NP}$-vollständig)

Eine Sprache L heißt $\mathcal{NP}$-**vollständig**, wenn gilt:

- $L \in \mathcal{NP}$ und
- L ist $\mathcal{NP}$-hart.

Das bedeutet: eine $\mathcal{NP}$-vollständige Sprache ist mindestens so schwierig lösbar wie jedes andere Problem in $\mathcal{NP}$.

$\mathcal{NP}$-hart und $\mathcal{NP}$-vollständig (II)

Bemerkung

Die Relation $\leq_p$ ist transitiv:

$$L_1 \leq_p L_2 \quad \text{und} \quad L_2 \leq_p L_3 \quad \Longrightarrow \quad L_1 \leq_p L_3.$$

Liefert Methode, um aus $\mathcal{NP}$-Härte von L_1 die $\mathcal{NP}$-Härte von L_2 zu folgern:

- Beweise: $L_1 \leq_p L_2$
- Dann gilt für alle $L \in \mathcal{NP}$: $L \leq_p L_2$ (da $L \leq_p L_1 \leq_p L_2$)

$\mathcal{NP}$-hart und $\mathcal{NP}$-vollständig (III)

Satz 10.3

Sei L' $\mathcal{NP}$-vollständig. Dann gilt:

$$\text{Ist } L \in \mathcal{NP} \text{ und } L' \leq_p L, \text{ so ist } L \; \mathcal{NP}\text{-vollständig.}$$

Denn: L' $\mathcal{NP}$-hart und $L' \leq_p L \implies L$ $\mathcal{NP}$-hart

Also: $L \in \mathcal{NP}$ und L $\mathcal{NP}$-hart.

$\mathcal{NP}$-Vollständigkeitsbeweise

Reduktionsmethode

um $\mathcal{NP}$-Vollständigkeit einer Sprache L zu beweisen:

1. Zeige $L \in \mathcal{NP}$; z.B. durch:

   angeben einer NTM M, die L entscheidet und polynom. Laufzeit hat

   Todo: begründen, warum M L akzeptiert; Laufzeitanalyse von M

2. Wähle ein schon als $\mathcal{NP}$-hart bekanntes L' und zeige $L' \leq_p L$:

   Todo: $f : \Sigma^* \to \Sigma^*$ definieren; DTM M_f angeben; begründen warum M_f f berechnet (und immer hält); Laufzeitanalyse von M_f

Bedeutung der $\mathcal{NP}$-Vollständigkeit

Bedeutung der $\mathcal{NP}$-vollständigen Probleme:

- Klassifizierung von Problemen:
  viele Praxis-relevante Probleme sind $\mathcal{NP}$-vollständig

  Exakte Lösungsalgorithmen für diese Probleme haben exponentielle Laufzeit.
  Man muss Algorithmen einsetzen, die „gute“ Lösungen finden. (z.B. Approximationsalgorithmen, Heuristiken).
- Könnte man für eine $\mathcal{NP}$-vollständige Sprache zeigen, dass sie in $\mathcal{P}$ liegt, so wäre $\mathcal{P} = \mathcal{NP}$.
  ($\implies$ alle $\mathcal{NP}$-Probleme haben Algorithmen mit poly. Laufzeit)

Satz 10.4

Sei L $\mathcal{NP}$-vollständig. Dann gilt: $L \in \mathcal{P} \iff P = \mathcal{NP}$.

$\mathcal{NP}$-vollständige Probleme

Ziel jetzt:

- ein erstes $\mathcal{NP}$-vollständiges Problem haben
  (Erfüllbarkeitsproblem für Aussagenlogische Formeln)
- weitere $\mathcal{NP}$-vollständige Probleme durch Reduktion erhalten

Grundbegriffe der Aussagenlogik (I)

Sei $V = \{x_1, x_2, x_3, \dots\}$ abzählbare Menge von **Aussagenvariablen**.

Induktive Definition der Menge $\mathcal{F}$ der **Formeln** (der Aussagenlogik):

1. $x_i \in V \quad \Longrightarrow \quad x_i \in \mathcal{F}$.
2. $F_1, F_2 \in \mathcal{F} \quad \Longrightarrow \quad \neg F_1, (F_1 \wedge F_2), (F_1 \vee F_2) \in \mathcal{F}$.

Eine **Belegung** ist eine Abbildung $\varphi : V \to \{0, 1\}$.
Jede Belegung φ wird fortgesetzt auf $\mathcal{F}$ wie folgt:

$$\begin{aligned} \varphi(\neg F_1) &= 1 - \varphi(F_1) \\ \varphi(F_1 \wedge F_2) &= \min(\varphi(F_1), \varphi(F_2)) \\ \varphi(F_1 \vee F_2) &= \max(\varphi(F_1), \varphi(F_2)) \end{aligned}$$

Eine Formel $F \in \mathcal{F}$ heißt **erfüllbar**, wenn es eine Belegung φ gibt mit $\varphi(F) = 1$.

Grundbegriffe der Aussagenlogik (II)

Beispiel

Sei $F = \neg x_1 \wedge (x_2 \vee x_1)$.

Wertetabelle von F:

x_1	x_2	$x_2 \vee x_1$	$F = \neg x_1 \wedge (x_2 \vee x_1)$
0	1	1	1
0	0	0	0
1	1	1	0
1	0	1	0

Es gibt also eine Belegung $\varphi(F) = 1$. Die Formel ist erfüllbar.

Erfüllbarkeitsproblem der Aussagenlogik (SAT) (I)

SAT-Problem:
Eingabe: eine Formel F der Aussagenlogik
Aufgabe: Entscheiden, ob F erfüllbar ist.
(Frage: gibt es eine Belegung der Variablen mit $0, 1$, so dass F den Wert 1 erhält?)

Kodierung von Eingabeinstanzen:

$$\text{Variablen } x_i \in V \text{ durch Binärcode von } i$$
$$\rightsquigarrow \text{ Formeln } F \text{ sind Wörter über } \{0, 1, \neg, \wedge, \vee, (,)\}.$$

SAT als **formale Sprache**:

$$SAT = \{w \in \{0, 1, \neg, \wedge, \vee, (,)\}^* \mid w \text{ ist Kodierung einer erfüllbaren Formel der Aussagenlogik}\}.$$

Erfüllbarkeitsproblem der Aussagenlogik (SAT) (II)

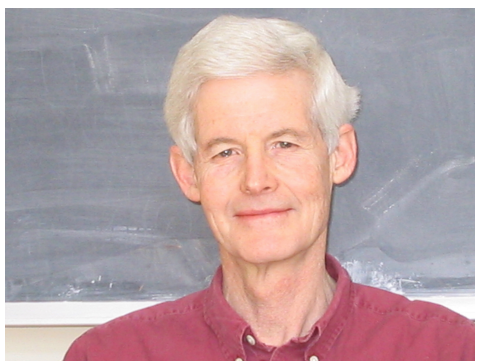
Satz 10.5 (Satz von Cook)

SAT ist $\mathcal{NP}$-vollständig.

Stephen Cook

Amerikanisch-Kanadischer Informatiker und Mathematiker

- einer der Väter der Komplexitätstheorie
- wichtige Beiträge auch zu Beweiskomplexität
- Komplexitätsklasse $\mathcal{SC}$ ($\mathcal{S}$teve's $\mathcal{C}$lass)



Stephen Cook (1939–)
Quelle: University of Toronto

Erfüllbarkeitsproblem der Aussagenlogik (SAT) (II)

Satz 10.5 (Satz von Cook)

SAT ist $\mathcal{NP}$-vollständig.

Beweis: zu zeigen ist

1. $SAT \in \mathcal{NP}$
2. Für alle $L \in \mathcal{NP}$ ist $L \leq_p SAT$.

Wir zeigen hier nur Teil 1: NTM M für erfüllbare Formeln:

1. Eingabe durchlesen und alle in F vorkommenden Variablen feststellen (**Annahme:** dies seien $x_1, \ldots, x_k$)
2. M „rät“ nichtdeterministisch k Werte $a_1, \ldots, a_k$ in $\{0, 1\}$ für die Variablen;

   diese werden in F eingesetzt

   (2^k mögliche verschiedene Rechnungen – für jede Belegung eine)

Erfüllbarkeitsproblem der Aussagenlogik (SAT) (III)

Beweis (Fortsetzung):

3. deterministisch den Wert von F berechnen
4. in akzeptierenden Endzustand genau dann übergehen, wenn der Wert gleich 1 ist

Nun gilt:

- $F \in SAT$ genau dann, wenn es eine nichtdeterministische Rechnung von M gibt, die F akzeptiert
- da außerdem $k \leq |F|$ ist, ist die (nichtdeterministische!) Rechenzeit von M polynomiell.

Also: $SAT \in \mathcal{NP}$.□

Bemerkung zu Teil 2. des Beweises ($L \leq_p SAT$ für alle $L \in \mathcal{NP}$)

für gegebenes L wird eine TM M_L und ihre Arbeitsweise in Aussagenlogische Formel übersetzt

Erfüllbarkeitsproblem der Aussagenlogik (3-SAT)

Definition (Konjunktive Normalform)

Eine Formel $F \in \mathcal{F}$ heißt in **konjunktiver Normalform (KNF)**, wenn

$$F \equiv F_1 \wedge \cdots \wedge F_n \quad \text{und} \quad F_i \equiv F_{i1} \vee \cdots \vee F_{im_i} \quad (i = 1, \ldots, n)$$

gilt mit $(n, m_i \geq 1)$, wobei $F_{ij} \equiv v$ oder $F_{ij} \equiv \neg v$ für ein $v \in V$ ist.

v und $\neg v$ heißen **Literale**.

Problem 3-SAT:
Eingabe: Formel F in KNF mit höchstens drei Literalen pro Klausel F_i
Aufgabe: entscheiden, ob F erfüllbar oder nicht.

Beispiel

$$F = (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee x_3) \wedge \neg x_1 \wedge \neg x_3$$

ist in der geforderten Form **aber** hat keine erfüllende Belegung.
Also $F \notin$ 3-SAT.

Erfüllbarkeitsproblem der Aussagenlogik (k-SAT)

Problem k-SAT:
Eingabe: Formel F in KNF mit höchstens k Literalen pro Klausel
Aufgabe: entscheiden, ob F erfüllbar ist oder nicht.
Formale Sprache:

$$k\text{-SAT} = \{w \in \{0, 1, \neg, \wedge, \vee, (,)\}^* \mid w \text{ ist Kodierung einer erfüllbaren Formel der Aussagenlogik in KNF mit höchstens } k \text{ Literalen pro Klausel}\}$$

Satz 10.6

2-SAT liegt in $\mathcal{P}$.

k-SAT ist $\mathcal{NP}$-vollständig für alle $k \geq 3$.

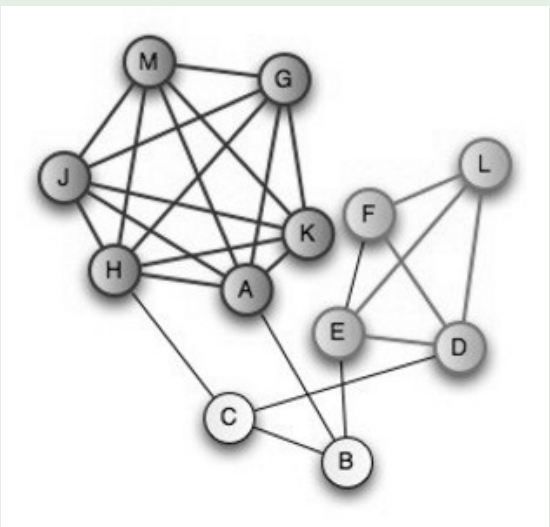
Weitere $\mathcal{NP}$-vollständige Probleme (I)

Cliquen-Problem (CLIQUE)
Eingabe: ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$

Aufgabe: entscheiden, ob G eine „Clique“ mit mindestens k Knoten hat.

„Clique“= vollständiger Untergraph (Kanten zwischen allen Knoten)

Beispiel



Cliquen sind z.B.:
$\{A, G, M, H, J, K\}$
$\{D, E, F, L\}$
$\{B, C\}$

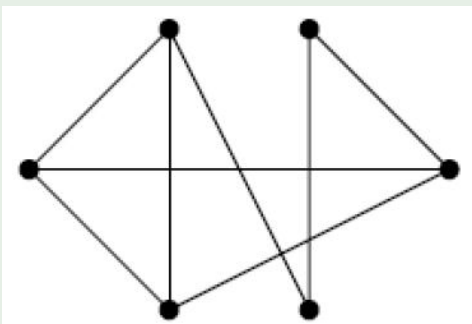
Weitere $\mathcal{NP}$-vollständige Probleme (II)

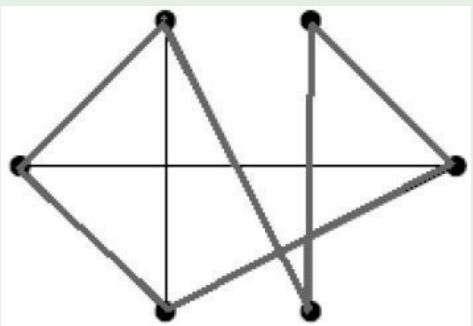
Gerichteter/ungerichteter Hamiltonkreis:

Eingabe: gerichteter/ungerichteter Graph $G = (V, E)$
Aufgabe: entscheiden, ob G einen Hamiltonkreis hat

Hamiltonkreis = geschlossener Weg, der jeden Knoten genau einmal besucht

Beispiel (ungerichteter Hamiltonkreis)





Weitere $\mathcal{NP}$-vollständige Probleme (III)

Das Rucksack-Problem (KNAPSACK):

Eingabe: endliche Menge O von Objekten;
Gewichtsfunktion $g : O \to \mathbb{N}$; Nutzenfunktion $n : O \to \mathbb{N}$;
$S \in \mathbb{N}$ Gewichtsschranke; $K \in \mathbb{N}$ Gewinnschranke

Aufgabe: entscheiden, ob es eine Teilmenge $K \subseteq O$ gibt mit

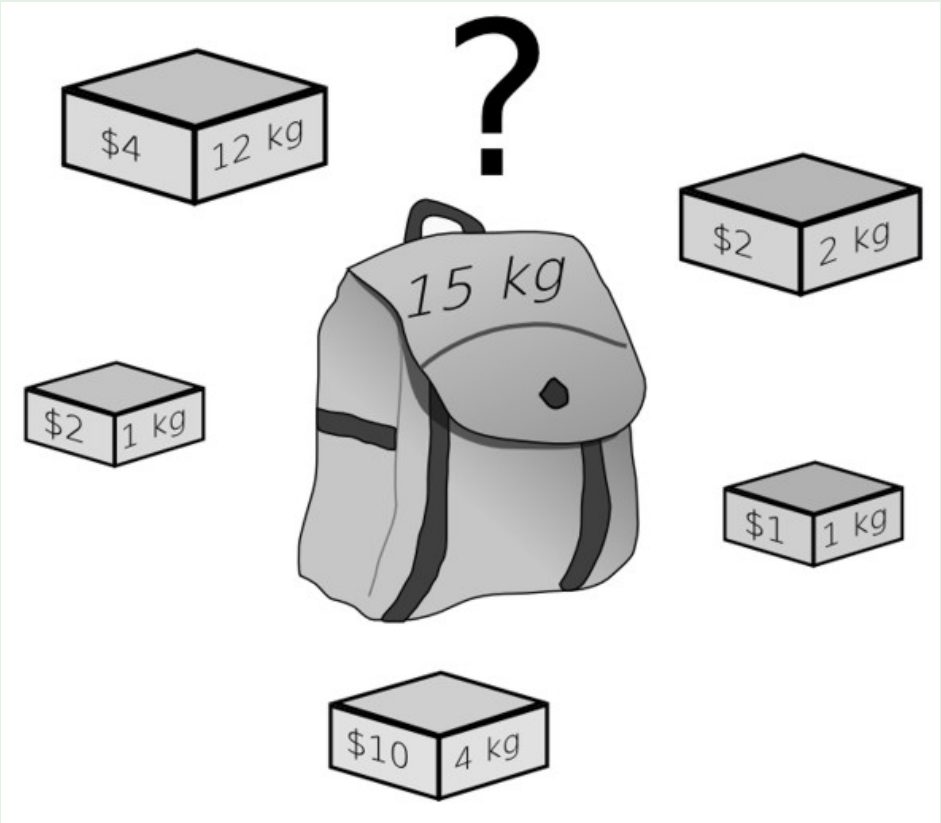
$$\sum_{o \in K} g(o) \leq S \quad \text{und} \quad \sum_{o \in K} n(o) \geq K$$

Anschaulich (eigentlich Optimierungsproblem):

- Gegeben eine Menge von Gütern mit zugeordnetem Gewicht und zugeordnetem Wert und ein Rucksack mit maximalem Transportgewicht.
- Finde eine Bepackung des Rucksacks, die den Gesamtwert der transportierten Güter maximiert.

Weitere $\mathcal{NP}$-vollständige Probleme (IV)

Beispiel



Weitere $\mathcal{NP}$-vollständige Probleme (V)

Das Travelling Salesperson Problem (TSP):
Eingabe: $n \times n$ – Matrix $(M_{i,j})$ und Zahl k
($M_{i,j}$ = „Entfernungen“ zwischen n „Städten“)

Aufgabe: Entscheiden, ob es eine Permutation π (= eine „Rundreise“) gibt, so dass

$$\sum_{i=1}^{n-1} M_{\pi(i),\pi(i+1)} + M_{\pi(n),\pi(1)} \leq k$$

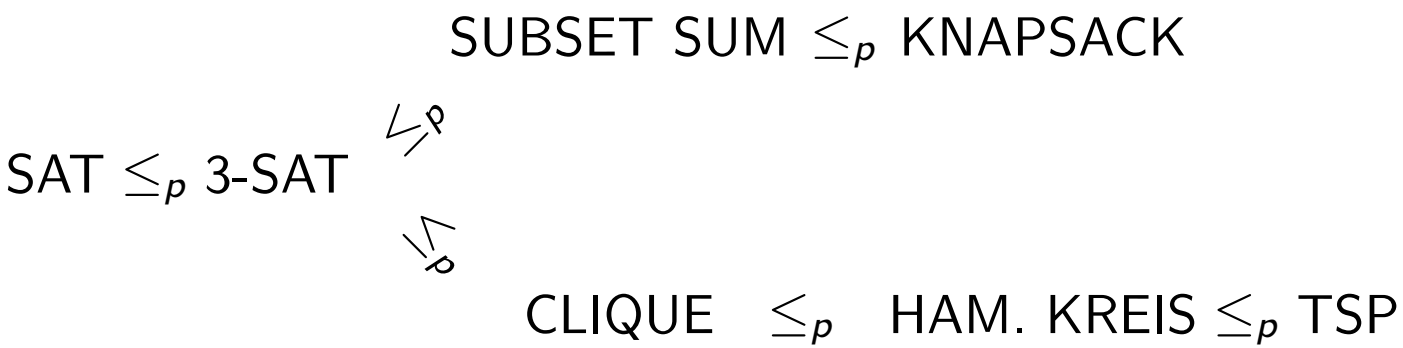
Anschaulich (als Optimierungsproblem):

- Finde für n Städte, für die alle Entfernungen voneinander gegeben sind, die kürzeste „Rundreise“ (die alle Städte besucht).
- Wie kann ein Vertreter n Städte mit möglichst wenig Reisekilometern bereisen?

Typische Reduktionen

Wegen **Cooks Satz**: SAT ist $\mathcal{NP}$-vollständig.

Weitere $\mathcal{NP}$-vollständige Probleme durch Reduktionen:



Kompendium $\mathcal{NP}$-vollständiger Probleme:
`http://www.nada.kth.se/~viggo/problemlist/compendium.html`

Raumkomplexität

- **Raumkomplexität** eines Algorithmus:
  Speicherbedarf des Algorithmus in Abhängigkeit der Größe der Eingabe
- **Bei Turingmaschinen:**

  $$\text{Speicherbedarf} = \text{Bandverbrauch einer Rechnung}$$

  (Gesamtanzahl der verschiedenen Felder des Bandes, auf denen der Schreib-/Lesekopf in der Rechnung zu stehen kommt.)

Raumkomplexitätsklassen

Analog zu den Zeitkomplexitätsklassen:

Definition

Die **Komplexitätsklassen** $\mathcal{P}$SPACE und $\mathcal{NP}$SPACE sind definiert durch:

$$\mathcal{P}\text{SPACE} = \{L \mid \text{es gibt } k \in \mathbb{N} \text{ und eine DTM } M \text{ mit Speicherbedarf in } O(n^k), \text{ die } L \text{ entscheidet } \}.$$

$$\mathcal{NP}\text{SPACE} = \{L \mid \text{es gibt } k \in \mathbb{N} \text{ und eine NTM } M \text{ mit Speicherbedarf in } O(n^k), \text{ die } L \text{ entscheidet } \}.$$

Hinweis: Die Zeitkomplexität eines Algorithmus ist *immer* größer oder gleich der Raumkomplexität.

Denn: Schreiben einer Speicherzelle benötigt jeweils einen Rechenschritt.

Satz 10.7 (Satz von Savitch)

Jede NTM mit polynomieller Raumkomplexität kann von einer DTM mit polynomieller Raumkomplexität simuliert werden; d.h.:

$$\mathcal{P}\text{SPACE} = \mathcal{NP}\text{SPACE}.$$

Zusammenfassung

Anhang

O-Notation

Beispiele

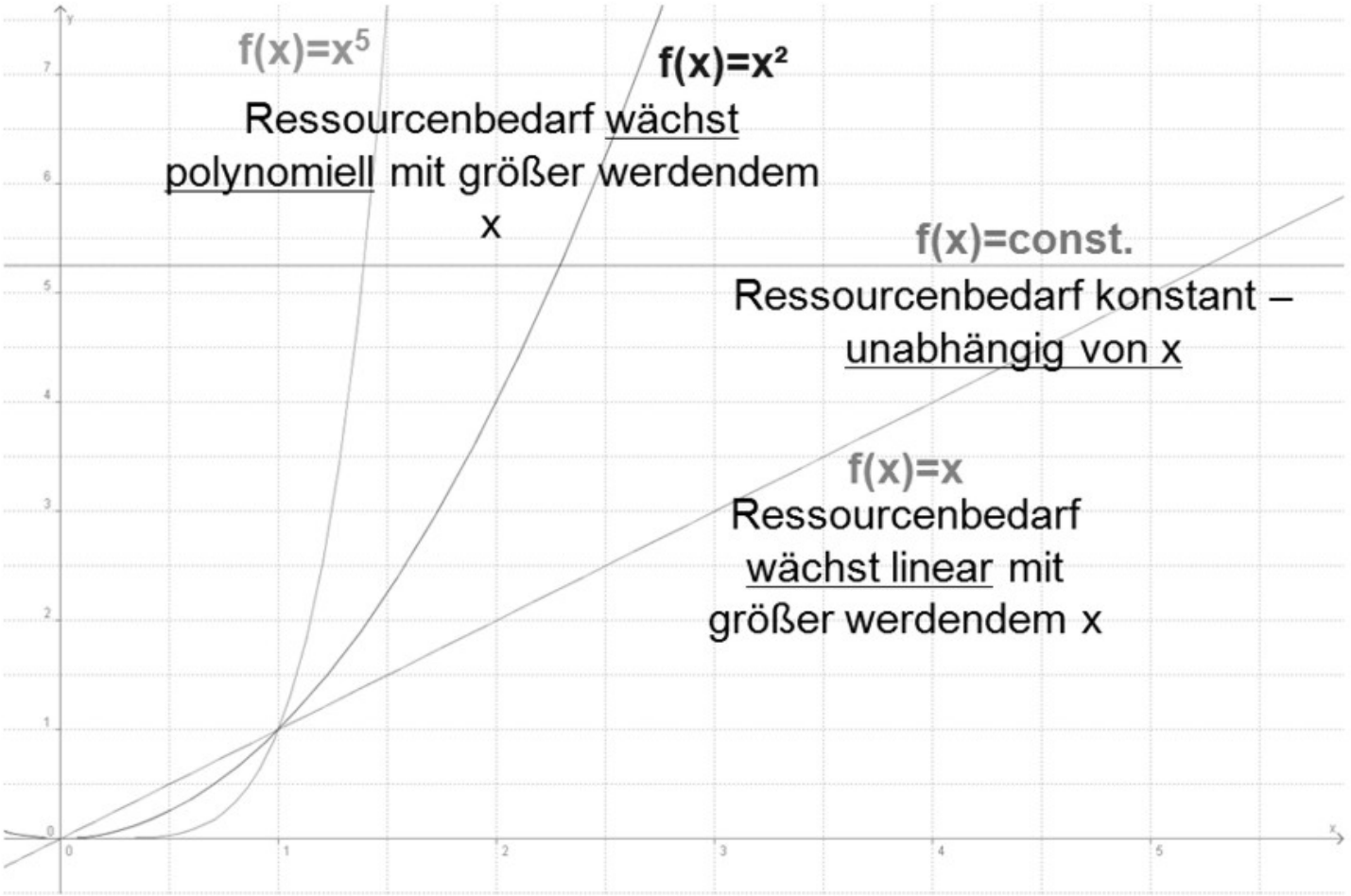
- $f_1(n) = n \in O(n)$
- $f_2(n) = 100n \in O(n)$
- $f_3(n) = 2n + 5 \in O(n)$
- $f_4(n) = n^2 \in O(n^2)$
- $f_5(n) = n^2 + 3n + 8 \in O(n^2)$
- $f_6(n) = n^3 \in O(n^3)$

Bemerkungen:

- Natürlich gilt auch $f_1 \in O(n^2)$, $f_4 \in O(2^n)$ und $f_6 \in O(2^n)$, man kann auch immer „großzügigere“ Asymptoten angeben. In der Praxis interessiert man sich jedoch nur für die kleinstmögliche Asymptote.
- Wir interessieren uns nur für das Verhalten für große n: für f_6 gilt z. B. $n^3 > 2^n$, falls $n \leq 10$.

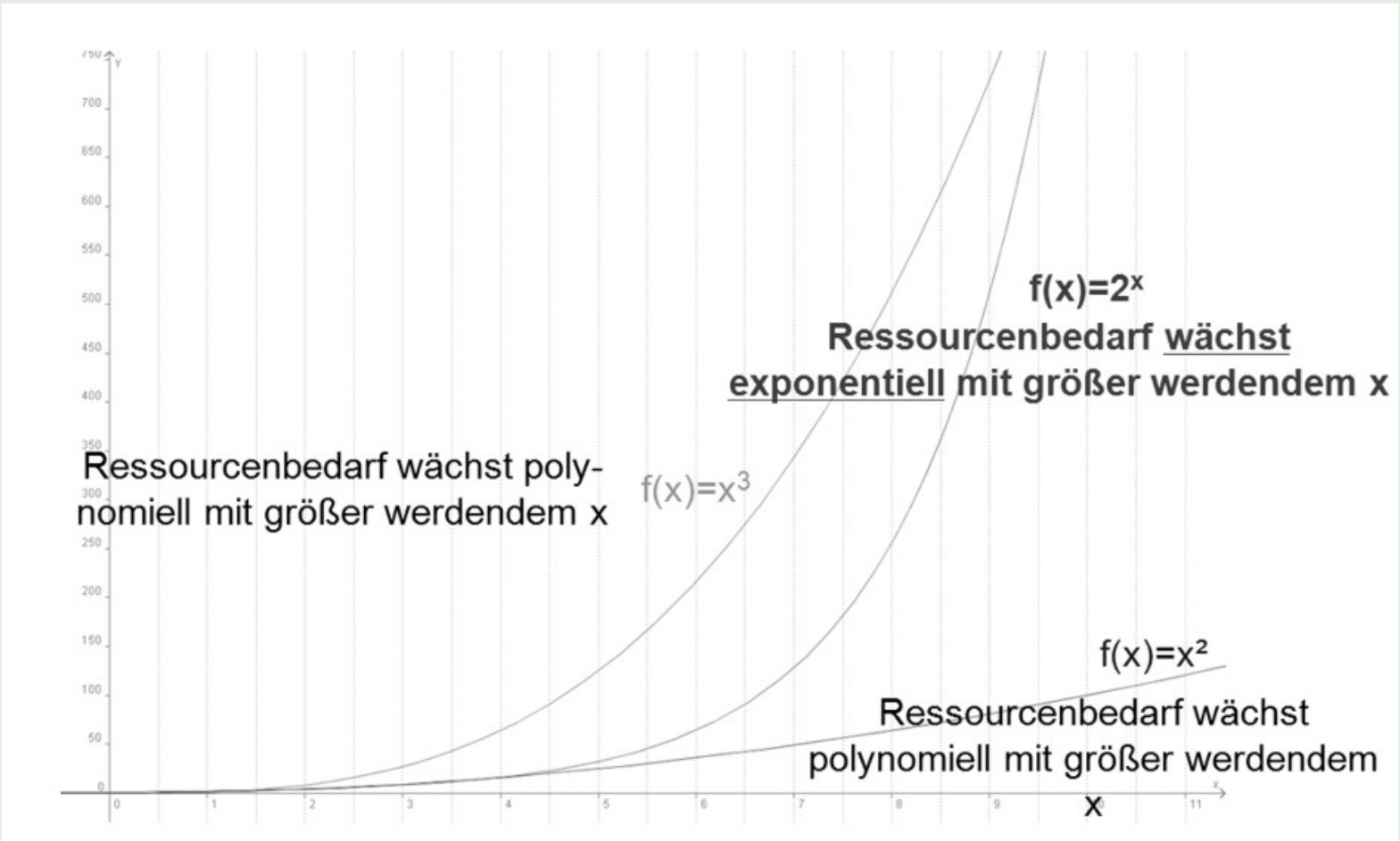
Wachstum von Funktionen (I)

Beispiel



Wachstum von Funktionen (II)

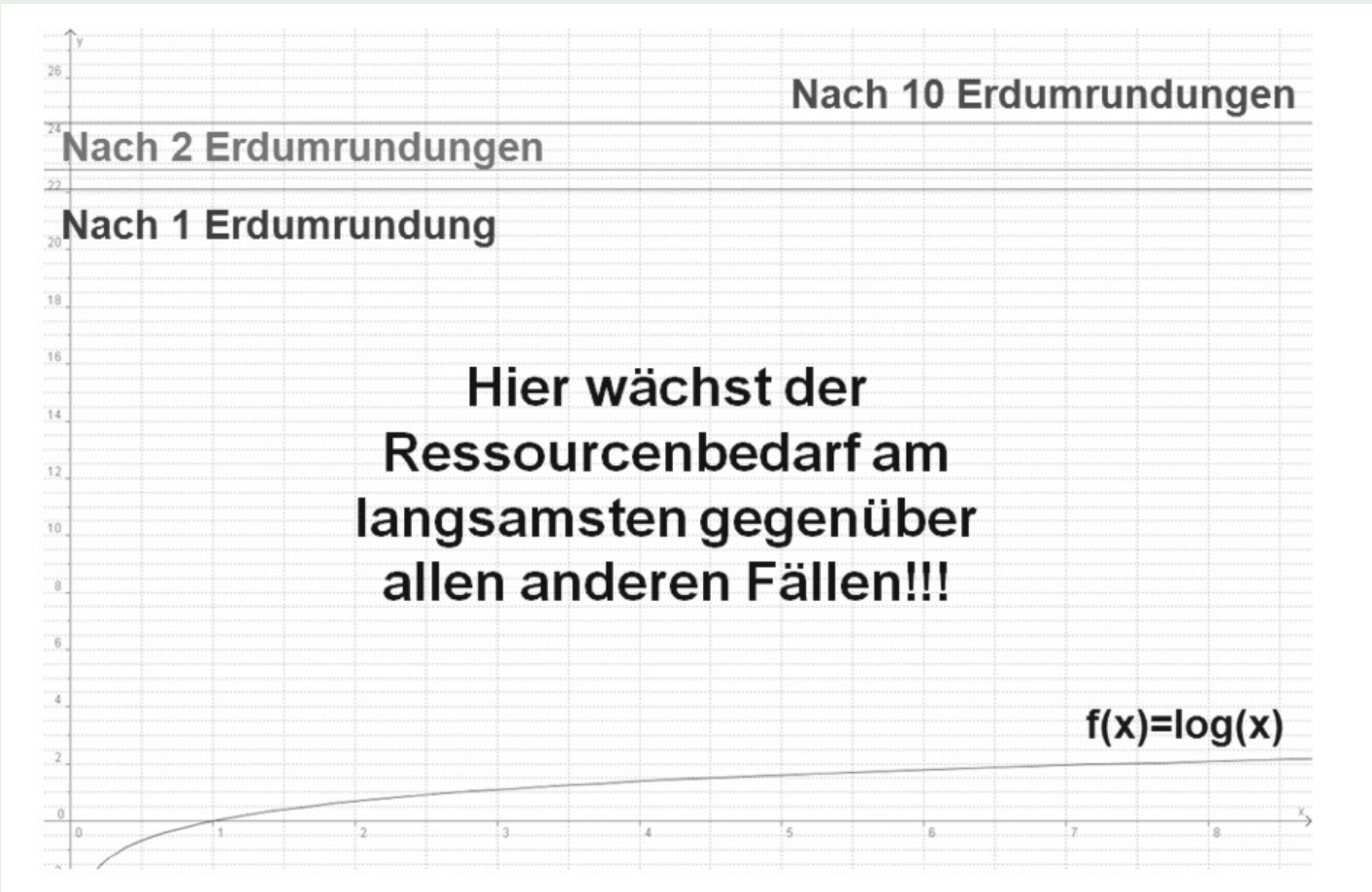
Beispiel



Wachstum von Funktionen (III)

Beispiel

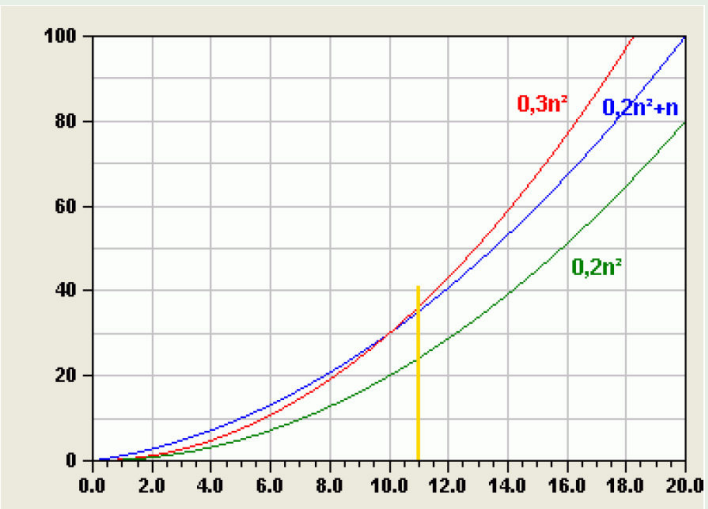
Es seien die x-Achse und $f(x)$ „um die Erde gewickelt“:



O-Notation (IV)

Beispiel

- Es sei $g(n) = 0.2n^2 + n$.
- Wähle $f(n) = n^2$ und $c = 0.3$.
- Für alle $n \geq 11$ ist dann $g(n) \leq c \cdot f(n)$, d. h. das Wachstum von $g(n)$ lässt sich abschätzen durch $O(n^2)$.
- Also: $g(n) \in O(n^2)$ bzw. $g(n) = O(n^2)$.



O-Notation (V)

Satz A10.1

Seien $f, g, h : \mathbb{N} \to \mathbb{N}$ Funktionen, dann gilt:

a) $f \in O(f)$

b) $d \cdot f \in O(f)$ für $d \in \mathbb{N}$

c) $f + g \in O(f)$, falls $g \in O(f)$

d) $f + g \in O(max\{f, g\})$

e) $f \cdot g \in O(f \cdot h)$, falls $g \in O(h)$

O-Notation (VI)

Beweis (von A10.1 d)): $f + g \in O(max\{f, g\}))$

$max\{f, g\}$ ist definiert durch:

$$max\{f, g\}(n) = \begin{cases} f(n), & \text{falls } f(n) \geq g(n) \\ g(n), & \text{sonst.} \end{cases}$$

Hieraus folgt:

$$\begin{aligned} f(n) + g(n) &\leq max\{f, g\}(n) + max\{f, g\}(n) \text{ für alle } n \geq 0 \\ &= 2 \cdot max\{f, g\}(n) \text{ für alle } n \geq 0 \end{aligned}$$

Somit gilt: $f + g \in O(max\{f, g\})$. $\square$

O-Notation (VII)

Beweis (von A10.1 e)): $f \cdot g \in O(f \cdot h)$, falls $g \in O(h)$

Aus $g \in O(h)$ folgt, dass es ein $c > 0$ und ein $n_0 \in \mathbb{N}$ gibt mit

$$g(n) \leq c \cdot h(n)$$

für alle $n \geq n_0$. Daraus folgt:

$$\begin{aligned} f(n) \cdot g(n) &\leq f(n) \cdot c \cdot h(n) \text{ für alle } n \geq n_0 \\ &= c \cdot f(n) \cdot h(n) \text{ für alle } n \geq n_0 \end{aligned}$$

Somit gilt: $f \cdot g \in O(f \cdot h)$. $\square$

Bezeichungen von Laufzeiten

Man spricht von

- linearer Laufzeit: $O(n)$
- logarithmischer Laufzeit: $O(log\ n)$
- quadratischer Laufzeit: $O(n^2)$
- kubischer Laufzeit: $O(n^3)$
- polynomieller Laufzeit: $O(n^k)$ für eine Konstante k bzw. $O(p(n))$ für ein Polynom p
- exponentieller Laufzeit: $O(2^n)$ bzw. $O(q^n)$ für $q > 1$

Polynomielles vs. exponentielles Wachstum

Satz A10.2

Seien $k \in \mathbb{N}, q \in \mathbb{R}$ mit $q > 1$. Dann gilt $n^k \in O(q^n)$.

Das bedeutet, dass jede Exponentialfunktion (mit Basis größer als 1) mindestens so stark wächst wie jedes Polynom (Exponentialfunktionen wachsen sogar echt stärker als Polynome).

Aufwandsänderung bei größeren Eingaben

Wird der Parameter n verdoppelt, so gilt:

- $O(1)$: keine Aufwandssteigerung
- $O(log\ n)$: konstante Aufwandssteigerung
  $(log\ 2n = log\ 2 + log\ n)$
- $O(n)$: Verdoppelung des Aufwandes $(2n)$
- $O(n\ log\ n)$: ca. Verdoppelung des Aufwandes
  $(2n\ log\ 2n = 2n\ log\ n + 2log\ 2 \cdot n)$
- $O(n^2)$: Vervierfachung des Aufwandes $((2n)^2 = 4n^2)$
- $O(k^n)$: Quadrierung des Aufwandes $(k^{2n} = (k^n)^2)$

Verarbeitbare Eingaben verschiedener Algorithmen bei festen Ausführungszeiten (I)

- Frage: Eingaben welcher Größe n können Algorithmen A_i mit den zugeordneten Laufzeiten in einer Sekunde, in einer Minute und in einer Stunde verarbeiten?
- Annahme: Ausführungsdauer (elementarer) Schritt (Zuweisung, arithmetische Operation, Vergleich) sei eine Millisekunde (10^{-3} s)

Algorithmus	Laufzeit	1 s	1 min	1 h
A_1	$O(n)$	1000	$6 \cdot 10^4$	$3.6 \cdot 10^6$
A_2	$O(n \log n)$	128	4615	$2 \cdot 10^5$
A_3	$O(n^2)$	31	244	1897
A_4	$O(n^3)$	10	39	153
A_5	$O(2^n)$	9	15	21

Verarbeitbare Eingaben verschiedener Algorithmen bei festen Ausführungszeiten (II)

- Beobachtungen:
  - A_4 benötigt also bei einer Eingabe w mit $|w| \leq 153$ höchstens eine Stunde Laufzeit.
  - A_5 kann in dieser Zeit nur Eingaben verarbeiten, die nicht länger als 21 sind.
  - Selbst in einer Zeit, die dem angenommen Alter unseres Universums entspricht, verarbeitet A_5 keine Eingaben, die länger als 70 sind.

- Nehmen wir im Folgenden an, wir könnten die Zeit für die Ausführung eines elementaren Schrittes auf ein Tausendstel des obigen Wertes verkürzen (also auf 10^{-6} s), z. B. durch schnellere Hardware.
- Die folgende Tabelle zeigt die möglichen Steigerungen der maximalen Eingabelängen l_i für die Algorithmen $A_i, 1 \leq i \leq 5$:

Verarbeitbare Eingaben verschiedener Algorithmen bei festen Ausführungszeiten (III)

Algorithmus	Laufzeit	Länge vorher	Länge nachher
A_1	$O(n)$	l_1	$1000 l_1$
A_2	$O(n \log n)$	l_2	$\approx 128 l_2$
A_3	$O(n^2)$	l_3	$\approx 31 l_3$
A_4	$O(n^3)$	l_4	$\approx 10 l_4$
A_5	$O(2^n)$	l_5	$10 + l_5$

Man sieht also, dass es sich mehr lohnt, schnellere Algorithmen zu finden (falls möglich), als die Hardware zu verbessern.